THE UNIVERSITY OF CHICAGO

INVESTIGATING STUDENT USAGE OF METACOGNITIVE PROBLEM-SOLVING STRATEGIES IN ALGORITHM DESIGN PROBLEMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
JONATHAN LIU

CHICAGO, ILLINOIS
AUGUST 2026

To the village it took to raise me the past five years.

"Perhaps I can summarize all of these remarks by saying that *methods are more important than facts*. The educational value of a problem given to a student depends mostly on how often the thought processes that are invoked to solve it will be helpful in later situations. It has little to do with how useful the answer to the problem may be."

– Donald Knuth, *Are Toy Problems Useful?*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This manuscript is an accumulation of the work from the central thread of my PhD, studying student development of metacognitive strategies in algorithms courses. Much of the content in this manuscript is work that has been now been completed and submitted to peer-reviewed venues, though the final work is in submission. Relevant information about each chapter is listed below.

Chapter 2: Systematic Literature Review. This work is from "Teaching Algorithm Design: A Literature Review" [Liu et al. (2025b)], published in TOCE 2025 with Seth Poulsen, Erica Goodwin, Hongxuan Chen, Grace Williams, Yael Gertner, and Diana Franklin.

Chapter 3: Characterizing Baseline Strategy Adoption. This work is from "Student Utilization of Metacognitive Strategies in Solving Dynamic Programming Problems" [Liu et al. (2025a)], published in SIGCSE TS 2025 with Erica Goodwin and Diana Franklin.

Chapter 4: Relating Strategy Usage to Course Outcomes. This work is from "Analogical Reasoning in Undergraduate Algorithms" [Liu et al. (2026)], accepted for publication in SIGCSE TS 2026, written with Erica Goodwin and Diana Franklin.

Chapter 5: Investigating Strategy Introduction Methods. This work is in submission, though the presentation is expanded upon in this manuscript. The work was conducted with Hongxuan Chen, Yael Gertner, Mike Shindler, Jeff Erickson, and Diana Franklin.

ABSTRACT

Algorithms is a core course in the undergraduate computer science curriculum and is known both for its relative difficulty and its career implications. A key learning goal in the course is algorithm design, but the problem-solving skills related to the task of algorithm design are nebulous and typically left implicit in algorithms courses and materials. Prior results hint at the possibility that students are learning the content material itself but not the metacognitive problem-solving skills necessary to generalize what they've learned to new problems, but much is left unstudied.

To address this, we present a series of work aimed at answering foundational questions about metacognitive problem-solving skills in algorithms courses. To begin, we conduct a systematic literature review mapping the landscape of algorithms education research, finding broadly that the literature is sparse. In our second work, we aim to create a baseline by characterizing students' natural strategy development in algorithms courses, finding that students are learning important metacognitive strategies on their own but the adoption is inconsistent and improper use of the strategies often hinders student progress. With this in mind, we aim next to demonstrate the potential for metacognitive strategies to benefit students. We conduct a study investigating analogical reasoning, a core metacognitive problem-solving strategy, and present evidence that analogical reasoning is very closely tied to success in algorithm design. These findings justify the importance of explicit modeling and instruction about metacognitive strategies in algorithms classrooms. Finally, we to begin developing pedagogical approaches for explicitly teaching metacognitive strategies, we conduct a randomized control trial investigating how different levels of scaffolded instruction affect adoption of a well-established strategy, finding both clear evidence the scaffolding can be helpful and some weaker evidence that over-scaffolding is also possible.

CHAPTER 1

INTRODUCTION + THESIS OVERVIEW

The study of algorithms is widely regarded as a critical part of an undergraduate Computer Science (CS) degree [Kumar et al. (2024)] and has clear practical applications, especially in software-related careers. Furthermore, algorithm skills often play a large part in job interviews for new graduates attempting to obtain a job after completing their computing degree [Behroozi et al. (2019)]. However, despite its importance and the topic being traditionally regarded as difficult [Enström and Kann (2017)], algorithms is under-represented in the computer science education literature, so little is known about learning and pedagogy surrounding the course.

This is particularly problematic for the core learning goal of algorithm design. Skill development is by nature more difficult to teach than content knowledge, especially by experts [Clark et al. (2008)], but most well-established resources for algorithms (textbooks, online sources) tend to be content focused and describe the different algorithmic paradigms and example problems for each paradigm. While this content is important, it is also vital for educators to consider what related problem-solving skills are necessary to succeed in applying these concepts to new problems and whether they are being developed properly during the algorithms course. Skill development can occur independently through practice, but the targeted introduction of metacognitive strategies can not only expedite this development but also improve self-regulation and confidence related to the task [Vaughn et al. (2011); Donker et al. (2014)]. In this work, we demonstrate that students are only partially developing these skills in class, further emphasizing the importance of research into the proper introduction of these strategies.

In this dissertation, we present of a series of investigations that establish a foundation for improving algorithms instruction through the explicit teaching of metacognitive strategies.

1.1 Overview of Dissertation

This dissertation consists of a series of works that establishes the importance of metacognitive problem-solving strategies in algorithms courses and makes a first step towards understanding how these strategies can be effectively taught. Here, we outline the work that comprises the dissertation.

Chapter 2: Systematic Literature Review. In order to understand the sparse existing work in the space, we conduct a systematic literature review across all major CS Education venues, looking for any research related to algorithm design. Within the corpus, the few papers related to algorithm design point to a mismatch between student concept understanding and student ability to apply these concepts to solve problems, hinting at the need for metacognitive strategies to properly orient students.

Chapter 3: Characterizing Baseline Strategy Adoption. To establish a baseline characterization of the adoption of metacognitive strategies among students in an algorithms course, we conduct a thinkaloud study analyzing strategy usage among students solving Dynamic Programming problems. Some scripted guidance invoking a couple of well-established metacognitive strategies was given when appropriate. We find that students were largely utilizing these strategies while working, but their adoption was often hindered by misuse of the strategy. These results indicate that existing instruction may be insufficient for encouraging proper strategy usage. The following chapters continue investigation into two major strategies and their related findings: some students referenced previously-seen problems but chose takeaways that were inapplicable to the new problem (Chapter 4), and some students could name problem-solving steps to follow but did not understand them (Chapter 5).

Chapter 4: Relating Strategy Usage to Course Outcomes. Of the strategies identified in Chapter 3, the skill of relating problems to previously-seen ones, known as analogical reasoning, stands out both for its broad potential utility and the frequency with which students were misguided by poorly-constructed analogies. Previous work outside of

CS has repeatedly established the importance of analogical reasoning for problem-solving, but related work in CS has only demonstrated that novices often fail to establish good analogies. In this chapter, we present work aiming to properly establish the importance of the skill. We investigate the relationship between student success in designing an algorithm and their ability to relate it to a previously-seen problem, and find that students who created structural analogies were significantly more likely to succeed at the algorithm design task.

Chapter 5: Investigating Strategy Introduction Methods. The metacognitive strategy of problem-solving steps has been shown in the CS1 context to help orient students in the problem-solving process, but our results from Chapter 3 reveal that students may be learning the step names without understanding their meanings enough to be useful. In this chapter, we conduct work investigating different scaffolding methods for the introduction of problem-solving steps for an algorithmic paradigm. Specifically, we seek to understand the extent to which increased scaffolding surrounding the steps affects student learning quality and rate. We propose two different levels of scaffolding, and find that while scaffolding around the core design steps of the strategy are helpful for students, additional implementation-level scaffolding shows no evidence of benefit for students, and hints at over-scaffolding.

CHAPTER 2

SYSTEMATIC LITERATURE REVIEW

This chapter's work is published with the following citation: Jonathan Liu, Seth Poulsen, Erica Goodwin, Hongxuan Chen, Grace Williams, Yael Gertner, and Diana Franklin. 2025. Teaching Algorithm Design: A Literature Review. *ACM Trans. Comput. Educ.* 25, 2, Article 17 (May 2025), 20 pages. doi:10.1145/3727987. Used with permission.

2.1 Introduction

Though the design of rudimentary algorithms has been extensively researched in an introductory programming context, most students will take a more in-depth algorithms-focused course later in their degree, as suggested by the ACM curricular guidelines [Kumar et al. (2024)]. Research on algorithm design instruction, as taught in upper-level algorithms courses, is critically underrepresented in the literature. As such, many open questions remain about student experiences and difficulties in learning this material, as well as pedagogical practices for teaching algorithm design in an effective manner.

In this chapter, we present a systematic literature mapping and review regarding algorithms education, specifically as found in upper-level college courses, to understand with greater clarity both where knowledge exists and where open questions remain. Our work aims to answer the following research questions:

- **RQ1:** Of the topics commonly taught in algorithms courses, how are they currently represented in the literature, and what kinds of studies are being conducted on these topics?
- **RQ2:** What are the major takeaways from the literature about undergraduate algorithm design courses and course topics?

The main contributions of this work are to:

- Characterize the current literature with respect to methods, topic, and authorship,
- Synthesize the existing knowledge about teaching algorithms, and
- Identify opportunities for future research.

2.2 Theory and Related Work

2.2.1 Systematic Literature Reviews

Systematic literature reviews (SLR) are an important part of pushing a research field forward. According to Cooper (1988), as a field expands, a synthesis of existing knowledge pertaining to the topic area becomes increasingly necessary to help researchers stay up to date with current developments, especially in topics related to one's primary specialty. Furthermore, according to Kitchenham and Charters (2007), an SLR can help researchers know what is and is not known in the field as they plan for future studies, and can provide a framework through which any new discourse on the topic can contribute.

Cooper (1988) also provides a taxonomy of six major characteristics through which SLRs can be classified. We list the six characteristics below, along with the appropriate categories for this review, to help situate our contribution.

- **Focus:** Our review primarily concerns itself with *research methods* and *research outcomes*.
- **Goal:** Our review aims to *synthesize prior literature* and *identify issues central to the field*.
- **Perspective:** Our review takes the role of *neutral representation*, aiming to describe what is discussed by the literature without arguing for a specific point of view.

- **Coverage:** Our review's coverage of papers is *exhaustive with selective citation*. Although we believe our review includes most of the literature on the topic, some of the papers are not cited in this review. That said, a list of all papers reviewed and their corresponding tags is publicly available.

- **Organization:** Our review organizes papers *conceptually*, both by topic studied and by results.

- **Audience:** Our review is intended for *specialized and general researchers*, as well as *practitioners*. CS Education researchers may use this review to understand the topic and situate new research, whereas practitioners who teach algorithms or related courses may use this review to discover interventions or insights for use in their own courses.

2.2.2 Literature Reviews in CS Education

SLR papers have appeared regularly in computing education venues in recent years, with reviews being conducted on topics like CS1 [Luxton-Reilly et al. (2018); Becker and Quille (2019)], Parsons problems [Ericson et al. (2022); Du et al. (2020)], undergraduate teaching assistants [Mirza et al. (2019)], meta-cognition in programming [Prather et al. (2020)], use of theory in computing education research [Malmi et al. (2019)], and demographic data reporting [Oleson et al. (2022)]. To our knowledge, there have been no literature reviews seeking to understand the state of algorithms education.

Methodologically, two reviews very similar to ours are by Sheard et al. [Sheard et al. (2009)], who use Simon's classification scheme [Simon (2007a,b)] to present a detailed quantitative landscape of programming education papers with some important outcomes listed, and by Švábenský et al. (2020), who quantitatively extract and present topics, methods, evaluation, impact, and contributors from cybersecurity education papers, analyzing trends of these characteristics without much focus on findings. These SLRs place a larger emphasis

on the focus and approach of existing research in the area. We take a similar approach because the relative sparsity of work means that findings are difficult to synthesize but the methods and topics explored by these papers can still be used to guide future work.

Topic-wise, the closest systematic reviews are on algorithm visualizations and their effectiveness [Shaffer et al. (2007, 2010); Hundhausen et al. (2002)], one important subarea of algorithms instruction. While there is some overlap, these reviews cover a largely different set of literature than the literature reviewed in this paper. Both studies cover algorithm visualization of traditional algorithms (e.g. greedy programming). However, algorithm visualization reviews include topics that we do not include, such as using algorithm visualizations to learn $\mathcal{O}(n^2)$ sorting algorithms or basic data structure traversals, as they are typically taught prior to algorithms courses [Joint Task Force on Computing Curricula and Society (2013); Luu et al. (2023)]. In addition, our review covers non-visualization interventions and studies that seek to better understand student experiences in learning algorithms without providing an intervention. In fact, such non-visual areas make up a large proportion of the studies incorporated in this review, though visualization tools for algorithms in traditional algorithms courses are still covered in the review.

2.3 Methods

To investigate our research questions, we conducted a systematic literature review, following guidelines from Kitchenham and Charters (2007) and Randolph (2009). Sections 3.1 and 3.2 describe the systematic collection of the corpus of literature used for the study. Section 3.3 describes how the literature was characterized to answer RQ1. Section 3.4 describes the consolidation of results from the corpus to answer RQ2.

2.3.1 Search Strategy

To capture our research questions with an appropriate query, we first identified the set of topics that could be considered appropriate for an algorithms course. Within the "Algorithms and Complexity (AL)" knowledge area proposed by the 2013 ACM Curricular Guidelines [Joint Task Force on Computing Curricula and Society (2013)], the sections *AL/Algorithmic Strategies* and *AL/Fundamental Data Structures and Algorithms* relate directly to our research questions. Our preliminary list of topics consisted of the core topics listed in these two sections. Luu et al. find that algorithms courses vary widely in topics covered Luu et al. (2023), so to maintain a reasonable scope for our review, we removed topics that most schools teach prior to their algorithms course as found by Luu et al. (2023), such as sequential and binary search algorithms.

Notably, we decided to include $\mathcal{O}(n \log n)$ *Sorting Algorithms* but not $\mathcal{O}(n^2)$ Sorting Algorithms. First, from a content perspective, though sorting algorithms are typically taught prior to a traditional algorithms course, mergesort and quicksort are canonical examples of the Divide-and-Conquer paradigm [Dasgupta et al. (2006); Erickson (2019)], so the relevant $\mathcal{O}(n \log n)$ *Sorting Algorithms* as per the ACM Curricular Guidelines [Kumar et al. (2024)] are being taught in traditional algorithms courses. Second, from a methodological perspective, our cutoff for the *AL/Fundamental Data Structures and Algorithms* knowledge area was set at topics being taught more often in prerequisite courses than in algorithms courses as per Luu et al. (2023), which supports keeping $\mathcal{O}(n \log n)$ Sorting Algorithms but omitting $\mathcal{O}(n^2)$ Sorting Algorithms.

The 2013 Curricular Guidelines were used because this work was started prior to the release of the 2023 ACM Curricular Guidelines [Kumar et al. (2024)]. However, while the 2023 version renames "Algorithms and Complexity (AL)" to "Algorithmic Foundations (AL)," much is the same - it also contains *AL/Algorithmic Strategies* and *AL/Fundamental Data Structures and Algorithms* sections, and the topics have a lot of overlap - the primary differ-

ences are that the section *AL/Fundamental Data Structures and Algorithms* explicitly lists more data structures than before (e.g., stacks, queues, linked lists), which is out of scope for our review, and that the *AL/Algorithmic Strategies* knowledge area of the 2023 guidelines also include the subtopics of "Decrease-and-Conquer" (e.g., insertion sort, binary search, Euclid's algorithm) and "Iteration vs recursion (e.g., factorial, tree search)," both of which are commonly taught prior to a dedicated algorithms course.

For each topic remaining, we designed and validated a query to capture the corpus of literature on the topic. For some topics, this was straightforward: *Greedy algorithms* was captured by the query "greedy". For others, this required more care: for *Reduction: transform-and-conquer*, "reduction" was too common to be a reasonable query, whereas "transform-and-conquer" was too specific, but the query {"algorithm reduction" OR "reduction algorithm"} captures papers relevant to our interests.Queries were validated by doing a manual search of recent SIGCSE TS proceedings for papers related to our topics and ensuring that the queries captured them. Included topics and their corresponding queries are shown in Table 2.1. Queries were combined into one large query to allow the database to perform de-duplication for us, so the final query began: {"branch and bound" OR ("brute force" AND algorithm) OR ...}.

With this set of queries, we conducted four searches. First, within the ACM Digital Library (DL)[for Computing Machinery (2024)], we searched for all *research articles* from venues sponsored by or in collaboration with SIGCSE (374 unique entries). Papers from SIGCSE TS and ITiCSE between 1970 and 2008 are classified as *articles* in the DL instead because are listed in the SIGCSE Bulletin, so we searched those as well (454 entries). We then repeated the search for research articles in *TOCE* (56 entries). Finally, we used the same set of queries to search the journal *Computer Science Education* in *Taylor and Francis Online* [Online (2024)] (62 entries). From this set of 946 papers, we removed papers that were retracted or otherwise were not accompanied with a PDF in the respective database.

We note that this literature search still omits some venues. This decision was made in large part due to consistency in search mechanism and capacity constraints. We recognize that venues like the Sage *Journal of Educational Computing Research* and the CCSC regional conferences, among others, provide important contributions to the CS Education discourse, and we encourage exploration of these venues for work related to our topic. That said, the set of venues included is consistent with other CS Education literature reviews, as found by Heckman et al. (2021).

Table 2.1: Topics and Queries Used

Topic	ACM Digital Library Query
Branch-and-bound	"branch and bound"
Brute-force algorithms	"brute force" AND algorithm
Depth- and breadth-first traversals	"breadth first search" OR "depth first search" OR "breadth first traversal" OR "depth first traversal"
Divide-and-conquer	"divide and conquer"
Dynamic programming	"dynamic programming"
Greedy algorithms	"greedy"
Heuristics	"heuristic algorithms"
Minimum spanning tree	"minimum spanning tree" OR "prim's algorithm" OR "kruskal's algorithm"
Pattern matching and string/text algorithms	"string algorithm" OR "text algorithm" OR "substring matching" OR "regular expression matching" OR "longest common subsequence"
Recursive backtracking	"recursive backtracking"
Reduction: transform-and-conquer	"algorithm reduction" OR "reduction algorithm"
Representations of graphs	"adjacency list" OR "adjacency matrix"
Shortest-path algorithms	"shortest path algorithm" OR "dijkstra's algorithm" OR "floyd's algorithm"
$\mathcal{O}(n \log n)$ Sorting algorithms	"quick sort" OR "quicksort" OR "heap sort" OR "heapsort" OR "merge sort" OR "mergesort"

2.3.2 Study Selection

We then applied three further exclusion criteria on the remaining papers. First, we aimed to omit papers not relevant to our search, but variation in algorithms course content made this difficult. For example, only around 60% of institutions teach Depth-First Search/Breadth-First Search (DFS/BFS) in their algorithms courses [Luu et al. (2023)]. To ensure that no relevant papers were excluded, we kept any paper that states involvement in an "algorithms course" as well as any paper explicitly teaching any of our topics (see Table 2.1). For example, a paper about a CS2 (Data Structures) course in general would be omitted, but a paper that describes teaching DFS/BFS in CS2 would be kept.

Papers were also only kept if they presented some form of data from students. Either quantitative or qualitative data was accepted, but papers that describe an intervention or tool but do not provide evaluation data were omitted. Finally, papers were only kept if the evaluation was on post-secondary-level students in order to maintain some consistency in course context.

The exclusion criteria were applied by four of the listed authors, who independently filtered a subset of 20 papers, discussed to reach a consensus, and continued. After the authors filtered the second subset, Fleiss' Kappa statistic [Fleiss (1971)] was used to compute the inter-rater agreement for four raters, and the authors achieved Fleiss' Kappa $\kappa > 0.87$, so the rest of the papers were processed independently. In the end, 102 papers were identified as relevant to our literature review.

2.3.3 Paper Tagging

The authors then developed a set of tags for describing and classifying the relevant papers.

Codebook Development

A Content Analysis protocol [Drisko and Maschi (2016)] was used to develop a codebook with appropriate categories for our review. We began with a draft codebook with deductive codes and descriptions that all coders agreed upon. We then iteratively improved the codebook by repeating the following steps:

1. Each coder independently tags 10 papers according to the codebook.
2. Any disagreements in coding are resolved through discussion.
3. The codebook is adjusted to reflect the new shared understanding of the coders. This may involve adding or removing codes, or adjusting the descriptions.

At the end of the third cycle, no changes were made to the codebook. Fleiss' Kappa statistic was used to calculate reliability with a fixed number of raters on categorical items [Fleiss (1971)], and the authors achieved Fleiss' Kappa $\kappa > 0.80$ inter-rater agreement on this batch, so the remaining papers were tagged independently.

Final Codebook

The finalized codebook's codes were split into four major sections, described here. Within each section, the list of possible tags does not exhaustively consider every possibility for a paper but does capture every paper found in our review.

Topic: Each paper was tagged by the topic or topics from the ACM curricular guidelines it focused on, and an extra *Not topic-specific* tag was added for papers that studied algorithms pedagogy as a whole without focusing on a specific topic. Our codebook specifies that topic tags are reserved for when the *Results* section of the paper explicitly describes the topic, so that whole-course interventions are tagged as *Not topic-specific* rather than with every individual topic taught in the course.

Evaluation: This set of tags aims to describe the data presented by the paper. These tags include *Quantitative Data* and *Qualitative Data* as broad categories, but also data gathering methods (e.g. *Survey*, *Interview*) and data analysis methods (e.g. *Statistical Tests*, *Qualitative Coding*). Most of the tags could be discovered in either the Methods or Results sections of a paper, but *Qualitative Coding* was sometimes unclear because qualitative data could be presented in thematic groupings without the data undergoing a formal thematic coding process. As such, our codebook focuses instead on the reported methodology - papers were only tagged with *Qualitative Coding* if, alongside the analysis of qualitative results, their analysis procedure was described explicitly in the methods section.

Note also that papers could and often did contain multiple of these tags and that some tags even imply others. For example, if a paper was tagged with *Qualitative Coding*, then it was also tagged with *Qualitative Data*.

Metric: This set of tags aims to determine the student metric measured by the paper. We found all papers presented data on at least one of *Performance*, *Affect*, or *Persistence* (whether students remain enrolled in the course).

Intervention: This set of tags aims to capture information about the intervention developed in the paper, if applicable. We categorized interventions based on the aspect of the course targeted, and introduced a *Tools* tag because some interventions developed a tool without directly affecting the course. These were not mutually exclusive — for example, a tool may be developed to help students during discussion. A full list and description of intervention tags can be found in Table 2.2.

All tags can be seen in Table 2.3.

2.3.4 Results Consolidation

To identify major takeaways from the corpus, the papers were split based on whether they presented an intervention. For papers containing interventions, the first and third authors

Table 2.2: Tagging Scheme to Characterize Interventions

Tag	Description
None	A study that does not change the way students learn the material, and instead investigates the status quo, (e.g. misconceptions or student behavior).
Tools	A study that develops and/or evaluates a tool that a student can interact and interface with, especially a software tool, that aids with student learning.
Course Policy	A study that changes course logistics unrelated to content (e.g. late policy or collaboration policy).
Content Presentation	A study that changes how the students are presented non-interactive course material (e.g. lecture).
Discussion/Lab	A study that changes staff-facilitated problem-solving sessions (e.g. labs and group work).
Student Work	A study that changes non-staff-facilitated work done by students (e.g. homework or projects).
Exams	A study that changes how students are evaluated in test settings.

read each paper, recording the results of the paper as well as any features of the intervention explicitly described in the paper as beneficial. These results and features were consolidated and inductively coded by the authors.

Papers not containing an intervention were instead inductively coded based on focus. The first and third author independently read each paper and created a summary of the findings, focused especially on the paper's description of the findings in the abstract, introduction, and discussion. Summaries were then consolidated and inductively coded by the authors. The most common theme found was *Student Approaches to Algorithm Design Problems* - see Section 2.5.1.

The final codes can be found as the headers in Section 2.5.

2.4 Corpus Survey

In this section, we describe the results of the tagging, as described in Section 3.3, in order to answer RQ1 by characterizing the space of literature regarding the topics commonly taught

in algorithms courses. The results of the tagging can be found in Table 2.3. Note that a single paper can be tagged in multiple categories and topics, so it can be represented in multiple rows and columns. The last row contains a summary of the results. The fully tagged corpus can be found in this citation Liu et al. ([n. d.]).

We begin by presenting a broad survey of the methodology used in the corpus. We then drill down to answer more specific questions about topic coverage, changes over time, "rigor" (which we define later), and the venues and authors that publish the most. Findings are accompanied by a discussion about implications and future steps.

2.4.1 Corpus Methodology

Nearly every paper (92/102) meeting our selection criteria presents quantitative data, and 73% (74/102) of papers use student scores and grades. Notably, around 40% (43/102) of the papers run statistical tests, and only around 18% (18/102) run controlled trials. On the other hand, slightly less than half (47/102) of the papers use qualitative data. Thematic coding, found in only 16% (16/102) of papers, was done for interview responses (e.g. [Toma and Vahrenhold (2018)]), student work (e.g. [Velázquez-Iturbide (2012)]), and survey responses (e.g. [Weber (2023)]). In 38% (39/102) of the papers, both quantitative and qualitative data are presented.

In terms of evaluation metrics, 73% (74/102) of papers use student performance, 62% (63/102) use affective measures, and 34% (35/102) of papers use both. Of the four papers measuring persistence [Gárcia-Mateos and Fernández-Alemán (2009); Deb et al. (2017); Lin et al. (2021); Coffey (2013)], all four measure performance as well, and three measure affect, so it may be worthwhile to conduct studies more centered around persistence in the algorithms class.

Most papers (76%, 78/102) present interventions, and around 40% (37/102) center around a tool. For example, five papers present a strategy for implementing active learning in the

Table 2.3: Tags by Topic. Note that some papers cover multiple topics, and contribute to each corresponding row in the table.

		Evaluation							Metric			Intervention						
Topic	Total	Quantitative	Survey	Stat. Tests	Control Trial	Qualitative	Interview	Thm. Coding	Performance	Affect	Persistence	None	Tools	Course Policy	Content	Discussion	Student Work	Exams
Branch-and-bound	1	1	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0
Brute-force Algorithms	2	2	1	2	2	0	0	0	2	1	1	0	2	0	0	1	1	0
DFS/BFS	7	6	5	3	3	3	1	1	4	5	0	0	3	0	2	1	3	0
Divide-and-Conquer	4	4	1	3	1	2	1	0	4	1	0	2	1	0	0	1	0	1
Dynamic Programming	12	11	6	6	3	6	3	2	10	6	1	4	4	0	1	1	4	1
Greedy Algorithms	8	8	4	0	0	6	2	5	6	4	0	2	3	0	3	2	2	0
Heuristics	3	3	0	0	0	2	0	2	3	0	0	1	1	0	0	0	2	0
Minimum Spanning Trees	6	6	5	2	2	2	0	0	4	4	1	0	4	0	0	2	4	1
Pattern Matching and String/Text Algorithms	1	1	1	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0
Recursive Backtracking	3	3	2	1	1	2	0	1	3	2	0	1	2	0	1	1	0	0
Reductions	7	5	3	1	0	5	1	2	7	2	0	4	0	1	2	1	2	0
Representations of Graphs	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Shortest-path Algorithms	7	7	5	3	3	1	0	0	6	5	1	0	7	0	1	2	2	0
$\mathcal{O}(n \log n)$ Sorting Algorithms	10	9	7	6	2	4	1	0	6	9	0	2	3	0	2	2	3	0
Total (any topic)	51	46	30	18	11	25	7	9	40	30	2	12	21	1	10	11	15	2
Not topic-specific	51	46	29	25	7	22	6	7	34	33	2	12	16	11	11	7	16	2
Total	102	92	59	43	18	47	13	16	74	63	4	24	37	12	21	18	31	4

classroom via interactive question-and-answer tools [Pargas (2006); Anderson et al. (2007); Kurtz et al. (2014); Phan and Hicks (2018); Deb et al. (2018)]. On the other hand, around a quarter (24/102) of the papers surveyed do not develop an intervention. Many of these papers analyze student problem-solving strategies when presented with algorithm design problems, and a number of others study topic-specific misconceptions.

We only find four interventions related to student examinations. Given the broad applicability of algorithms to test questions and the effect that question design and test logistics can have on exam validity, we expect research on this subject to be potentially useful.

2.4.2 Representation of Topics

Of the 102 papers included in our literature review, half focus on a specific topic or set of topics, whereas the other half present results that do not pertain to any particular topic, such as studies about broad problem-solving skills or policy-based interventions.

Most topic-specific papers only focus on one topic, though thirteen papers focus on two topics, two were tagged with three topics each [Taylor et al. (2009); Brown et al. (2022)], and one paper was tagged with four (*Branch-and-Bound*, *Greedy Algorithms*, *Heuristics*, *Recursive Backtracking*) [Velázquez-Iturbide (2019)]. However, many areas are still understudied. In this section, we begin by discussing trends in individual topics and then compare topic-specific papers with those that are not topic-specific.

Topics with More Coverage

Research about the topic of **Dynamic Programming** has taken a relatively wide variety of approaches. Two papers contribute interesting course projects that motivate the use of Dynamic Programming through various problems, one from a centuries-old mathematical text [Pengelley et al. (2006)] and the other from the modern world [Bird and Curran (2006)]. Tools have also been developed to help scaffold different steps of solving DP problems, specifically interpreting the problem constraints [Lin et al. (2021)], visualizing subproblem recurrences [Velázquez-Iturbide et al. (2016); Velázquez-Iturbide and Pérez-Carrasco (2016)], and writing a proof [Xia and Zilles (2023)]. Research has been conducted [Danielsiek et al. (2012); Paul and Vahrenhold (2013)] and reproduced [Zehra et al. (2018); Shindler et al. (2022)] investigating common roadblocks in solving DP problems as well, finding that stu-

dents either don't recognize that DP is the correct approach, fail to split the problem into appropriate subproblems, or have trouble identifying the correct recursion of subproblems. Finally, Enström and Kann [Enström and Kann (2017)] find that teaching the DP process as separate steps can increase student self-efficacy.

On the other hand, the 8 papers about **Greedy Algorithms** have markedly less breadth, with one paper about misconceptions [Velázquez-Iturbide (2019)], one about collective argumentation [Kallia et al. (2022)], and the other six about interactive learning interventions. Though five provide thematic coding of student responses, none provide statistical tests in their evaluations, limiting our understanding of intervention effectiveness.

At 10 papers, $\mathcal{O}(\mathbf{n \log n})$ **Sorting Algorithms** is another topic commonly covered in the literature, though we expect that $\mathcal{O}(n^2)$ algorithms are studied far more thoroughly. Most of these papers leverage sorting as a familiar task to develop engaging active-learning interventions for introducing faster sorting algorithms, including through games [Hakulinen (2011); Hosseini et al. (2019)] and exploratory activities [Rasala et al. (1994); Rebelsky (2005); McCann (2004)], though a couple instead use assessment results to provide insight into common misconceptions [Winters and Payne (2006); Taherkhani et al. (2012)].

Topics with Less Coverage

We found no papers related to **Graph Representations** and one related to **Pattern-Matching and String/Text Algorithms** [Bird and Curran (2006)], though the latter may be partially captured in research about Dynamic Programming and Greedy Algorithms. Despite **Divide-and-Conquer**, **Minimum Spanning Trees (MST)**, and **Shortest-path Algorithms** being taught in more than 80% of algorithms courses [Luu et al. (2023)], the topics are only studied by 4, 6, and 7 papers, respectively. Of the four Divide-and-Conquer papers, three detail a multi-study project by Danielsiek, Paul, and Vahrenhold on misconception detection for a wider set of topics that includes Divide-and-Conquer [Danielsiek et al.

(2012); Paul and Vahrenhold (2013); Vahrenhold and Paul (2014)], while the fifth involves a tool for learning in the classroom [Phan and Hicks (2018)], so questions remain about presenting, testing, and assigning work on the topic. Five of the six MST papers relate to the development of student assignments [Deb et al. (2017); Taylor et al. (2009); Erkan (2018); Stasko (1997); Hansen et al. (2003)], and the other paper studies exam questions [Gaber et al. (2023)], leaving the presentation of the material or common student misconceptions unexplored. All seven Shortest-path papers are tool-based interventions, leaving plenty of room for exploration of misconceptions or pedagogical strategies.

Papers Without Specific Topics

Within non-topic-specific papers (half of our set), there is considerable variation in the extent to which the knowledge extracted applies specifically to algorithms courses. For example, some papers test interventions in an algorithms course but could have reasonably tested them in any CS course (e.g. [Belleville et al. (2020); Pargas (2006)]), some studies include algorithms courses alongside other CS courses in their data set (e.g. [Deb et al. (2018); Bennedssen and Caspersen (2008)]), and some studies are topical to algorithms in particular (e.g. [Danielsiek et al. (2017); Collberg et al. (2004)]).

For the most part, these non-specific studies mirror the topic-specific papers in terms of evaluation and intervention methods used. The most notable difference is in *Course Policy*, where all but one paper focuses on the course as a whole rather than individual topics. This is unsurprising, as we expect course policy changes to affect all topics in a course equally, though Crescenzi et al. [Crescenzi et al. (2013)] study a sizeable modification to course structure targeted at the course's NP-completeness unit.

That said, the studies are still notably different in their aims. For example, while most topic-specific papers without interventions focus on identifying misconceptions for the topic, papers without specific topics instead focus on broader measurements like skill development

[Ginat (2001)], learning outcomes [Bennedssen and Caspersen (2008)], and even student affect [Danielsiek et al. (2017)]. Likewise, while most content-specific tools presented are related to algorithm visualization, papers without specific topics feature not only visualization tools (e.g. [Naps et al. (2000); Lee and Rößling (2011)]) but also tools for facilitating improvements in grading [Gárcia-Mateos and Fernández-Alemán (2009); Coore and Fokum (2019)], lectures [Pargas (2006); Kurtz et al. (2014)], and assignments [Marshall et al. (2006); Zeller (2000)].

2.4.3 Topics over Time

In Figure 2.1, we display the distribution of topics covered by year. Sorting Algorithms is the most-studied topic early on, representing 6 of 14 topic-specific papers between 1994-2008. In every four-year period, at least half of the papers within our study relate to either Sorting, DFS/BFS, Greedy Algorithms, or Dynamic Programming.

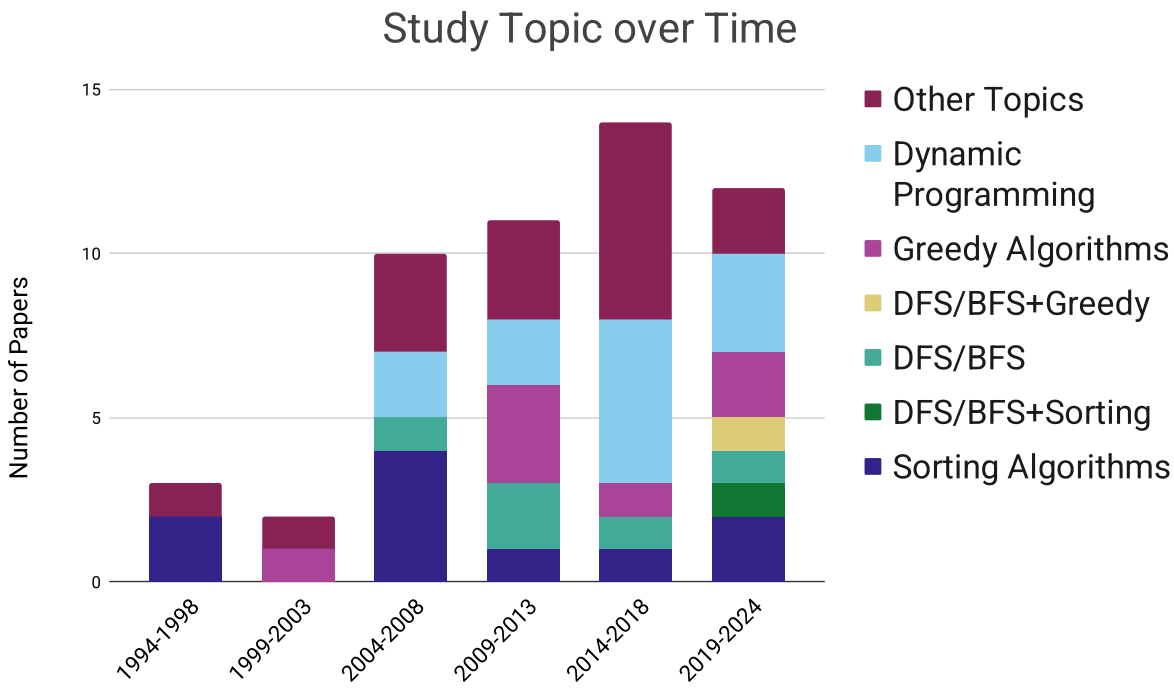


Figure 2.1: Topic of topic-specific papers, by year

2.4.4 Rigor over Time

Scientific rigor is by nature tough to quantify, as the best practices for any paper depend on the specific research questions. For papers with quantitative data, the use of *statistical tests* or *controlled trials* serves as our proxy for a metric of rigor [Sanders et al. (2019)]. Rather than try to quantify the rigor of qualitative methodologies, we look at whether papers report their analysis procedure for coding the qualitative data at all, employing replicability as a looser but more measurable standard for rigor. These papers are captured by the *Qualitative Coding* tag - See Section 2.3.3.

Note that papers with no evaluation or student data were already filtered out of our review. Figure 2.2 shows the percentage of each type of study, presented in 5-year bands.

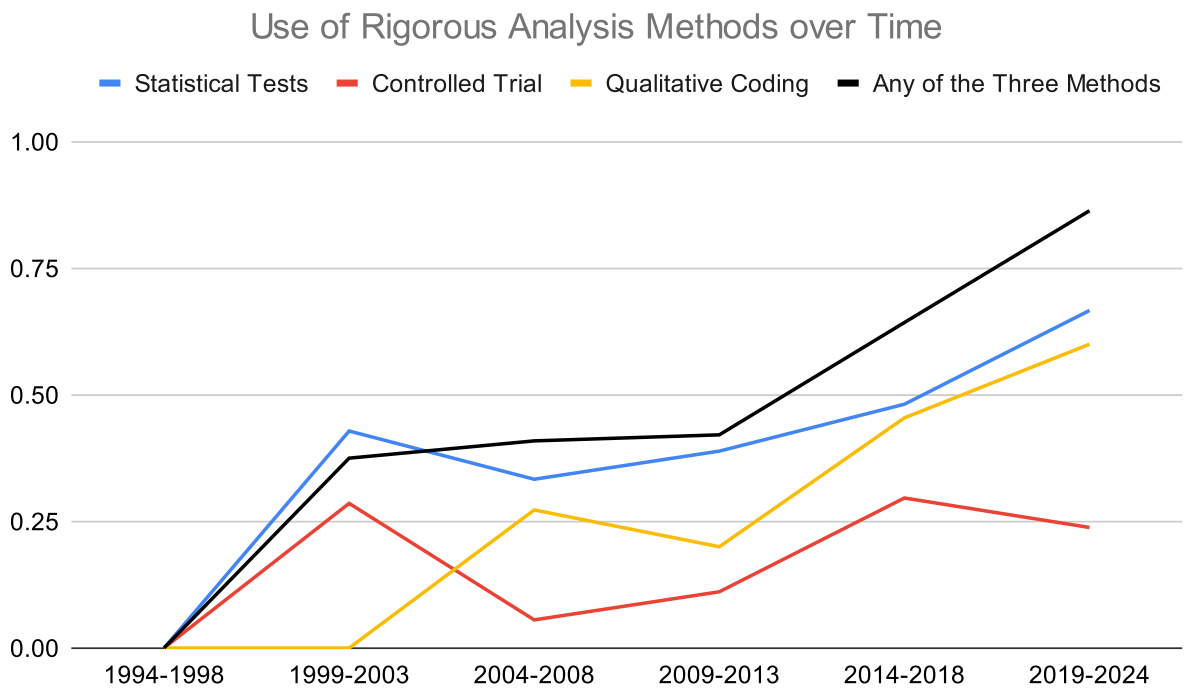


Figure 2.2: Percentage of papers with statistical tests (quantitative), controlled trials (quantitative), thematic coding (qualitative), and at least one of the above

We see no "rigorous" studies initially, but there is an encouraging upward trend in the usage of rigorous procedures in algorithms papers as the field matures. In the most recent

period, 2019-2023, 60% of qualitative papers use thematic coding, and two thirds of quantitative papers use either statistical tests or controlled trials. Note that a paper may present both quantitative and qualitative data but only rigorously analyze one of those forms of data.

Comparatively, Sanders et al. find that 51% of ICER papers between 2005-2018 used inferential statistics [Sanders et al. (2019)] and Švábenský et al. [Švábenský et al. (2020)] find that 54 papers about cybersecurity education between 2010 to 2019 report descriptive statistics while only 23 (43%) use inferential statistics, but these studies do not provide information about historical trends.

2.4.5 Where and Whom are Studies Coming From?

Our search consisted of all publications sponsored by or in collaboration with SIGCSE, as well as *TOCE* and *CSE*. The distributions of papers per conference, split by year, can be found in Figure 2.3.

As expected, SIGCSE's largest conferences, the Technical Symposium and *ITiCSE*, account for a vast majority of the papers (44 and 32, respectively) in our literature review, with *ICER* contributing more recently. We found seven relevant papers from *CSE* and five relevant papers from *TOCE* (and its previous moniker *JERIC*) as well.

We also seek to understand whether much of the work is accomplished by just a few groups or whether it is widespread in the community. The tally of papers per author can be found in Table 2.4.

We can see that the most active groups are two groups of collaborators. Velázquez-Iturbide / Pérez-Carrasco have studied optimization algorithms and developed visualization tools to assist with a variety of algorithm paradigms (e.g. Velázquez-Iturbide (2012); Velázquez-Iturbide et al. (2016); Velázquez-Iturbide (2019); Velázquez-Iturbide and Pérez-Carrasco (2016)) and Vahrenhold / Danielsiek / Paul have explored instruments for mea-

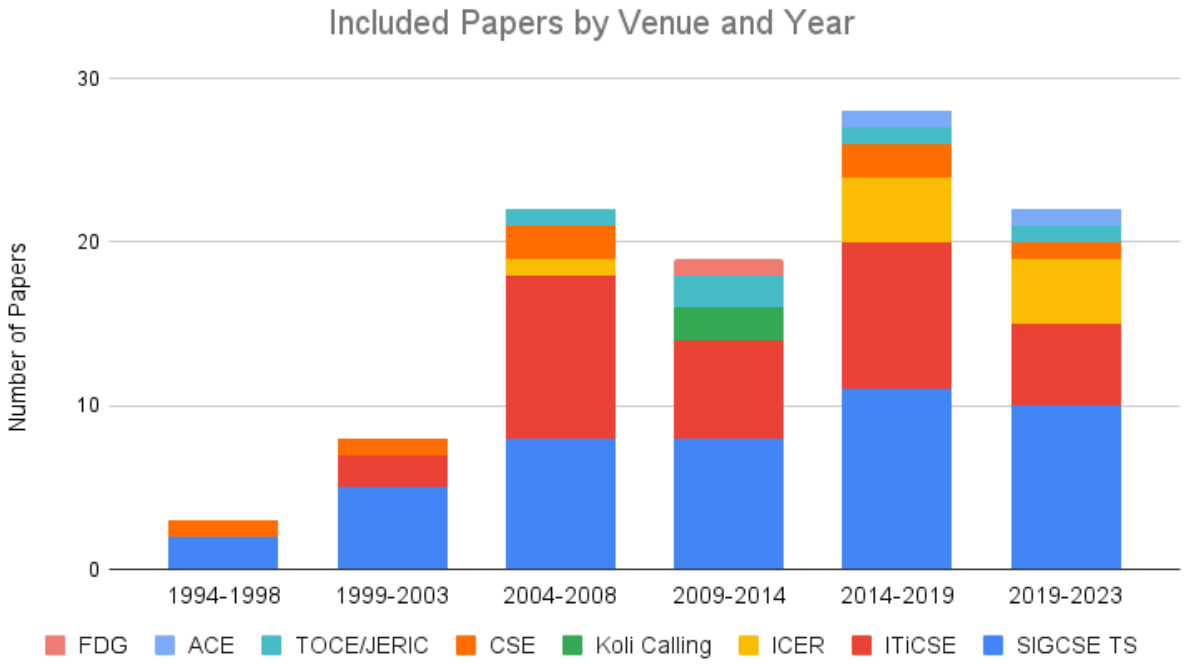


Figure 2.3: Papers per venue, by year

suring student affect and success in algorithms courses (e.g. Danielsiek et al. (2012, 2017); Paul and Vahrenhold (2013); Toma and Vahrenhold (2018)). In addition, David Ginat has written four papers discussing metacognitive strategies for algorithmic problem-solving (e.g. Ginat (2001); Ginat and Blau (2017)), and Michal Armoni has conducted a series of studies on reductive thinking Armoni (2009).

The vast majority of authors publish only one paper (205/251, 82%), hinting that few specialize in the area. Though we do not have data about how this compares to the CSEd literature as a whole, this mirrors the result by Švábenský et al. who find that 87% (175/202) of authors who have published on cybersecurity education have only published one paper on the topic Švábenský et al. (2020).

Table 2.4: Number of Papers per Author. Asterisks indicate frequent collaborators.

Author	Number of Publications
J. Ángel Velázquez-Iturbide*	7
Jan Vahrenhold**	5
David Ginat	4
Antonio Pérez-Carrasco*	4
Michal Armoni	3
Holger Danielsiek**	3
Wolfgang Paul**	3
[18 authors]	2
[205 authors]	1

2.5 Corpus Takeaways

In this section, we go beyond the numbers to provide insights into the major takeaways from the corpus as a whole. Subsections are broadly sorted in order of decreasing specificity to algorithms courses.

2.5.1 Student Approaches to Algorithm Design Problems

Though the skill of algorithm design is somewhat nebulous, there is general consensus that it is closely tied to abstraction ability. Bennedssen and Caspersen (2008) examine 10 courses mandatory for the CS degree and find that abstraction ability correlates with performance in only the algorithms course. Think-alouds have been run to identify varied extents to which students utilize abstraction to solve problems [Ginat and Blau (2017); Izu et al. (2017)].

More specific algorithm design skills have been studied as well, with the general finding that students under-utilize them even after having taken an algorithms course. In problems where recursion [Ginat (2004)] and reduction [Armoni (2009)] would have been helpful, students understand the concepts but underestimate their utility in general problem-solving contexts. On the other hand, students demonstrate an overreliance on intuitive greedy approaches, even in the face of counterexamples [Ginat (2003)].

At the topic level, a limited number of papers explore common misconceptions in algo-

rithms topics. Danielsiek et al. (2012) use both student work and thinkalouds to cover a wide variety of topics, but other studies have focused on more specific topics like dynamic programming [Zehra et al. (2018)] and optimization algorithms [Velázquez-Iturbide (2019)], and reductions [Gal-Ezer and Trakhtenbrot (2016)].

2.5.2 Problem Selection

We find a recurring theme that intentional problem selection can be leveraged to motivate students. The broad applicability of algorithms lends itself nicely to problems tied to the real world, which can give students both familiarity with the problem setting and motivation to solve it. For example, linking shortest path algorithms to GPS navigation is a setting in which multiple papers demonstrate that students enjoy and engage more with the real-world map [Holdsworth and Lui (2009); Teresco et al. (2018)]. Mehta et al. (2012) presented bipartite matching in the setting of forming balanced class project groups, resulting in both more successful groupings and an increased understanding of bipartite matching. Problems can also tie into the real world through the researcher: Pengelley et al. (2006) presented a Dynamic Programming problem posed by a mathematician's letter from 1838, and Wirth and Bertolacci (2006) presented a problem from the instructor's research, both finding that the humanized context motivated students.

Problems with a variety of solutions can also be particularly helpful. In multiple studies, students indicate that writing both efficient and inefficient solutions to problems and comparing their runtimes on real inputs provided more motivation for designing efficient algorithms than theoretical runtime analyses [McCann (2004); Coffey (2013)]. Similarly, studies have designed projects asking students to compare BFS against DFS [Erkan and Gochev (2008)], Prim's algorithm against Kruskal's algorithm [Erkan (2018)], and several different sorting [Rasala et al. (1994)] and matching [Lucas (2015)] algorithms, finding in all cases that students describe the comparisons as valuable and insightful. In each of these

projects, the comparison is facilitated in part by a visualization of either the evaluation or output of the algorithms.

Another approach that promotes the connections between different algorithms is advocated for by Libeskind-Hadas (2013), who describes a derivation-first teaching strategy that motivates the discovery of classical algorithms through simpler, more intuitive algorithms.

2.5.3 Algorithm Visualization

In our data set, we identified 15 studies about algorithms visualizations. Broadly, students state that they appreciate visualizations, though few studies experimentally measure learning outcomes. Of these papers, three found that introducing visualizations improves student learning [Velázquez-Iturbide (2013); Deb et al. (2017); Farghally et al. (2017)], though only one uses statistical tests. On the other hand, Jarc et al. (2000) find that introducing visualizations provides no statistically significant learning outcomes, and Velázquez-Iturbide and Pérez-Carrasco (2016) find that students are more efficient but do not perform better. Taylor et al. (2009) find that student learning significantly improves when the visualization forces students to take an active, predictive role. For a more thorough review on algorithms visualizations that is not specific to algorithms courses, we recommend the work by Hundhausen et al. (2002) and Shaffer et al. (2010).

2.5.4 Coding Practice

Though around a quarter of algorithms courses have no programming component [Luu et al. (2023)], research has found that algorithmic coding assignments increase student engagement [Gárcia-Mateos and Fernández-Alemán (2009)] and helps students develop their algorithmic design [Izu and Alexander (2018)] and runtime analysis [Coore and Fokum (2019)] skills. Online coding platforms allow for a public leaderboard of coding problem scores, which some studies have used in an attempt to motivate students [Coore and Fokum (2019); Färnqvist

and Heintz (2016)] with mixed results: Coore and Fokum (2019) observed that the strongest students worked hard to maintain their ranking, but other students were demotivated when the top scorers had yet to solve a problem.

2.5.5 Assessment Policy

The use of algorithmic coding problems permits automated assessment, which students find fair and constructive [Färnqvist and Heintz (2016); Weber (2023)], though additional human interpretation of automated feedback may be helpful [Leite and Blanco (2020)]. Weber (2023) finds promise in standards-based grading for algorithms courses, along with allowing students to re-submit work as they continue to master course concepts. While students did continue to improve on past assignments throughout the course, some took advantage of the chance to procrastinate work until the end of the term.

Two papers investigate the utility of two-stage quizzes (consisting of an individual and a group phase), finding that team quizzes increase students' perceived and measured learning gains [Belleville et al. (2020); Ham and Myers (2021)]. Gaber et al. (2023) make the case for repeating previously-seen problems on exams, showing that these questions are still considerably challenging for students yet perceived as fair.

2.5.6 Active Learning

Research on algorithms-based active learning exercises in lecture corroborates findings in other settings that students enjoy exploring a problem space with their peers [Belleville et al. (2020); VanDeGrift (2017); Toma and Vahrenhold (2018)], feel an increased sense of engagement in class [Anderson et al. (2007); Gehringer and Miller (2009)], and report learning both course material and collaboration skills [VanDeGrift (2017)]. Additionally, incorporating in-class interaction into historically lecture-based courses led to higher instructor satisfaction in some cases [Hosseini and Perweiler (2019); Bouchez-Tichadou (2018)].

Most studies incorporate active learning by having students work through problems in class, whether exploring a problem instance [Anderson et al. (2007); Kurtz et al. (2014)], coding an algorithmic solution [Phan and Hicks (2018)], or performing algorithm analysis [Pargas (2006); Deibel (2005)]. Some of these activities were equipped with auto-grading as well, which Deb et al. (2018) find improved student retention of course material. To dissuade a "guess-and-check" approach while using an auto-grader, Kurtz et al. (2014) capped grades based on the number of submissions.

Game-based exercises have also shown promising results, with research finding that these activities can drive participation from students who are typically silent [Hosseini and Perweiler (2019); Hosseini et al. (2019)]. Some of the algorithms-based games developed include a sorting algorithm design contest [Hosseini et al. (2019)], a matching game in which students choose algorithms for given criteria, and a draw-and-guess game in which students illustrate course content [Hakulinen (2011)].

However, despite the success of interactive in-class activities, facilitating productive group work may require further attention: in a study on communication behaviors during a collaborative algorithmic problem solving session, Kallia et al. (2022) observed that first-year undergraduates struggled to build off of each others' ideas while MSc students were overly dismissive of others' ideas. Two papers find success using a variety of intentional group-formation strategies to encourage productive collaboration, making groups based on learning style, prior student knowledge, or mixed instructor/student input [Deibel (2005); Mehta et al. (2012)].

2.5.7 Psychological Factors

Little is known about psychological factors in algorithms courses, but a series of papers by Danielsiek, Toma, and Vahrenhold develop and validate an algorithms-specific instrument to measure self-efficacy [Danielsiek et al. (2017)] and use it along with other instruments to mea-

sure the cognitive load and perceived benefit of different collaborative lab assignments [Toma and Vahrenhold (2018)]. Their work finds that collaborative labs are perceived as effective and inclusive, and demonstrate the potential for using measurements of various emotional responses to guide intervention design. Bennedsen and Caspersen (2008) investigate affective predictors of success in multiple computer science courses and find no correlation between emotional health and exam performance in the algorithms course, though the authors note that their results are generally inconclusive due to small sample size.

2.6 Limitations

Though we designed our search query to capture papers related to algorithm design, some papers may have been missed. First, if a paper does not use our search terms, our query may not capture it. Furthermore, our search was limited to either venues sponsored by or in collaboration with SIGCSE in the ACM DL or in the Taylor & Francis journal *Computer Science Education*, so research published outside of these venues was not included though we expect to have captured most of the work in the area. See Section 2.3.1.

The variability in course offerings across institutions also may have resulted in some omitted papers. Topics that we consider to be relevant to algorithm design are sometimes taught in prerequisite courses like CS2 [Luu et al. (2023)], but we may not be able to tell based on the paper. As such, knowledge about teaching these topics in other courses is missed if the paper does not explicitly mention our topic of interest.

As with any qualitative coding endeavor, we also note that misinterpretations of papers in the review are possible. However, we hope that the content analysis procedure, with many rounds of codebook refinement and discussions about potentially different interpretations of the codebook, mitigated the likelihood of these outcomes.

2.7 Future Work

As discussed, the sparsity of research on algorithms education suggests many areas for future work. As a whole, Table 2.3 reveals many potential unanswered questions about teaching algorithms. Section 2.4.2 discusses potential areas of exploration at a more fine-grained level.

At a broader level, our review finds that research in the area relies heavily on quantitative data, especially measuring student scores or grades, but relatively few studies use statistical tests and even fewer run controlled trials. Though we acknowledge the difficulty of running controlled trials in an educational setting, more rigorous approaches to this data would be valuable to develop a better understanding of the effectiveness of various interventions. As is, though many papers report interventions or assignments with positive outcomes, it is difficult to determine effect size, significance, or the influence of potential confounds like the instructor or classroom environment.

At the same time, this reliance on quantitative data is accompanied by a dearth of qualitative data, especially about psychological factors like sense of belonging or self-efficacy. Rigorous explorations of student experiences in these classes would be useful for not only determining the extent to which the mathematical content of algorithms courses affects psychological factors but also orienting future research around mitigating impostor syndrome, stereotype threat, or other discovered barriers.

While in this review we were able to cover all topics from the ACM Curricular Guidelines sections *AL/Algorithmic Strategies* and *AL/Fundamental Data Structures and Algorithms* which are usually covered in an algorithms course, we were not able to cover topics listed in the related sections *AL/Basic Automata Computability and Complexity* about computing theory and *AL/Basic Analysis* about algorithmic analysis. These topics are also required learning for CS majors according to the ACM Curricular Guidelines, though they are less frequently covered in algorithms courses [Luu et al. (2023)]. Future reviews could use these as topics of focus.

2.8 Conclusion

In an effort to synthesize research related to algorithm design education, we categorized all papers in the ACM Digital Library containing search terms for algorithm design topics covered in algorithms courses, guided by the ACM curricular guidelines and surveys of algorithms course instructors [Joint Task Force on Computing Curricula and Society (2013); Luu et al. (2023)]. We found that algorithms research is a growing field, with more rigorous methods being applied over time. Studies have generally agreed that active learning interventions and interesting problem-selection can contribute to both learning and motivation, corroborating research in the broader CS Education space. However, even among the most studied areas in algorithm design, there are still large gaps in the literature. For some topics, there is adequate literature on student misconceptions, but concrete instructional interventions are lacking. In others, researchers have created interventions, but have not yet rigorously tested their efficacy using controlled trials. Exams and Psychological Factors stand out as important areas with very little research at all. We hope this review can help guide the future of algorithm design education.

CHAPTER 3

CHARACTERIZING BASELINE STRATEGY ADOPTION

This chapter's work is published with the following citation: Jonathan Liu, Erica Goodwin, and Diana Franklin. 2025. Student Utilization of Metacognitive Strategies in Solving Dynamic Programming Problems. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1* (Pittsburgh, PA, USA) (SIGCSE TS 2025). Association for Computing Machinery, New York, NY, USA, 687–693. doi:10.1145/3641554.3701888. Used with permission.

3.1 Introduction

Dynamic Programming (DP) is a quintessential algorithms topic, listed as a core Algorithmic Strategy by the Joint Task Force on Computing Curricula [Kumar et al. (2024)] and covered in more than 80% of college Algorithms courses [Luu et al. (2023)]. DP has seen slightly more research focus than other algorithms topics, as described in Chapter 2, perhaps due to its perception as being one of the most conceptually challenging topics [Enström and Kann (2017)].

However, little work has investigated metacognitive interventions for teaching DP. Prior work has explored common roadblocks experienced by students independently solving DP problems [Danielsiek et al. (2012); Paul and Vahrenhold (2013); Zehra et al. (2018)] and the order in which to provide instruction [Enström and Kann (2017)].

In this paper, we explore student metacognitive approaches to DP. We conduct think-aloud interviews during which students articulate their thinking and receive on-demand semi-structured guidance as they work through DP problems. This guidance is focused on metacognitive approaches to solving the problems. The study is designed to answer the following research questions:

- **RQ1:** What metacognitive approaches do students develop while learning to solve DP problems?
- **RQ2:** To what extent does structured metacognitive guidance benefit students solving DP problems?
- **RQ3:** What obstacles do students face when using metacognitive strategies while solving DP problems?

We find that these strategies generally help students make progress in solving DP problems, but that they can mislead students as well. We also find that the adoption of these strategies is an individualized process and that structured strategy guidance is often insufficient in allowing students to solve individual DP problems, indicating the need for more extensive strategy instruction.

3.2 Background

3.2.1 Metacognition

Metacognition, somewhat colloquially described as "thinking about thinking" [Miller et al. (1970)], refers to a student's self-regulatory knowledge and mechanisms related to their own learning, though its scope and overlap with *self-regulation* can vary [Dinsmore et al. (2008)]. In our context, we operationalize the concept as a student's ability to evaluate their own learning and problem-solving strategies and reuse effective ones. Though these strategies are internally developed by nature, explicit instruction related to metacognitive strategies has proven effective in various fields [Donker et al. (2014)]. The concept is well-established within CS education, demonstrated in part by the review conducted by Loksa et al. (2022) about metacognition in programming education.

Metacognition has been studied to a limited extent within algorithms. Through interviews with students designing algorithms, Izu et al. (2017) identify metacognitive strategies

requiring varying levels of abstraction, and Ginat (2001) describes how a problem-solving process incorporating self-reflection helped students develop metacognitive awareness. Further work exploring teaching metacognitive strategies in algorithms is necessary: three studies find that students often rely on intuitive but incorrect strategies [Ginat (2003, 2004); Armoni (2009)].

Problem-Solving Steps

One method for developing metacognitive problem-solving skills is in-process scaffolding, in which students are explicitly provided with a series of problem-solving steps to help them self-orient and reflect as they think through solving a problem (e.g., [Prather et al. (2019)]). Loksa et al. (2016) presents the following six steps for solving open-ended programming problems:

1. Reinterpret the problem prompt,
2. Search for analogous problems,
3. Search for solutions,
4. Evaluate a potential solution,
5. Implement a solution,
6. Evaluate the implemented solution.

Their work also demonstrates that the presentation of a problem-solving process can improve student independence, self-efficacy, and growth mindset in learning.

As algorithms courses tend to place more weight on the design and evaluation of solutions [Luu et al. (2023)], these steps may not be sufficient for more domain-specific tasks within algorithms courses. The popular algorithms textbook by Cormen et al. (2022) provides a set of explicit steps to design a DP algorithm:

1. Characterize the structure of an optimal solution,

2. Recursively define the value of an optimal solution,

3. Compute the optimal solution value in a bottom-up fashion.

These steps are partially validated by prior works that identify these steps as common places where students falter in solving DP problems (see Section 3.2.2). Within our study, we take inspiration from both of these lists in orienting and guiding students.

Creation of Small Examples

Manual simulation of an algorithm's steps has been shown to help students understand a new or difficult concept, most commonly researched in teaching sorting algorithms [Butler and Ahmed (2016); Battistella et al. (2017)]. The guidance provided in our study relates to this strategy, in which students work through example problem instances while designing an algorithmic approach [Ginat (2001)].

3.2.2 Dynamic Programming (DP)

Research on DP has introduced new tools and investigated the student problem-solving process. A number of papers present tools to help with specific steps of a DP problem, including understanding problem constraints Lin et al. (2021), defining a set of subproblems Xia and Zilles (2023), and visualizing subproblem recurrences Velázquez-Iturbide et al. (2016); Velázquez-Iturbide and Pérez-Carrasco (2016).

Enström and Kann (2017) experimented with teaching DP as two distinct skills: (1) recognizing the subproblem structure and writing a recurrence relation and (2) proving the evaluation order and implementing the algorithm, with students practicing step 2 alone before solving problems in full. Student self-efficacy increased with this phased teaching, though this did not correlate with performance.

A number of papers provide insight into students' conceptions of DP problems. Danielsiek, Paul, and Vahrenhold [Danielsiek et al. (2012); Paul and Vahrenhold (2013)] develop and validate the use of multiple-choice survey items to detect misconceptions about various algorithmic paradigms including DP, and three studies [Danielsiek et al. (2012); Shindler et al. (2022); Zehra et al. (2018)] present think-aloud data from students solving DP problems to capture aspects of the process that cause students trouble. These studies corroborate one another in finding that students often use greedy or brute-force techniques when DP is the correct approach, and that when attempting a DP strategy, students often conflate DP with divide-and-conquer or are unable to define a proper subproblem or recurrence. Our work builds upon the work of these papers in a few ways: by identifying metacognitive strategies utilized, by studying how they relate to the roadblocks identified, and by examining the effect of guidance on student problem-solving.

3.3 Methods

3.3.1 Study Population and Recruitment

This IRB-approved study involved think-aloud interviews in Winter 2024 at a private R1 university in the US. Participants were recruited from the undergraduate Algorithms course through announcements about an opportunity to participate in a study in which they would receive practice and guidance solving DP problems. Students were informed that participation was voluntary and that instructors would not be informed of student participation.

Interviews were conducted in two rounds: one for the two weeks immediately after the course's DP lectures (R1, 19 interviews), and the other for two school days prior to the course final (R2, 11 interviews). Seven participants were interviewed in both R1 and R2. R1 interviews are labeled I1-I19, and R2 interviews are labeled I20-I30, but labels were shuffled within rounds to preserve anonymity.

3.3.2 Think-Aloud Interview Protocol

Each interview was conducted by a member of the research team. Participants were presented with a series of DP problems and asked to narrate their thought process to the best of their ability as they worked through each problem. Participants were informed that the interviewer could provide guidance at any time, but *only upon request*. This delineation was made both to allow the participant to work as independently as possible and to analyze what types of help are requested. Each interview lasted approximately one hour. Two forms of data were collected: a transcript of audio recordings and a scan of the final state of the participant's scratch paper.

The first problem presented to all students was *Coin Row*, in which students are asked to find the highest-value non-adjacent subset of a row of coins [Danielsiek et al. (2012); Zehra et al. (2018)]. Participants solved between 1-3 problems of increasing difficulty in each session. Problems were selected to vary in a number of aspects including subproblem array dimension, data type, and extraction method. Returning participants received similar, but not identical, problems.

During the last few minutes of each interview, participants were asked questions about their experience solving algorithms problems. First-time participants were asked to reflect on the session and identify which resources contributed to their algorithms problem-solving approach; returning participants were asked to reflect on their learning process for DP problems since the first interview.

3.3.3 Provided Guidance

To provide the minimum help necessary, when students requested help, interviewers asked follow-up questions to identify the level of help appropriate, modified from prior work [Franklin et al. (2013)]. Interviewers used their best judgment to select the level of help needed, and often gave multiple levels of help for a problem. Listed below are the levels

of help and corresponding help given. Note that L3 and L4 are guidance strictly about a metacognitive strategy.

L1 – Validation: Reassured their approach was correct.

L2 – Error-Check: Given location of minor computational error.

L3 – Steps: Presented with a list of problem-solving steps, modified from prior work [Loksa et al. (2016); Cormen et al. (2022)] to align with the iterative (bottom-up) approach in class instruction. The steps were not presented in class.

1. Reinterpret the problem prompt,
2. Select a structure for subproblems,
3. Write a recurrence relation between subproblems,
4. Describe the bottom-up evaluation to compute the solution,
5. Describe the base case(s).

L4 – Example: Instructed to create a small example [Butler and Ahmed (2016); Battistella et al. (2017)] and asked a set of generic questions that helped them use the example to complete the step they were on.

L5 – Specific: Received more problem-specific and participant-specific guidance as necessary.

3.3.4 Data Analysis

Codebook Development

A coding scheme was inductively developed [Thomas (2006)] to categorize moments of progress, help requests, and metacognitive strategies used as students worked. The first two authors initially read through four interview transcripts, making note of the ways in

which they show student approaches. These notes became an initial coding scheme that was independently applied by both authors to each of the transcripts; tagging was discussed to resolve differences and amend the codebook as needed.

Phase 1: Organizational Coding

Quotes were divided by quote type, then by problem step and guidance level.

Quote Type: Only quotes relevant to the student's problem-solving process were tagged, categorized as `Work`, `Help Request`, `Guidance`, or `Comment`. The `Work` tag was used for any student quotes illustrative of their process while working on a problem. The tags `Help Request` and `Guidance` were used to capture when a student requested help and when it was provided, and `Comment` was used for student comments about problem-solving that were not specific to the problem they were working on.

Problem Step: All `Work` quotes were categorized into the step of the DP problem they corresponded to according to the steps listed in Section 3.3.3: `Reinterpret`, `Subproblem`, `Recurrence`, `Evaluation`, `Base Case`. Guidance relevant to a specific step was also tagged accordingly. Quotes that described students pursuing a non-DP approach were tagged as `Off-Track`. Note that the `Reinterpret` step is omitted from the results, as students rarely verbalized or needed help with it.

Guidance: Quotes tagged as `Guidance` were further categorized by what level of help was given according to Section 3.3.3.

Phase 2: Descriptive Coding

The first two authors then performed open coding to extract recurring themes of interest.

Metacognitive Strategy: First, the authors inductively classified all `Work` quotes by the metacognitive strategy (if any) employed by the student. Authors identified four strategies students used; they are described in Table 3.1. The `Steps` and `Examples` strategies matched

our provided guidance (L3 and L4, respectively); `Priors` and `Recursion` strategies emerged independently from the data. However, these strategies are not novel: `Priors` is steps (2) and (3) of Loksa's steps [Loksa et al. (2016)] and the recursive paradigm for DP is commonly mentioned as an aside in algorithms textbooks [Cormen et al. (2022); Dasgupta et al. (2006)]. The authors then independently coded all *Work* quotes and resolved disagreements through discussion.

Comments: Open coding was performed on the comments in a similar manner. In this paper we do not discuss emergent themes from this category, but some comments providing insight into metacognition are included in the results.

3.3.5 Limitations

As with all think-aloud studies, participants may not verbalize every thought or strategy, and open coding may be imprecise even with multiple coders. Furthermore, the study was advertised as a guided problem-solving session, so we expect volunteer bias towards students struggling with DP. This may be especially true in R2, as students may be more strategic about their time near finals.

3.4 Results

We begin by presenting overall findings about metacognitive strategy development and usage (RQ1), as well as the extent to which metacognitive guidance was able to help students solve problems (RQ2). Strategy-specific benefits and obstacles to adoption are then presented by strategy (RQ2/RQ3).

3.4.1 Overall Findings

Strategy Usage

In Figure 3.1, we show the percentage of students who used each strategy (Table 3.1). `Independent Usage` occurs without, or prior to, guidance on the strategy and `Guided Usage` indicates usage as a result of prompting based on a request for help. `Partial Usage`, which is specific to `Steps`, captures awareness of the `Subproblem` step but not all steps.

Finding: The strategy used independently by the most students was `Examples`. We find that 25/30 (83%) of students independently created example inputs and outputs to help design DP algorithms.

Finding: Most students knew to start by defining a subproblem, but knowledge of the complete set of steps was limited. While 80% (24/30) knew to define a subproblem, in only 37% (11/30) of the interviews did students independently know all of the steps.

Table 3.1: Metacognitive Strategies Identified

Strategy	Description	Example Quote
Problem-Solving Steps (`Steps`)	The explicit separation of the DP algorithm design process into a series of five steps: `Reinterpret`, `Subproblem`, `Recurrence`, `Evaluation`, `Base Case`.	"Alright, so based off of what I know about dynamic programming problems, *I'd probably try to split it up into subproblems first.*" (I21)
Example Input and Output (`Examples`)	The creation of an example input and output to experiment with in order to gain a better understanding of the problem's structure.	"[My work] would involve I guess just *visualizing it and like manually playing with the little cases...* and that might give us some pattern." (I11)
Prior Problems (`Priors`)	The usage of a previously-seen problem and solution to provide insight into the current problem being solved.	"So I guess my new thought is that you can, for each prize p_i, you can kind of *do what we did in the last problem,* you could figure out the maximum size." (I28)
Recursive Paradigm (`Recursion`)	The attempt to design a memoized recursion (top-down) DP algorithm, as opposed to following the iterative (bottom-up) paradigm for DP algorithms as taught in class.	"We'll take `calc_max(arr,i)`, ... and we'll say that you return the max of `arr[i]+calc_max(arr, i-2)` and `calc_max(arr, i-1)`." (I5)

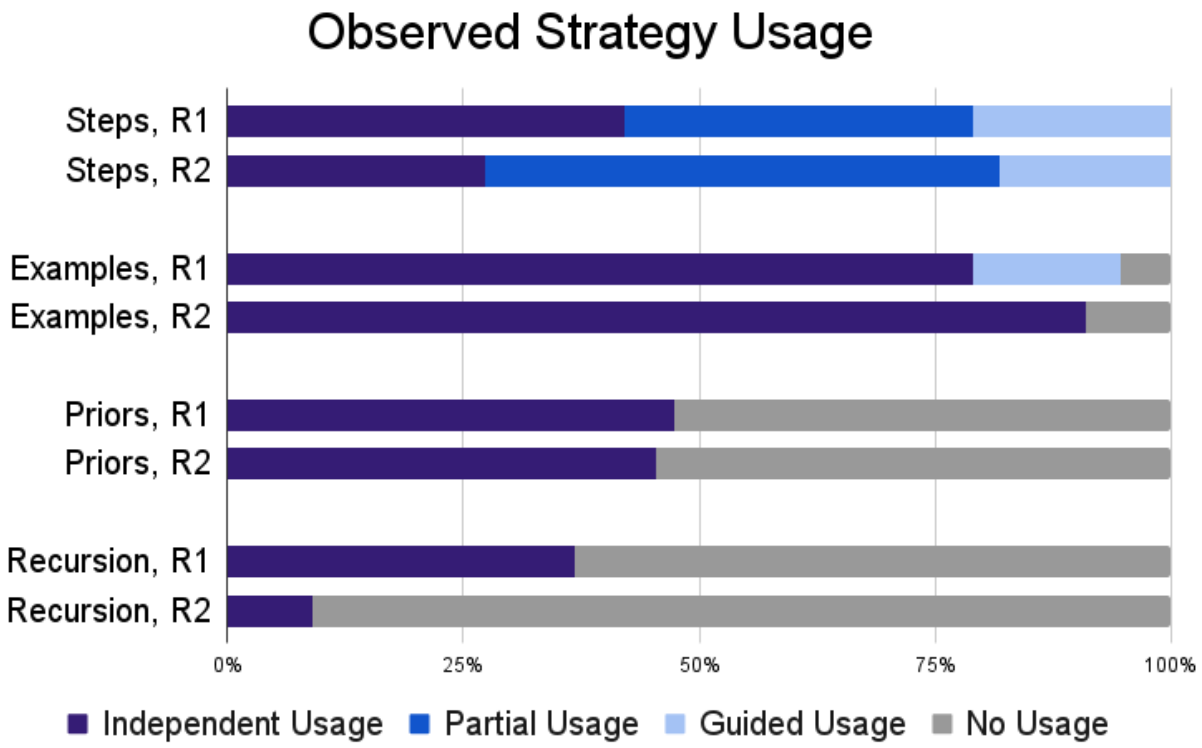


Figure 3.1: Observed student usage of each identified strategy, split by interview round.

Finding: Independent use of `Priors` *and* `Recursion` *occurred much less often than* `Examples` *use.* Only about half of all students explicitly mention a prior problem while working (14/30), and less than a third of students (8/30) tried following a recursive paradigm. Of the seven students in both R1 and R2, four used `Priors` in one interview but not the other, suggesting that use of this strategy may change by time or problem. Furthermore, the use of `Recursion` is the only strategy with a noticeable change between the two interview rounds, being used far less often in R2.

Level of Help Needed

Because some students only solved the first problem given, *Coin Row* is the only problem that every student was presented and solved. In Figure 3.2, we present the maximum level of help students needed to solve it.

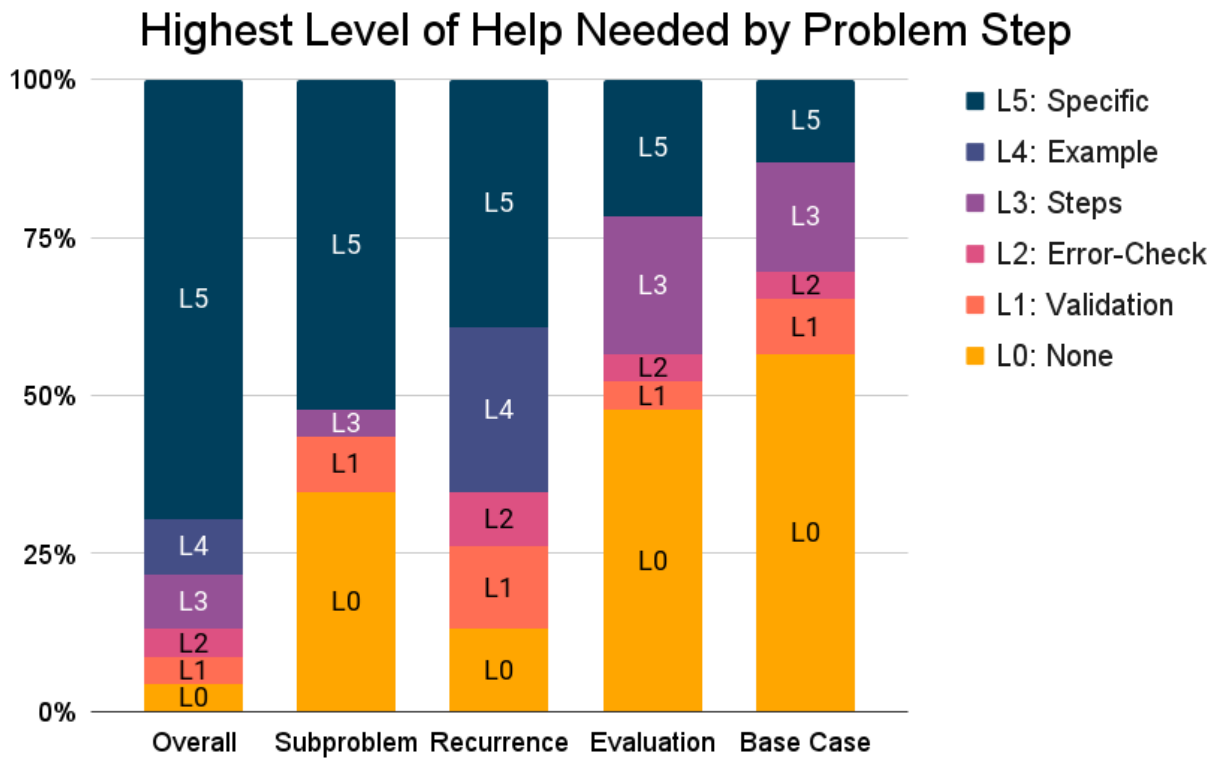


Figure 3.2: Maximum level of help needed for each student to solve Coin Row, split by problem step.

Most students (16/23 or 69.6%) needed L5 help for the problem, though this help was not needed for every step; only three students needed L5 help with more than two steps of the problem, and seven students never needed L5 help. However, we also see that metacognitive help (L3/L4) was generally not sufficient, with only four students being able to solve the whole problem with it.

Finding: Students most often needed help with the `Recurrence` *step, but help with* `Example` *usage (L4) was often sufficient at this step.* With more fine-grained examination, we also see differences in the amount of help required by each step. While `Recurrence` is the step that the fewest students solved without help, it is also the step most conducive to metacognitive guidance, with six students able to solve it with L4 help. It is also the only step where L4 help was enough for some students, suggesting that working with `Examples` may be most useful for illuminating recurrence relations, especially compared to subproblems

which may require more targeted help.

On the other hand, the `Evaluation` and `Base Case` steps tended to be relatively easy, with most students needing at most a pointer (L3) to complete them. This corroborates findings by prior work that `Subproblem` and `Recurrence` are particularly challenging steps. Struggles with the last two steps were not necessarily related to difficulty solving the first two: of the 7 students needing L5 help on the last two steps, only 4 needed L5 help during the first two.

Sources of Learning Metacognitive Strategies

Students were asked to identify sources that most influenced their *approach* to DP problems. Table 2 presents a breakdown of students' self-reported sources of learning. Some students reported multiple sources.

Table 3.2: Self-reported influences.

Source	Count	Source	Count
Practice Problems	10	Lecture Notes	5
YouTube	9	Textbook	3
Lecture	6	LeetCode	3
Office Hours	5	Peers	2

Finding: Students' individual preferences and experiences contribute to their decision of which sources to learn from. The learning resources reported by students varied widely, with some students preferring instructional content while others articulated a need to work through a problem themselves. When stuck on a problem, some students looked beyond class resources for example solutions to emulate: *If I'm confused on a problem, I'll Google the problem and see if there's similar LeetCode ones. (I12)* For some, this reflected a preference to avoid higher-level theory: *For me a lot of Youtube... because Youtube would be focused on how would I go about solving these problems, intuition behind them, not the theory behind it. (I26)*

We also find that individual student experiences can have lasting effects on where (and

from whom) students learn to solve problems. Two students describe influential experiences in their willingness to study with peers, though with opposing conclusions:

> A group of students went outside to talk about what we kind of discussed in office hours, and that was very helpful for thinking of what we discussed... and how I could actually apply it. (I15)

> I think sometimes listening to other people try and solve it who don't have a good grasp on it was very confusing, because that kept guiding me and then I would go to office hours and they'd be like, no, none of that's right. (I30)

3.4.2 Problem-Solving Steps

In this section, we explore the effect of explicitly providing students with the `Steps` strategy (L3 help).

Finding: Providing students with a list of problem-solving steps helped students with orientation and self-regulation while solving problems. We found several ways the `Steps` were helpful. One student stated in their second interview:

> I think the study last time ... was very helpful like trying to be like, oh, I need to first solve a subproblem and then try and see how I can use that to solve the rest of it. (I25)

This orienting feature could be seen in student work as well. Of the 19 students instructed to follow the `Steps`, 12 were working out of order or on a non-DP approach prior to the instruction, and 9 of these 12 subsequently completed the problem in the correct order without deviating. For example, one student (I3) requested help after being unable to define a recurrence without an explicit subproblem in mind, and when presented the `Steps`, completed the problem without needing any further guidance.

The `Steps` were also adopted by students as a regulating question. After being given the list while working on the first problem, one student asked the interviewer to repeat the `Steps` before starting the second problem (I10), and another requested the `Steps` after hitting a roadblock in the second problem: *Can you repeat the list of steps that you break dynamic programming problems into?* (I28)

Finding: Providing students with a list of problem-solving steps was not enough to allow students to solve problems, especially when they had an incomplete understanding of a step. An awareness of the `Steps` was not sufficient for successfully designing an algorithm; of the 19 times students were given the `Steps` to follow, only 3 needed no additional help. Naming the `Steps` was not sufficient if students did not have the theoretical understanding behind them, and the presentation of `Steps`, especially "subproblem" and "recurrence," may actually contribute to the well-documented conflation of divide-and-conquer and DP [Danielsiek et al. (2012); Zehra et al. (2018)]:

> When I think dynamic programming I'm thinking like splitting it into two subproblems and then like maximizing those subproblems. (I12)

Furthermore, one students made clear that the presentation of `Steps` did not substitute for practice:

> Saying, oh, you break it into subproblems and then you find recursive subproblems... this stuff is great, but for me personally, it's not very helpful. And not, you know, that you didn't do a good job. But I'm certainly saying that I learned a lot more from just thinking about these two problems... and being able to ask clarifying questions. (I5)

This student's insight emphasizes the role of an individual's self-regulation in the effectiveness of teaching metacognitive strategies.

3.4.3 Example Inputs and Outputs

In most (25/30) of the interviews, students worked through `Examples` on their own accord at some point. Three students who did not initially use `Examples` were prompted to do so during a guidance request, and two completed the interview without using `Examples`.

Finding: Example inputs help students reason about the problem structure in a concrete way, but was often not enough for students to solve problems. Students turned to this strategy for insight about the problem: *I'm going to look at this if I have numbers, to see if that helps me think through it.* (I10) In some cases, the `Examples` helped students understand how the solution algorithm needed to work:

> So I'm assuming there's some sort of sum of k cases that needs to be maximized, because that's sort of the way I did it when picking values [in my example]. (I9)

However, use of `Examples` alone was not enough: of the 19 students who used `Examples` in solving *Coin Row*, 16 students still needed the highest level of help to complete the problem. One particular issue with the strategy was with students who did not create an illustrative example – 6 students tried developing greedy algorithms for *Coin Row* after creating an example input that lent itself well to a greedy paradigm.

3.4.4 Prior Problems

In approximately half (14/30) of the interviews, students independently utilized the `Priors` strategy, attempting to understand the current problem through the lens of a prior problem. Planned guidance did not include suggesting this due to the inconsistency of problems previously seen by students, so this metacognitive strategy was entirely student-initiated within our data.

Finding: Students who referenced previously-seen problems often struggled with an over-reliance on the details of the prior problem. In one case, a student who had an incorrect

initial conception of dynamic programming subproblems was led to success when referencing a prior problem:

> Okay, so I know dynamic programming, you just find a small sub-problem to solve... maybe one common way is to just keep taking halves, but that doesn't really work. From our previous problem... I think you just take every single index... and then keep increasing until we get to n, so maybe that'll work. (I1)

However, the `Priors` strategy proved detrimental when students relied on a takeaway from a prior problem that was not applicable to the current problem, in terms of what worked previously:

> I think I've solved more 2D problems... [so] I ended up defaulting to 2D... And then I think I realized halfway through that I was kind of defaulting to 2D, and then started not defaulting to 2D, and then for whatever reason, I defaulted back to it even though I realized I didn't need to. And then before I got too far, I was like, oh, wait, I should be in 1D. (I14)

as well as what didn't:

> Interviewer: Was there a reason you were trying to avoid ordering the subproblems?
>
> Student: Well, today in class I suggested [solving] in order, and [the instructor] said it didn't work. (I3)

These students may be extracting the wrong patterns from prior problems, and in doing so hindering their own progress. This was especially exacerbated for students who referenced prior problems with greedy or divide-and-conquer solutions.

3.4.5 Recursive Paradigm

Though DP was taught in the course with the iterative (bottom-up) paradigm, eight students attempted top-down design (`Recursion`).

Finding: Some students express favoring the recursive DP paradigm in helping them develop a recurrence relation. One student stated, soon after beginning to work on the first problem, *I like the more recursive structures when doing these problems.* (I4) The student then proceeds to write the correct subproblem and recurrence in the format of a recursive solution to the problem, before using this insight to write out a full iterative solution, with no guidance needed. Another student instead wrote out the fully recursive algorithm before adding memoization for a full recursive DP algorithm. The student also seemed to be learning why they chose that process:

> I guess now I'm realizing, I think the core of this is understanding the recurrence relation. And probably once you understand the recurrence relation, it's... maybe a bit more trivial because then you just find a way to memoize... so it's like a halfway step. (I5)

This student's moment of self-discovery highlights the varying degree of awareness of one's own metacognitive processes. Despite using the same strategy, they learned it from different sources: I4 stated that this process was developed through LeetCode practice, while I5 cited a short aside during a lecture.

Finding: This reframing is less effective for students who are not as comfortable with recursion or recursive ideas. One student brought up the `Recursion` strategy up as something to avoid: *I did [recursion] at first. I tried to get away from that because it wasn't really working for me* (I10). Furthermore, a number of students tried to create recursive algorithms due to their conflation of DP with divide-and-conquer algorithms. For these students, the `Recursion` strategy is likely to cause increased confusion. We also notice in Figure 3.1 that the use of the recursive paradigm decreased significantly in R2. Though we do not have

interview data about this, it may be explained in part by the decoupling of DP with the recursive divide-and-conquer, or by the lack of formal support for the strategy in class.

3.5 Discussion

Our results provide insight into the individualized nature of metacognitive strategies. Students develop their strategies from a variety of sources, and the value of each source depends on personal experience. Similarly, there is a varied level of intentionality expressed in why students select strategies, as highlighted by the students using the recursive paradigm. We also notice clear self-regulatory awareness in the pushback against both `Steps` ("for me personally, it's not very helpful" (I5)) and `Recursion` ("it wasn't really working for me" (I10)) strategies, suggesting that differences in strategy use were not driven by differences in ability, but rather individual self-regulation leading to differing strategy prioritization.

That said, our results also demonstrate the general effectiveness of metacognitive strategy utilization. Students use these strategies to orient themselves and find direction within a problem, to search for insight when at a roadblock, or to reframe the problem altogether. Though it is difficult to quantify the degree to which the strategies helped, it is clear that the strategies allow the student to make progress, and Figure 3.2 provides evidence that providing generic strategy-related guidance can allow students to solve problems that they otherwise could not, especially in the `Recurrence` step. Furthermore, I25's quote referencing being told the problem-solving steps in the previous session speaks to the potential for longer-lasting positive effects of this kind of guidance, regardless of whether it allows the student to solve the current problem.

Even though we find evidence of both short-term and long-term benefits to the explicit teaching of these metacognitive strategies, reinforcing previous work [Loksa et al. (2016); Donker et al. (2014); Loksa et al. (2022)], our work also reveals pitfalls in the use of each strategy. Within our results, we present ways in which each of the first three strategies

misguided students, and discuss potential for the fourth strategy to do the same. Thus, the presentation of a strategy should not be viewed as an immediate panacea, as the successful adoption of the strategy may be hindered by incomplete foundational knowledge or potential misuse.

Overall, our results demonstrate that the adoption of metacognitive strategies is a highly individualized process dependent on students' personal preferences and their understanding of both the strategy and the related content. Instruction about these strategies must go beyond simply presenting the strategy by recognizing and accounting for the obstacles these factors may present to the strategy's effective use.

3.6 Conclusion

If used successfully, metacognitive strategies provide a structure that allows students to not only solve problems but also reflect on their experience and grow as an algorithm designer. In our study, we find that students rely on metacognitive strategies and exhibit some self-regulatory behavior without prompting, though the source, usage patterns, and success of the strategies can vary between students. We also find evidence that providing guidance towards using these strategies for solving DP problems is beneficial, but not in itself sufficient. As one student describes, "Different people are different. I'm not someone who understands things abstractly, I just have to do it. If you put me in a room and gave me 20 of these problems and gave me someone to ask reasonable questions, I feel like I would learn it [better than] if you gave me the most talented lecturer in the world." (I5) Though the strategies themselves may be sound, developing good teaching techniques for these strategies will require not just good presentation but a careful examination of the individualistic process of learning to adopt them.

CHAPTER 4

RELATING STRATEGY USAGE TO COURSE OUTCOMES

This chapter's work is published with the following citation: Jonathan Liu, Erica Goodwin, and Diana Franklin. 2026. Analogical Reasoning in Undergraduate Algorithms. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1* (USA) *(SIGCSE TS 2026)*. Association for Computing Machinery, New York, NY, USA, 673–679. doi:10.1145/3770762.3772624. Used with permission.

4.1 Introduction

The ability to draw meaningful relationships between a new situation and previously seen problems is a crucial problem-solving skill [Loksa et al. (2016); Polya and Conway (2004)]. This skill, known as *analogical reasoning*, relies on both the proper categorization of prior scenarios into schemas capturing the problem structure and the ability to retrieve the prior scenario and draw appropriate connections and inferences [Gick and Holyoak (1983)]. Prior work has demonstrated that experts perform this categorization well, using structural features to organize problems, while novices often construct schemas that rely more on surface-level features [Chi et al. (1981)].

The development of analogical reasoning abilities may be especially important in the undergraduate algorithms course, which is typically taught in paradigms and emphasizes the structural similarities between problems in the paradigm. However, this skill is traditionally not explicitly taught, and students are expected to learn to develop their ability to construct schema and reason analogically on their own. Research suggests that this is not working, and students often fail to utilize algorithmic strategies that they are familiar with when solving new problems [Armoni (2009); Ginat (2004)]. For this skill in particular, we also found in Chapter 3 that students are selecting relevant previously-seen problems but drawing

inferences that are inapplicable to the new problem.

In this study, we conduct a thorough investigation into the development of analogical reasoning skills and schemas within the algorithms course. We employ a lightweight intervention introducing the explicit modeling and teaching of analogical reasoning and schema development into an undergraduate algorithms course, and ask students solving algorithm design questions on the course's exams to describe the similarity between the exam problem and a previously-seen problem that influenced their design. With their responses, we answer the following research questions:

- **RQ1:** How accurate are student-identified similarities between algorithm design problems?
- **RQ2:** Among the correctly identified similarities between algorithm design problems, how much depth of understanding of the problems do they demonstrate?
- **RQ3:** Does the student's previously-seen problem selection correlate with the depth of their comparison?
- **RQ4:** Does depth of similarity correlate with their success on the new problem?

Broadly, we find that comparison depth correlates closely with performance on the actual algorithm design. Furthermore, comparing student-selected similar problems to expert-selected ones, we find that students whose problems align with experts are signficantly more likely to also create a structural analogy, which is correlated with increased success on the problem. Our results reinforce the importance of analogical reasoning in algorithm design, provide deeper insight into the link between problem selection and analogy creation, and suggest that further work should investigate the development of interventions teaching analogical reasoning in algorithms courses.

4.2 Background and Related Work

4.2.1 Theoretical Grounding

We are interested in understanding the problem-solving skill of drawing meaningful comparisons from previously-seen problems to inform the development of a solution to the new problem. This skill, known as **analogical reasoning**, can be thought of as a three-step process: *Retrieval*, the selection of a prior similar situation; *Mapping*, the construction of an alignment between the two situations that allows for inferences about the new problem; and *Evaluation,* the judging of any inferences made for correctness and applicability [Gentner and Smith (2013)].

Retrieval and Schema Construction

In the construction of a successful analogy, a student must first be able to recall a prior situation with structural similarities to the new one. Unfortunately, problem solvers often fail to remember potentially helpful prior situations, especially when not prompted to do so [Gick and Holyoak (1980); Ross (1984)]. This has been shown to be related to the organization and storage of the prior situation in memory, or the *schema* [Gick and Holyoak (1983)]. In a variety of settings, studies have shown that novices often construct schema that draw similarities between problems that share *surface features*, or aspects of the context of the situation (e.g. both problems are about arrays), while experts and successful retrievers tend to categorize by *structural similarities*, which relate more to the problem's underlying structure and its corresponding solution (e.g. both problems can be solved with dynamic programming) [Chi et al. (1981); Catrambone (2002)]. The field of computer science is no different – novices in computer science are often unable to recognize when to apply learned strategies to novel situations as well [Perkins and Fay (1985); Teague and Lister (2014); Izu and Mirolo (2021)].

Mapping and Evaluation

The core of the analogical reasoning process is the *Mapping*, which involves the alignment of structural components of the two situations and the subsequent transfer of ideas or inferences, and is followed by an *Evaluation* of whether those inferences hold in the new situation [Gentner and Smith (2013)]. Studies have also shown that even when novices are presented with a useful related problem and scaffolding for structural schema construction, they are still often unable to recognize the structural similarities that allow them to apply it to the new situation [Gentner et al. (2003); Gick and Holyoak (1980)].

4.2.2 Related Work

Problem-solving Skills in Algorithms

Prior work studying problem-solving skills within the realm of algorithm design suggests issues with analogical reasoning skills as well. Upper-level undergraduate students have been found to not utilize previously-seen problem-solving strategies when solving algorithm design problems, despite familiarity with problems in the paradigm [Ginat (2004); Armoni (2009)], likely reflecting an issue in either *Retrieval* or *Mapping*. Even when recently taught a new strategy [Izu et al. (2017)] or working within a known paradigm (Chapter 3), algorithms students often fail to correctly reason about connections to problems they've seen before, suggesting issues with *Mapping* and *Evaluation*. While prior work was all conducted in the thinkaloud setting, involving interviews focused on the usage of a specific strategy, our work deepens our understanding of the importance of analogical reasoning for algorithm design by analyzing students' usage of analogical reasoning in the natural context of their course and at a larger scale than prior work. This allows us to investigate broader patterns about how students are using it and the extent to which it is necessary for successful algorithm design.

Interventions for Schema Construction

Interventions have been developed to support students in the schema development process, in the hopes of facilitating stronger analogical reasoning in applicable situations. There is a large body of evidence suggesting that asking students to compare different instances of a problem with an shared structure of interest helps them develop structural schema, even if the problems are both new to the student [Alfieri et al. (2013); Gentner et al. (2003); Gick and Holyoak (1983)]. In computer science education, Muller develops *Pattern-Oriented Instruction*, a pedagogical structure that explicitly names common programming patterns in CS1 (e.g. "Do all items satisfy a condition?") in instruction to assist students with the construction of appropriate structural schema [Muller (2005); Muller and Haberman (2008)]. This structure was also implemented in an algorithms course [Enström and Kann (2017)], though in conjunction with other interventions, so results are unclear.

4.3 Methods

4.3.1 Course Context

This study was conducted on the Spring 2025 offering of an undergraduate algorithms course at a private research-oriented university in the US. The course had around 100 students. Students were divided into 8 recitation sections during which one of four Teaching Assistants (TA) worked through a worksheet of problems with students. The course is split into content units by algorithmic paradigm, and assessment consists of one midterm and one final exam. These exams had medians of 51 and 52.5, respectively.

4.3.2 Intervention

In order to ensure that students were given the opportunity to develop analogical reasoning skills within the course, the first author met with course staff to discuss the integration of

explicit analogical reasoning guidance. The instructor and TAs agreed to model reflecting on a problem and its takeaways during lecture and office hours, and questions prompting students to reflect upon and compare problems were added to recitations and homework problem sets. See Table 4.1 for a description of all changes made to the course. Students were not informed that changes were made. Note that our study does not measure the effectiveness of this intervention; the intervention was introduced to give students practice with explicitly using analogical reasoning before a test situation.

Table 4.1: Additions to Course Pedagogy

Class Aspect	**Description**	**Example Addition**
Lecture	The instructor modeled comparing different algorithms, especially within a paradigm.	"How is this Greedy algorithm similar from the last one we saw in class? How are they different?"
Homework + Recitation	Questions were added that highlighted problem structure to help students with schema construction.	"What was one key insight that allowed you to design this algorithm?"
Office Hours	Course staff was instructed to ask students to reflect on previously-seen questions when appropriate to help solve problems.	"Where have we seen a constraint like this before? How did we solve that problem?"

4.3.3 Recruitment and Data Collection

With IRB approval, students were pointed to a consent form, which stated that researchers were hoping to collect anonymized student work from the course to better understand algorithms pedagogy, and asked whether students wished to opt into having their classwork analyzed. Students received a small amount of credit for submitting the form, regardless of their response, and responses were not revealed to course staff while the course was ongoing. Roughly half of the class (50 students) provided consent.

After course grades were finalized, a course TA was provided a list of consenting students.

These students' graded exams were anonymized and assigned a random identifier so that a student's two exams could be linked to one another, but not back to the student. All consenting students took the midterm, and all but two took the final exam. As the exams were graded, we also obtained students' scores on each problem.

4.3.4 Measurement of Analogical Reasoning

Across the two exams, each student was asked three algorithm design questions, listed in Table 4.2. To measure student analogical reasoning skills, the questions were each appended with the following analogical reasoning reflection question worth two points (out of 90 and 92 total on the exam):

> Identify a problem or algorithm we've seen in class (lecture, homework, or discussion) that influenced your work on this problem. Explain the attributes that are similar in the two problems and how that influenced your work.

We isolate different parts of the analogical reasoning process by analyzing students' answers to these follow-up questions. Answers are analyzed in two parts: *Retrieval* is measured through the question referenced, and *Mapping and Evaluation* are measured through the similar attribute named.

4.3.5 Data Processing

Two students did not take the final, and four reflection questions were left blank, so we collected 142 responses. For each response, the first two authors independently extracted the retrieved problem and the stated similarity, resolving differences through discussion. 18 responses listed a problem but no similarity, and 8 described a subjective or uninterpretable similarity, so they were removed. On the other hand, 26 responses named two problems or two similar attributes, and were split into separate entries. This resulted in 142 total entries, each with a retrieved problem and a stated similarity.

Table 4.2: Exam Questions. The final exam had two versions with slightly modified questions, indicated by the brackets.

Paradigm	Problem Statement
Divide and Conquer	**Concatenation Median:** Design a divide-and-conquer algorithm that takes as input two sorted arrays A and B and outputs the median value of their concatenation.
Graph Algorithms	**One-way [Cost/Quota]:** You are given a map (as a directed, weighted graph) where some streets are one-way. A fire truck may go the wrong way on one-way streets, but [for extra cost k/only up to k times]. Design an algorithm that provides a fire truck's shortest path from the fire station to all other vertices.
Dynamic Programming	**[Longest/Number of] Paths:** You are given a DAG G. Find the [longest path starting from/number of paths from $s \in G$ to] each node in the graph.

4.3.6 Qualitative Coding

The quality of the student's *Mapping* was measured through the stated similarity, with a metric we refer to as *depth*. To measure *depth*, we follow prior work by using `Structure` and `Surface` tags to capture attributes that did or did not indicate understanding of the structure/solution to the problem. Analysis from a pilot study prompted us to add two additional intermediary tags for the algorithm design context: `Paradigm`, for responses that are beyond `Surface` but could apply to any problem in the same algorithmic paradigm, and `Usage`, for responses that named an algorithm directly called by the student's solution (e.g. a student's solution may begin by doing a DFS traversal of a graph, and their response would say that DFS influenced their work on the problem).

In creating this analogy, the student is actually making two claims (that the stated similarity applies to the exam question, and that it applies to the retrieved question), so each response was tagged twice. While `Surface` and `Paradigm` generally stayed the same between both sides of the analogy, students often described a `Structure`-level attribute about their retrieved question that was actually not applicable to the exam question's solution, so

we added an `Incorrect` tag as well. For attributes that were `Incorrect` about the exam problem, we then tagged whether it was true about the student's submitted algorithm.

In Table 4.3, we provide the codebook and examples for our tagging scheme. The first two authors tagged each entry independently, convening repeatedly to discuss and resolve disagreements.

Table 4.3: Similarity Tagging Scheme

Code	**Description**	**Example**
Structure	The response indicates specific knowledge about the problem's structure or solution, beyond `Paradigm` or `Surface` tags.	"This reminded me of the colored edges problem... The solution for that problem also involved duplicating the graph and connecting edges between them to represent the different states."
Paradigm	The stated attribute is generally true about problems in the paradigm.	"Mergesort influenced my thinking for this question, breaking down the problem into smaller subsets is similar."
Usage	The referenced algorithm is called in the student's solution to the problem.	"Since it's DAG, the immediate thought is the algorithm for DAG linearization (topological sorting)."
Surface	The stated similarity can be found directly in the problem statement.	"The roadtrip problem about calculating fuel costs, which also had edge costs & optimizing time."
Incorrect	The stated similarity is not true about the problem.	"I like how you create an inverse graph with Dijkstra's to create this inverse flow/mapper. So Prim's algorithm?"
Inconclusive	The stated similarity is subjective or difficult to interpret.	"I was honestly more influenced by Knapsack. The structure ... was very similar to mine."

To create a metric for problem selection that would allow us to answer **RQ3**, we wanted an understanding of how experts would use analogical reasoning on the problem. To do so, we consulted TA-written solutions to the reflection questions, which listed one or two potential analogy candidates. Any reference question mentioned by a TA solution was tagged as Expert-Retrieved, or `ER`.

4.3.7 Data Analysis

Research questions **RQ3** and **RQ4** are addressed with the help of statistical tests. To answer **RQ3**, whether problem selection is related to depth of similarity, we run a chi-square test of independence, grouping by `ER` and by depth. For post-hoc analysis, we determine the Pearson residual for each cell to identify cells with critical residuals, as recommended by Sharpe Sharpe (2015). To answer **RQ4**, whether similarity depth is related to score on the exam problem, we measure whether the score distributions differ when grouped by depth tag. A Shapiro-Wilk test for normality Riffenburgh (2006) finds that scores on the problems are significantly non-normal ($p < 0.001$). As such, we use the (non-parametric) Kruskal-Wallis test Ostertagova et al. (2014) for differences in multiple groups. We then use the Dunn test for post-hoc analysis, with the Bonferroni correction for multiple comparisons Dinno (2015).

4.4 Results

4.4.1 Overall Reflection Quality

We begin by providing descriptive statistics about the depth and correctness of mappings to answer **RQ1** and **RQ2**. The tags discussed in these results refer to the depth of the mapping made in relation to the *exam problem*. In most (80%) cases, this tag matches that of the reflection made in relation to the *retrieved problem*. `Paradigm` mappings always applied to both problems, and `Surface` level mappings typically did (all but two), but the others had more variation. For example, those that were `Structure`-level for the exam problem could be tagged as `Usage` if it described using the retrieved problem, and those that were `Structure`-level for the retrieved problem could be `Incorrect` about the exam question.

Students retrieved 37 unique problems, with some variation and some overlap: 18 were only referenced by one or two students, 16 were referenced by 3-9 students, and the remaining 3 were referenced by between 10-20 students.

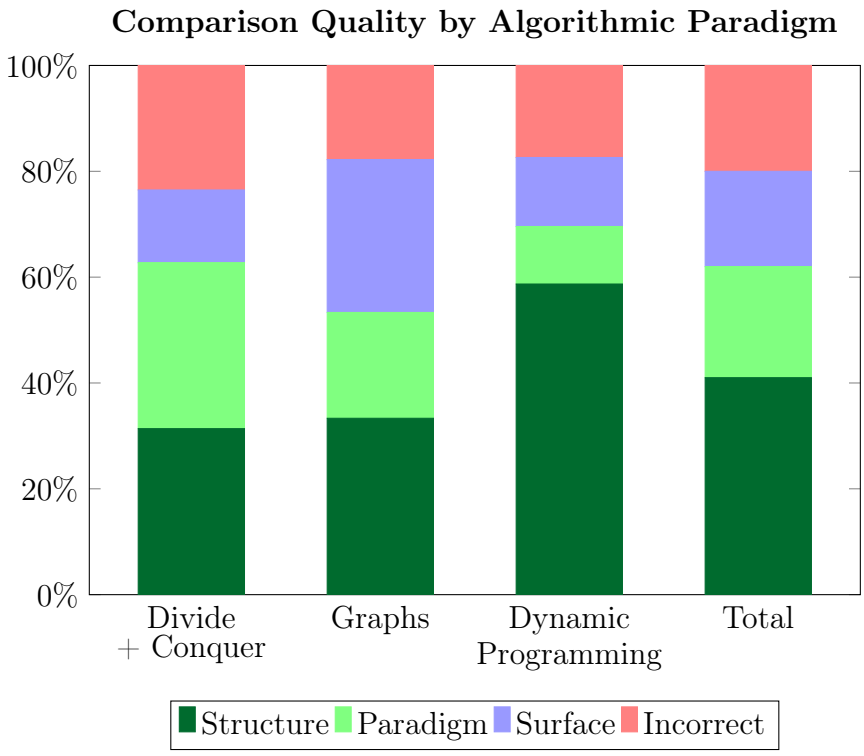


Figure 4.1: Depth of Comparisons by Exam Problem Paradigm. In total, around 40% of comparisons made are structural, and the rest are evenly split between the other three tags.

Figure 4.1 shows the distribution of student responses, sorted by `Structure`, `Paradigm`, `Surface`, and `Incorrect` tags, for each algorithmic paradigm. **In total, we see an even distribution of `Paradigm` (21%), `Surface` (18%), and `Incorrect` (20%) responses, with responses at the `Structure` level appearing twice as frequently (41%).** Across the three algorithmic paradigms, the four tags are still distributed approximately equally, though Dynamic Programming saw a greater number of `Structure`-level responses.

Structural Mappings

The most common tag was a `Structure` level mapping: 53 of our 142 tagged responses constructed a structural analogy between the retrieved problem and the exam problem.

Five reflections displayed a `Structure`-level similarity about the exam problem but not the retrieved problem. Two of these correctly described the `Usage` of another algorithm in

their exam solution, one stated "I don't remember but there was a problem... that also used copying nodes," indicating awareness of `Structure` without explicit retrieval of the relevant problem, and two named attributes relevant to the exam question but `Incorrect` about the retrieved problem. For example, one student reflected on the similarities between the Longest Path exam problem and the Bellman-Ford algorithm, writing that both involved "starting at the last edge and work[ing] back", which is `Incorrect` about Bellman-Ford but correct about the Longest Path exam question.

Incorrect Mappings

On the other hand, 28 students stated an attribute that was `Incorrect` about the exam question. Of these, most (71%) referenced a `Structure`-level insight about the retrieved problem that was `Incorrect` about the exam problem, reflecting a failure in the *Evaluation* of the analogy. That said, in approximately one-third of these reflections (7/20), the `Incorrect` attribute was never used in the student's submission, suggesting a disconnect between the student's answer to the reflection question and their true problem-solving process.

Of the remaining `Incorrect` mappings, four students named an attribute `Incorrect` about both problems, two cited `Usage` of algorithms that the exam solution did not call for, and two cited a `Surface`-level property that was not true about the exam question.

4.4.2 Effect of Problem Selection on Depth

We explore **RQ3** by studying whether a student's reference problem selection relates to the depth of their tag. As described in Section 4.3.6, a referenced problem was tagged Expert-Retrieved or `ER` if it aligned with staff-written solutions. Nine problems were designated `ER` across the five exam questions. Though all nine appeared in the retrievals, they were relatively uncommon: only around a third (48/142) of analogies made by students were to an `ER` problem. The depth of similarity when divided by alignment to `ER` is displayed in

Figure 4.2.

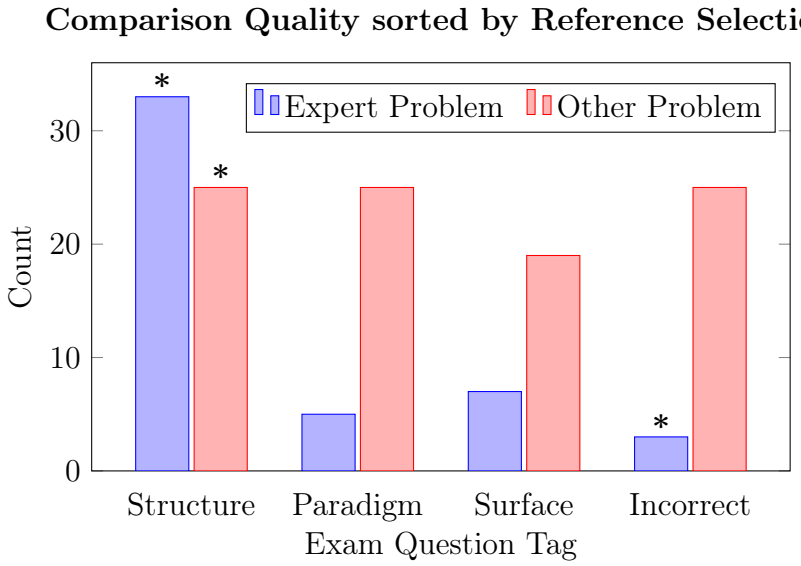


Figure 4.2: Depth of Comparison to Exam Problems, split by whether Referenced Problem was Expert-Retrieved. Asterisks represent measurements with critical residuals ($\alpha = 0.05$). When students aligned with experts on problem selection, the analogy was significantly more likely to be structural.

As shown in Figure 4.2, there is a stark difference in the distributions depending on whether the problem was `ER`. We ran a chi-square test of independence and found a significant difference between the two distributions ($\chi^2(3, 142) = 24.98, p < 0.0001$). Post-hoc analysis reveals that the `Structure` counts for both groups as well as the `Incorrect` count for the `ER` group have critical residuals, indicating that their values are significantly ($\alpha = 0.05$) different from expected. **We conclude that students who selected a problem that aligned with expert recommendation were significantly more likely to name a `Structure` similarity and significantly less likely to name an `Incorrect` one. On the other hand, tags were distributed fairly evenly when the problem was not `ER`.** We thus see that students can still make a `Structure`-level analogy with a non-`ER` problem, but often identified less meaningful similarities instead.

4.4.3 Relationship between Tag and Score

Finally, to answer **RQ4**, we analyze how response depth relates to student performance on the actual algorithm design problem.

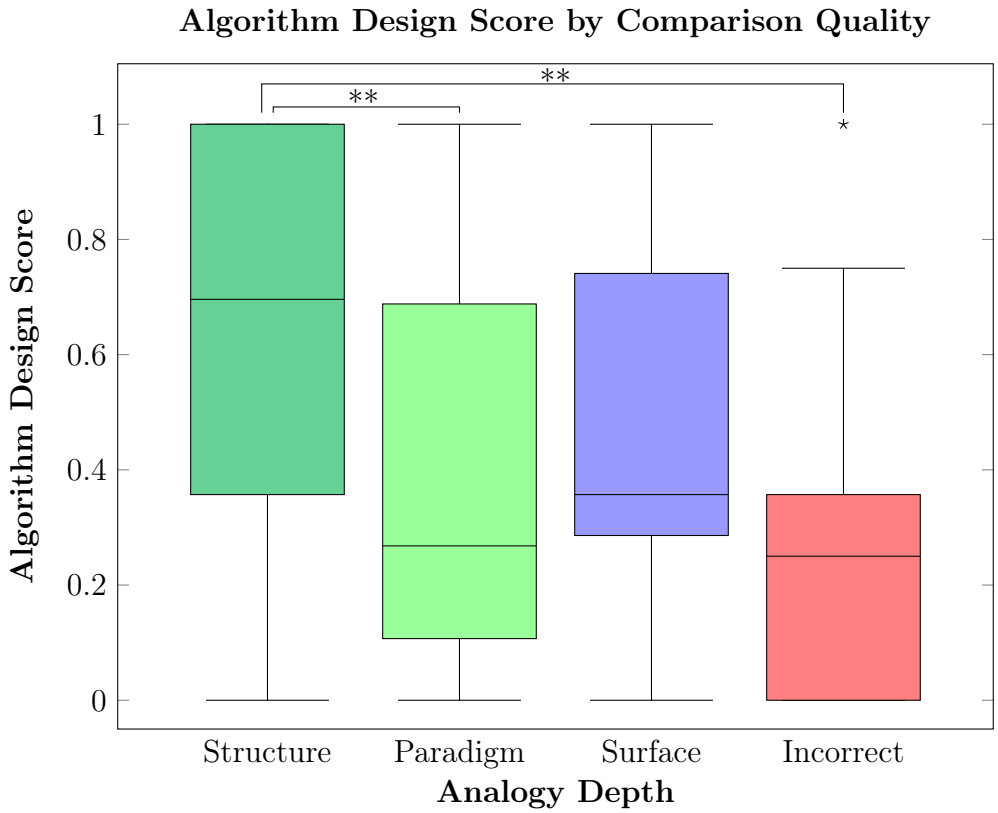


Figure 4.3: Score on Algorithm Design question, split by depth of analogy. Structure analogies scored significantly ($p < 0.001$) higher than both Paradigm and Incorrect level analogies.

We find that student score varied significantly by response depth and that `Structure` similarities are tied to higher scores. Figure 4.3 displays student performance on the problem, grouped by tag. On one end, students with an `Structure`-level similarity perform well on the problem, with over 70% of them scoring at least 50% of the points on the question. For no other tag do even half of the students score more than 50% on the question. On the other end, we see that students with an `Incorrect` similarity score comparatively poorly, with 35% scoring a 0 and 39% more scoring less than half. A Kruskal-Wallis test [Ostertagova et al. (2014)] test for group differences finds that the four groups

are significantly different ($H = 23.3, p < 0.0001$). A post-hoc analysis on the group pairings through Dunn's test with Bonferroni corrections [Dinno (2015)] finds a significant difference between `Structure` and both `Paradigm` ($p < 0.001$) and `Incorrect` ($p < 0.001$).

The lack of statistically significant difference between scores in the `Structure` and `Surface` categories also corroborates the observation that `Surface` similarities performed better than the other two non-`Structure` categories, with over 40% of the `Surface` responses earning above half score on the question (as opposed to around 25% of the responses from both other non-`Structure` categories earning above half score).

Because the number of non-`Structure` tags for `ER` problems is so low, we do not run statistical tests on the interaction between similarity depth and expert selection. However, we note that students who find `Structure`-level similarities to `ER` problems score an average of 66.77% on the question, whereas students who find `Structure`-level similarities to non-`ER` problems score a similar average of 62.65%, implying that similarity depth is a stronger factor than problem selection in terms of algorithm design performance.

4.5 Discussion

Overall, we find that the ability to do analogical reasoning is closely tied to algorithm design ability. Students who create a `Structure`-level analogy between an exam problem and a previously-seen problem also tend to score significantly higher on the exam problem.

4.5.1 The Impact of Problem Selection

Our results strongly suggest that the recognition of a structurally similar problem to the exam problem (as measured by expert selection) is not easy, but that doing so is strongly tied to success in solving the problem. Only around a third of students construct an analogy to an `ER` problem, but students who do so largely (69%) describe `Structure`-level analogies, despite it still being possible to make `Paradigm`-level or `Surface`-level mappings. Furthermore, we

then find that `Structure`-level similarities were significantly tied to higher score on the problem. Even though `Structure`-level similarities scored higher than others regardless of whether the problem was `ER`, the strength of the tie between `ER` and similarity depth indicate that good retrieval problem selection is strongly beneficial for the algorithm design process. On the other hand, the finding also suggests that the ability to find *any* prior problem with a `Structure`-level similarity, and not just expert-identified close problems, is still a strong predictor of success on the actual algorithm design.

As such, it is critical to ensure that students are learning to store and compare problems in schema that are appropriate for future retrieval and structure-level analogical reasoning. Our study implements a lightweight intervention aimed at explicitly modeling the skill in the algorithms course. Though our data does not measure the effect of our intervention, our findings reinforce that interventions with this goal can help students succeed in the course, and should be explored in future research.

4.5.2 Re-evaluation of Schema Sorting

Somewhat contradicting prior literature on novice vs. expert schema, we find that `Structure` and `Surface` tags are not associated with significantly different performance on the problem, despite `Structure` being significantly different from both `Paradigm` and `Incorrect`. This could be a result of inaccurate measurement: the reflection question is a low-value question on a high-stakes exam, so some students may just write the first thing that comes to mind.

However, the results may reveal a disconnect between student problem-solving ability and metacognitive reflection ability. As the reflection question comes after the algorithm design question, students may be properly solving the problem but unable to identify previous problems that influenced their work afterwards, and may be defaulting to a more surface-level comparison just to write something in and move onto the next problem. Splitting the responses by both similarity depth and alignment to `ER`, we find that at any depth level,

students who selected an `ER` problem perform better. These numbers are low — each non-`Structure` depth level is only tagged in around 5 responses with `ER` problems — but they may reflect a student's inability to articulate the deeper similarity between the problems, or that they simply opted to describe a shallower similarity despite knowing a deeper one. It is worth exploring the extent to which student answers to our reflection question properly capture the process with which they solved the problem.

4.6 Limitations

The ability for a student to perform analogical reasoning is dependent on both the exam problem and the questions they have previously been assigned in class, so different exam questions may have resulted in different overall patterns. We also relied on the course's rubric and assigned grades for each problem, so a more standardized grading scheme may have created different trends.

The reflection question was also a short, low-value question, so students had little incentive to spend time constructing the best possible analogy. We expect both contextual and metacognitive factors to potentially limit the accuracy with which student responses reflect their true analogical reasoning ability, and recommend more in-depth study of these processes in the future.

Especially because the reflection *followed* each exam question, we emphasize that our findings are correlational. Students who successfully found `Structure`-level and `ER` analogies performed better on the problem, but our data should not be interpreted as implying causation in any direction.

4.7 Conclusion

In this work, we demonstrate at a classroom-level scale that analogical reasoning is an important skill for algorithm design. Specifically, we find that students who retrieve problems that experts would retrieve also tend to create meaningful analogies with them, and that these meaningful structural analogies are associated with stronger performance on the algorithm design question. Our work introduces a lightweight intervention to give students some practice with the skill, but our findings encourage the development of a more systematically developed intervention that leverages prior work on both schema development and analogy creation to support students in developing the analogical reasoning ability necessary to solve algorithm design problems.

CHAPTER 5

INVESTIGATING STRATEGY INTRODUCTION METHODS

5.1 Introduction

Traditionally, algorithms courses and textbooks do not provide explicit guidance about metacognitive problem-solving skills. In our results from Chapter 3, we find evidence that students are beginning to adopt these skills, but even at the end of the course students often apply them incorrectly, which prevents students from properly utilizing these skills when designing algorithms. In Chapter 4, we demonstrated that the successful development of these skills is closely tied to success in algorithm design. Furthermore, prior work in a wide variety of fields ranging from introductory CS [Loksa et al. (2016); Franklin et al. (2020)] to reading [Vaughn et al. (2011)] and business [Gentner et al. (2003)] find that explicit scaffolding for metacognitive skills is tied to increased adoption of the skill and overall success in the original learning goal. Taken together, these results suggest that the introduction of explicit guidance about metacognitive skills in algorithms courses is a promising avenue to explore in order to improve students' algorithm design skills.

In this work, we investigate the introduction of problem-solving steps to students during the Dynamic Programming (DP) unit of undergraduate algorithms courses. The DP unit was chosen due to its prevalence [Luu et al. (2023)] and perceived difficulty [Enström and Kann (2017)], as well as the paradigm's natural breakdown into a series of design steps. In fact, the steps taught here are similar to the series of steps introduced to students in Chapter 3. We also note that traditional algorithms problems ask students to write pseudocode for an algorithm, which requires not only the *design* but also a subsequent *implementation* of the algorithm. Prior work has demonstrated that some DP-related misconceptions involve this *implementation* specifically [Shindler et al. (2022)]. Practice problems that provide more explicit scaffolding for these steps may give students the guidance needed to properly adopt

them to design and implement DP algorithms.

Though prior work often references these problem-solving steps, there is little existing research establishing the connection between steps-level scaffolding, learning the steps, and successfully solving DP problems. We perform a randomized controlled trial where students are presented with the steps and then solve practice problems with either design-level scaffolding (**Design**) or both design-level and implementation-level scaffolding (**D+I**), compared to a control condition with the steps but no further scaffolding (**Control**), to investigate the following research questions:

- **RQ1:** To what extent does step-level scaffolding impact student **process steps recall** for Dynamic Programming algorithm design problems?
- **RQ2:** To what extent does step-level scaffolding impact the ability for students to **design and implement** in Dynamic Programming algorithm design problems?
- **RQ3:** To what extent does step-level scaffolding impact student **pace** when solving Dynamic Programming algorithm design problems?

5.2 Background

5.2.1 Dynamic Programming Problem-Solving Steps

When novices are faced with complex, multi-step problems, the challenge posed by the problem itself is often coupled with the task of navigation and self-orientation during the problem-solving process. Failure to self-orient may lead students to be unsure about why they're not succeeding and less able to ask for help [Prather et al. (2018); Loksa et al. (2016)]. The introduction of a reusable set of steps can help orient students and promote self-reflection as they learn to solve problems, as has been shown in intro CS and other fields (see Chapter 3.2.1). In Chapter 3, our exploration of student usage of metacognitive strategies included,

for some students, the introduction of a series of problem-solving steps for them to follow when designing DP problems. While we found that providing students with the steps often provided them with some orientation in their problem-solving process or determine the next step to take, it was generally not enough to help them complete the problem.

In this work, we conduct a more targeted investigation into the value of introducing these steps to students as they are learning DP. For this work, we view designing Dynamic Programming algorithms as a process consisting of the following sequence of five **design process steps**:

1. Select a **subproblem** structure.
2. Use a **recurrence** relationship to fill in subproblem values.
3. Select the **iteration order** over the subproblems.
4. Initialize **base case(s)**.
5. **Return** a solution.

While there is no universal agreement on what the DP design process steps are, these steps mirror those used by prior work. Textbooks by Cormen et al. (2022) and Erickson (2019) list steps 1, 2, 3, and 5, folding step 4 into step 3. Work by Enström and Kann (2017) uses the first three steps , and our work from Chapter 3 use the first four. We chose this set of 5 because it represents a union of prior work.

However, though the sources agree that the DP design process occurs in this order, DP algorithms when implemented in code follow a general outline that does not follow this order. This is intrinsic of the process and can not be fixed by adjusting the design steps. For example, the determination of which base cases to compute (Step 4) necessarily depends on the recurrence being used (Step 2), but proper code requires us to compute the base case before using the recurrence, so Step 4 must come before Step 2 in the code.

Figure 5.1 gives an example implementation for a DP algorithm, with the design steps labelled. As shown, the implementation can be constructed through only code from the design steps, but the ordering followed during the design process is not the ordering that the final functional code is written in.

```
def max_sum(C): # Suppose C denotes a list of n coin values.
    # Design Step 1: Select a reasonable subproblem structure.
    # Each subproblem coin_sum[i] is the maximum sum of values of coins
    # picking up from $c_0$ up to $c_i$.
    coin_sum = 1D array with length n

    # Design Step 4: Initialize base case(s) for cost.
    # When you only have one coin, you should pick it up.
    coin_sum[0] = C[0]
    # When you have two adjacent coins, you should pick whichever is larger.
    coin_sum[1] = max(C[0], C[1])

    # Design Step 3: Select the iteration order over the subproblems.
    # We need to use prior subproblems to compute new ones,
    # so we go in increasing order.
    for i in 2...n-1:

        # Design Step 2: Use a recurrence relationship to fill in
        # subproblem values.
        # At coin i, you can either pick it up and combine with the maximum sum from coin 0 up to coin i-2,
        # or you can skip it so it's the same as the maximum sum from coin 0 up to coin n-1
        coin_sum[i] = max(coin_sum[i-1],
                    coin_sum[i-2] + C[i])

    # Step 5: Return a solution.
    # We return the maximum sum from coin 0 up to coin n.
    return coin_sum[n-1]
```

Figure 5.1: A sample pseudocode implementation of a DP algorithm. The design steps (in the pseudocode's comments) are not implemented in the same order, potentially adding to student confusion.

To that end, in this work we view the process of solving a DP problem as a two-phase problem: first a student must **design** the algorithm using the steps listed above, and then the student must **implement** the algorithm by piecing together the code (or pseudocode) related to each step.

5.2.2 Micro Parsons Problems

Parsons Problems are a seminal problem style in Computer Science Education in which students must place a provided scrambled set of code blocks in the correct order to complete a task [Parsons and Haden (2006)]. In restricting the solution space (compared to a free-

response text/code box), these problems allow students to focus on and reason about code structure and syntax, presenting a problem that can help transition students from conceptual understanding to implementation [Ericson et al. (2022)]. An example can be seen in the bottom-right corner of Figure 5.3. This method has been explored thoroughly in introductory [Ericson et al. (2022)] and intermediate courses [Caraco et al. (2025)], and the idea has been extended to proof-writing as well [Poulsen et al. (2022); van den Berg et al. (2024)]. Prior work has shown that practice with Parsons problems is significantly more engaging and efficient while remaining equally effective for learning [Ericson et al. (2018); Haynes and Ericson (2021)], even when adapted to the proof-writing context [Poulsen et al. (2023)]. In this work, we implement Parsons Problems to allow students to reason about how the code from the various design steps can piece together to create a coherent and functional algorithm.

However, each design step must first be converted to code before the code blocks can be rearranged in a final algorithm. To scaffold this process as well, we use Micro Parsons Problems, a recent extension of Parsons Problems in which the blocks come together to form a single line of code [Wu and Smith (2024); Wu et al. (2023)]. This smaller scale allows us to scaffold the implementation of individual lines of code for students, as seen in the bottom-left corner of Figure 5.3. On the other hand, these problems still have a larger sample space than a multiple choice question, providing more flexibility in the level of scaffolding provided to students.

5.3 Methods

5.3.1 Study Population and Recruitment

Participants were recruited for this IRB-approved intervention from four algorithms course offerings at two institutions in the United States (See Table 5.1). In each course instance,

the study occurred in the few days following the final DP lecture, so students were expected to be novices. Students were recruited through course forums and were told that the study would involve solving practice problems that could help with the transition from lecture to homework. Students who completed the intervention were compensated with either extra course credit or $30, pursuant to institutional IRBs. Course sizes varied by institution and term. Per our IRB, participant demographic data was not collected.

Table 5.1: Each row represents a unique course offering.

Institution	Population	Quarter	Participants
Public R1	UG/MS	Winter 2026	8
Private R1	MS	Winter 2026	29
	UG	Winter 2026	21
	UG	Spring 2026	8

5.3.2 Intervention Design

Intervention Overview

The intervention lasted approximately 2 hours, split into two sections: a 75-minute practice problem session and a 50-minute knowledge check. Sessions were implemented on the online assessment platform PrairieLearn [West et al. (2015)], and students in the intervention worked independently through the website. A researcher was present to monitor students and answer clarification questions but did not provide guidance. Students who finished the practice problems early were free to proceed to the knowledge check. All students were asked the same problems, but the practice problems' scaffolding varied by experimental condition. Conditions were assigned at random, stratified by course.

Five problems were asked across the intervention, selected from textbooks (#6.1 [Dasgupta et al. (2006)], #3.1 [Erickson (2019)]), prior work (*Coin Row*: Chapter 3, [Danielsiek et al. (2012); Zehra et al. (2018); Shindler et al. (2022)]), and online ([GeeksforGeeks (2025a,

2026b)]). All problems were checked with instructors to ensure appropriate difficulty and to avoid overlap with coursework.

Practice Problems (by Condition)

The practice problem section consisted of three DP algorithm design problems. Participants were permitted to reference course materials during this section to mimic a normal practice environment, and were provided with the list of **design process steps**. The experimental conditions varied in the scaffolding provided. See Figure 5.2 for an overview. All five problem statements can be found in Appendix A.1.

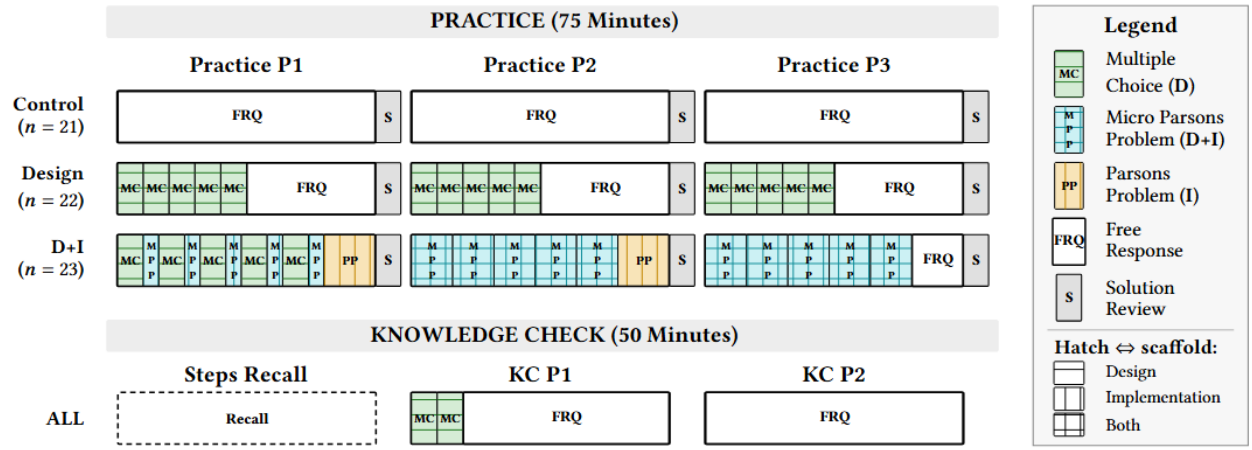
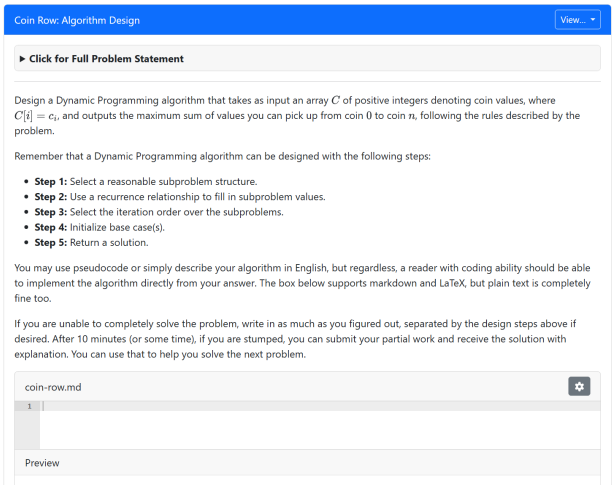


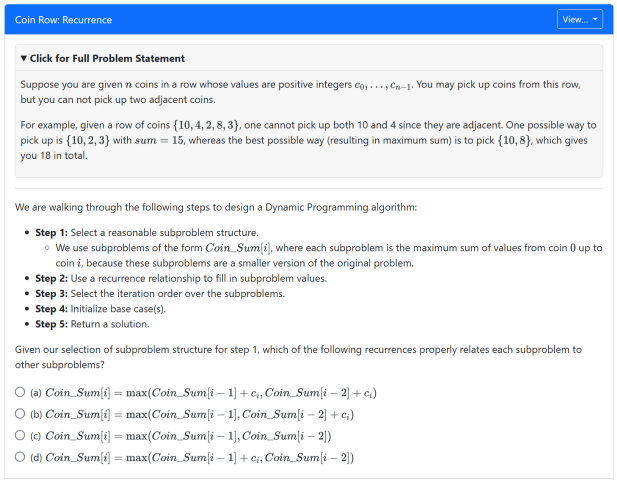
Figure 5.2: A visualization of the intervention. The D+I condition provides both design and implementation scaffolding through a variety of problem types, fading scaffolding between problems as well. See examples of each problem type in Figure 5.3.

In the **Control (C)** condition, participants were given the problem statement and a free-response question (FRQ) asking them to implement the algorithm (Figure 5.3a), mimicking traditional homework. After each problem, participants were shown a detailed solution, including a walkthrough of the proper decision for each design step and pseudocode with annotations referencing the steps.

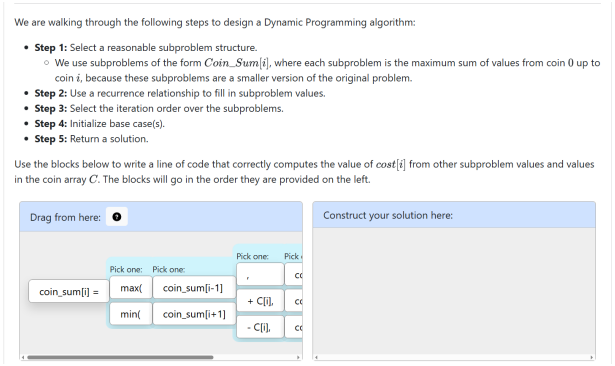
In the **Design (D)** condition, the **design process steps** were scaffolded by multiple choice questions (MC), each one asking for the correct decision for one step (Figure 5.3b).



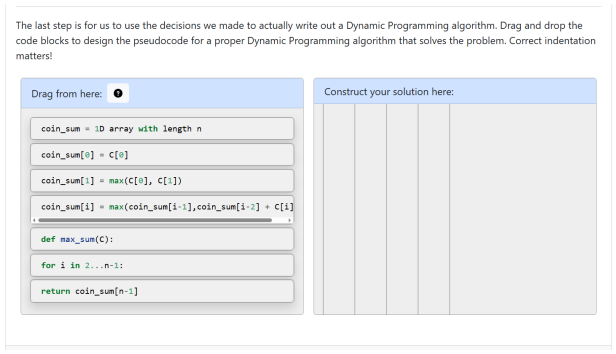
(a) Free-Response Box (FRQ). Unscaffolded.



(b) Multiple Choice (MC) Question. Design Scaffolding.



(c) Micro Parsons Problem (MPP). Design + Implementation Scaffolding.



(d) Parsons Problem (PP). Implementation Scaffolding.

Figure 5.3: Examples of the four problem formats participants could encounter during the session. Alongside what is shown here, participants could also see the problem statement and a list of the design steps in order.

Distractor answers were written following established misconceptions and instructional experience. The platform provided feedback on their selection. After answering an MC for each of the steps, the participant was directed to the FRQ for implementation. These FRQ answers received no feedback, but participants were shown a detailed solution.

In the **Design+Implementation (D+I)** condition, both *design* and *implementation* were scaffolded. Because each design process step corresponds to a single line of code, Micro Parsons Problems (MPP) were used to scaffold the design and implementation of each step (Figure 5.3c), and Parsons Problems (PP) scaffolded the final implementation (Figure 5.3d). These problems were automatically graded, and participants saw the detailed solution afterwards.

Because the *D+I* condition contained multiple problem formats, it followed best practices by fading scaffolding between problems. In P1, participants answered an MC for each design step along with an MPP to implement that design step as code. For P2/3, participants were only asked the MPPs for each step, scaffolding both the design and implementation at once but with a wider solution space and thus less scaffolding. Furthermore, in P1/2 the final implementation was scaffolded as a PP, but in P3 this was faded and participants were given the same FRQ as the other conditions.

Knowledge Check

The Knowledge Check (KC) section was the same across all conditions and consisted of three tasks. First, participants were asked to do a **Steps Recall**: "List, in your own words, the steps that should be taken to design a Dynamic Programming algorithm." Afterwards, participants answered two DP algorithm design questions. The first began with MC scaffolding for the first two design steps before participants implemented the entire algorithm in an FRQ box. In the second, participants were provided no scaffolding. This variation in scaffolding allowed us to get a more holistic measure of student understanding, particularly for students

who would have struggled with an unscaffolded P1.

5.3.3 Data Collection

As students work through the intervention, the facilitation platform collected data on their progress. Specifically, in the **Practice Problems** and **Knowledge Check** sections, each question students are asked in the section, data was collected on what the student submitted and the time it took to make the submission.

5.3.4 Data Analysis

In this subsection, we describe how data was processed and which tests were run. Due to the somewhat small sample size, we only run statistical tests for our core research questions. All FRQ responses were coded by two PhD candidates in CS Education, and disagreements were resolved through discussion.

Steps Recall

For the **Steps Recall** task, responses were evaluated for any added or omitted steps. Steps did not need to be an exact wording match (e.g. "Decide the order of solving the subproblems" was accepted for *Step 3: Iteration Order*). For responses that added or reordered steps, coders used their own judgment for validity. For example, it was valid to list *Step 4: Base Case* before *Step 3: Iteration Order* as these do not block each other, but it was invalid to list *Step 2: Recurrence* before *Step 1: Subproblem* because the recurrence depends directly on the subproblem. Substantial inter-rater reliability was reached (Cohen's $\kappa = 0.87$) [Landis and Koch (1977)].

To perform data analysis, responses were mapped to a binary *Success* variable, where responses that omit one of our steps or list steps in an invalid order were *unsuccessful* and the rest were *successful*. Given the binary variables and somewhat small sample size per

condition, we run Fisher exact tests between each pair of conditions and report Benjamini-Hochberg-adjusted p-values to account for multiple comparisons [Benjamini and Hochberg (1995)].

Free Response Grading

Free-response algorithm design questions were graded out of 15 points, coded deductively using two guiding principles: (1) evaluate the algorithm's design and implementation independently to best measure the different conditions' effects and (2) mimic a reasonable algorithm course's grading rubric. Below is an overview of the rubric: the full codebook, with full descriptions and examples, can be found in Appendix A.2.

Design: Design was graded out of 10 points, split unevenly between the five steps. **Steps 3-5** were each worth two points, with a *Progress* (1 pt) tag given for responses that made clear progress towards the correct answer. Each step had an explicit criterion (e.g., for *Base Case*, *Progress* required at least one correct base case index).

Recurrences (Step 2) are the most difficult design step [Danielsiek et al. (2012); Enström and Kann (2017); Shindler et al. (2022); Zehra et al. (2018); Erickson (2019)], so this step was worth 3 points, with *Minor Progress* (1 point) for any response naming a recurrence that uses prior subproblems to compute a new one, and *Major Progress* (2 points) for responses that would work in some cases but not all.

Subproblem (Step 1) selection is a crucial step of the DP design process, but this selection does not need to explicitly appear in an implementation. As such, only around half of the responses describe their selection as a comment, mirroring the provided practice solutions. A stated and correct subproblem earned 1 point.

Implementation: Implementation was graded out of 5 points, coded deductively for the structure of the pseudocode regardless of design step outcomes. An overview of the codebook is given below.

Table 5.2: Implementation Codebook

Tag	Description	Score
Iterative: ✓	Solution is iterative. The only issues with the code are with the design steps.	5
Top-Down: ✓	Solution performs memoized recursion. This is a valid DP implementation method, but is not taught in the courses of our participants.	5
Minor Issue: ∼	Solution is iterative and contains minor formatting issues, like bad indents or referencing uninitialized arrays.	4
Recursive: ∼	Solution is fully recursive, without memoization. Technically correct, but may be very inefficient.	3
Mixed: ∼	Some parts of the solution follow a standard iterative implementation, and other parts are recursive. Example given in text.	2
Other: ✗	Solution is incorrect, beyond mixing iterative and recursive strategies. Often, it is not DP at all.	0
None: ✗	No structure provided.	0

The *Mixed* code refers to responses that had properties of both iterative and recursive implementations, resulting in a self-contradicting algorithm. For example, one submission included:

```
for i=1 to v:

    return canMake(i-1) or canMake(i-2)
```

The first line gives an iterative structure, but the second follows a recursive one. Given that these students are expected to be proficient coders, this is likely an artifact of an incomplete understanding how to implement a DP algorithm rather than a control flow error.

To conduct statistical tests, we sum students' performance on both KC problems for a total score out of 30. Total scores had "excellent" intraclass correlation between raters (ICC(2, 1)=0.966) [Cicchetti (1994)]. Individually, each rubric item reached "substantial" inter-rater reliability (quadratic weighted Cohen's κ between 0.761-0.954) [Landis and Koch (1977)]. We run pairwise Mann-Whitney U tests and report Benjamini-Hochberg-adjusted p-values along with Cliff's δ for effect size [Meissel and Yao (2024)].

5.4 Results

5.4.1 RQ1: Steps Recall

At the start of the knowledge check, participants were asked to recall the **design process steps**. Figure 5.4 depicts the accuracy of students in each condition; around half of students in both scaffolded conditions were successful in this task (*D*: 68%, *D+I*: 48%) compared to only 29% of the *Control* participants. Fisher exact tests were run, finding a statistically significant difference between the *C* and *D* conditions ($p_{adj} = 0.030$) but not between *C* and D+I ($p_{adj} = 0.277$) or between *D* and *D+I* ($p_{adj} = 0.277$).

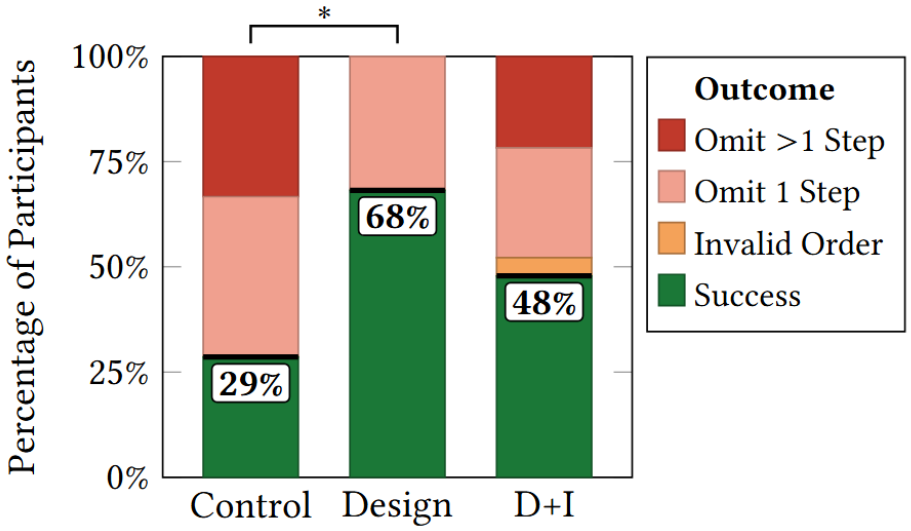


Figure 5.4: Outcome of the Steps Recall question, with success percentage displayed. Success rates of *Control* and *Design* conditions are significantly different, $p_{adj} = 0.030$.

In Figure 5.5, we provide a summary of the steps that participants omitted, grouped by condition. We see that the most commonly omitted step across all conditions was *Iteration Order* (C: 57%, D: 18%, D+I: 39%). The next highest was *Return Value* (C: 33%, D: 9%, D+I: 9%).

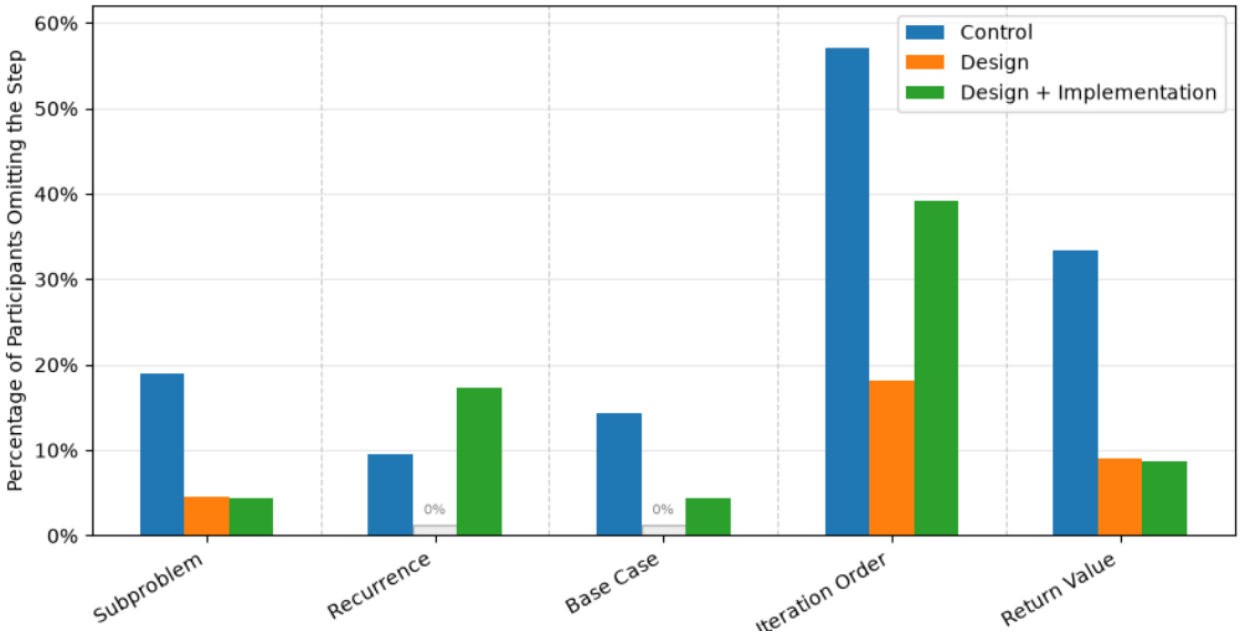


Figure 5.5: Steps Omitted during Recall, split by Condition. Iteration Order is the most-omitted step in all conditions.

5.4.2 RQ2: Knowledge Check Performance

RQ2a: Total Performance

Each of the two DP questions in the Knowledge Check (KC) was graded on the 15-point scale (see Section 5.3.4). For KC P1, the *Subproblem* and *Recurrence* were scaffolded by MC questions, so points were awarded based on correctness on those questions rather than in the FRQ. As shown in Figure 5.6, both scaffolded conditions average around 7 points higher than the C condition, with gaps in both P1 and P2.

Mann-Whitney U tests were run between each pair of conditions, finding significant differences with medium-to-large effect sizes [Meissel and Yao (2024)] between C and D ($U = 114.5, \delta = -0.504, p_{adj} = 0.014$) and C and $D{+}I$ ($U = 136.5, \delta = -0.435, p_{adj} = 0.027$), but not between D and $D{+}I$ ($U = 281.0, \delta = 0.111, p_{adj} = 0.597$).

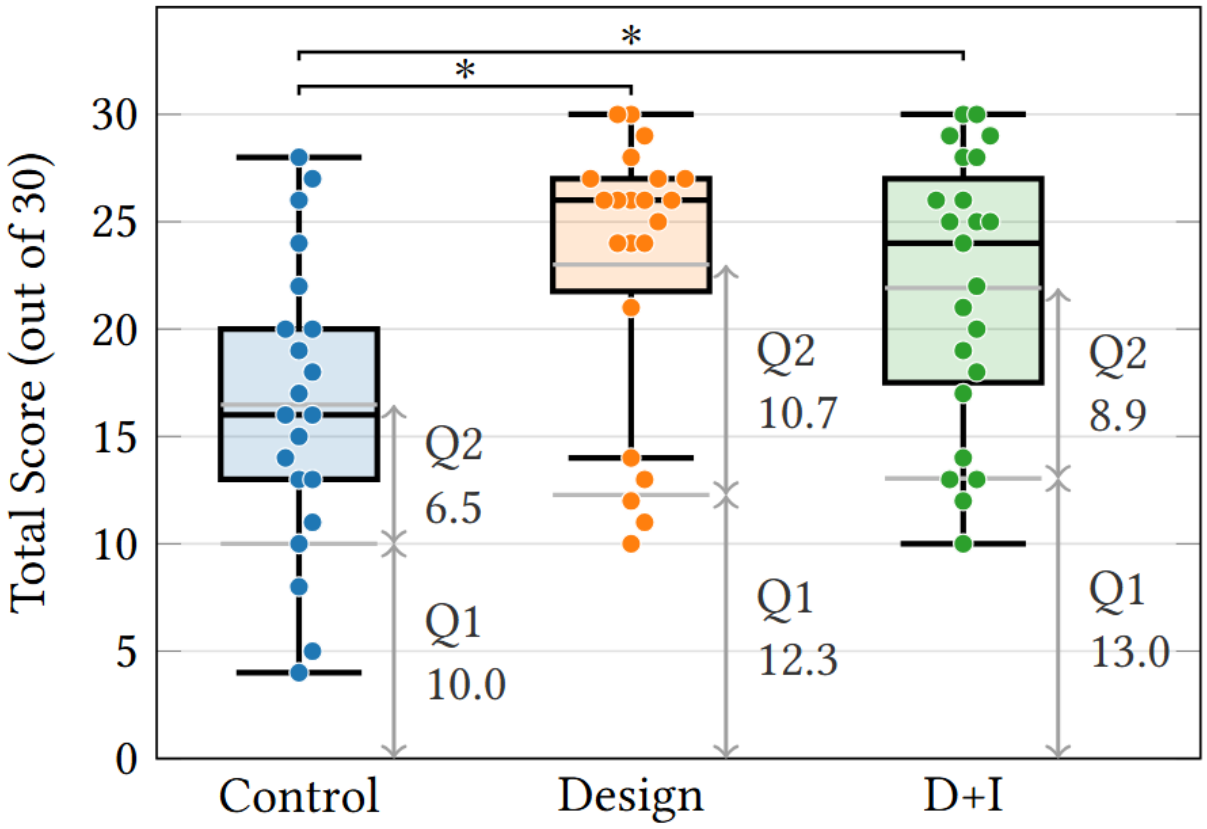


Figure 5.6: Total Knowledge Check Performance by Condition, with per-question averages shown on the right. Each dot represents an individual participant. Control distribution significantly differs from each scaffolded condition, $p<0.05$.

RQ2b: Design-Specific Results

Though every step of the design process is necessary, not all of them are equally difficult. Figure 5.7 shows performance in the Knowledge Check broken down by step. Though the steps have different averages, the by-condition hierarchy is consistent across steps and problems: participants in the *C* condition perform the worst in every step, whereas *D* and *D+I* have similar performance, trading off which condition did better by step.

RQ2c: Implementation-Specific Results

Figure 5.8 shows implementation results split by condition and problem. Outcomes are split into three groups mirroring Correct/Progress/Incorrect, denoted by a symbol and split by

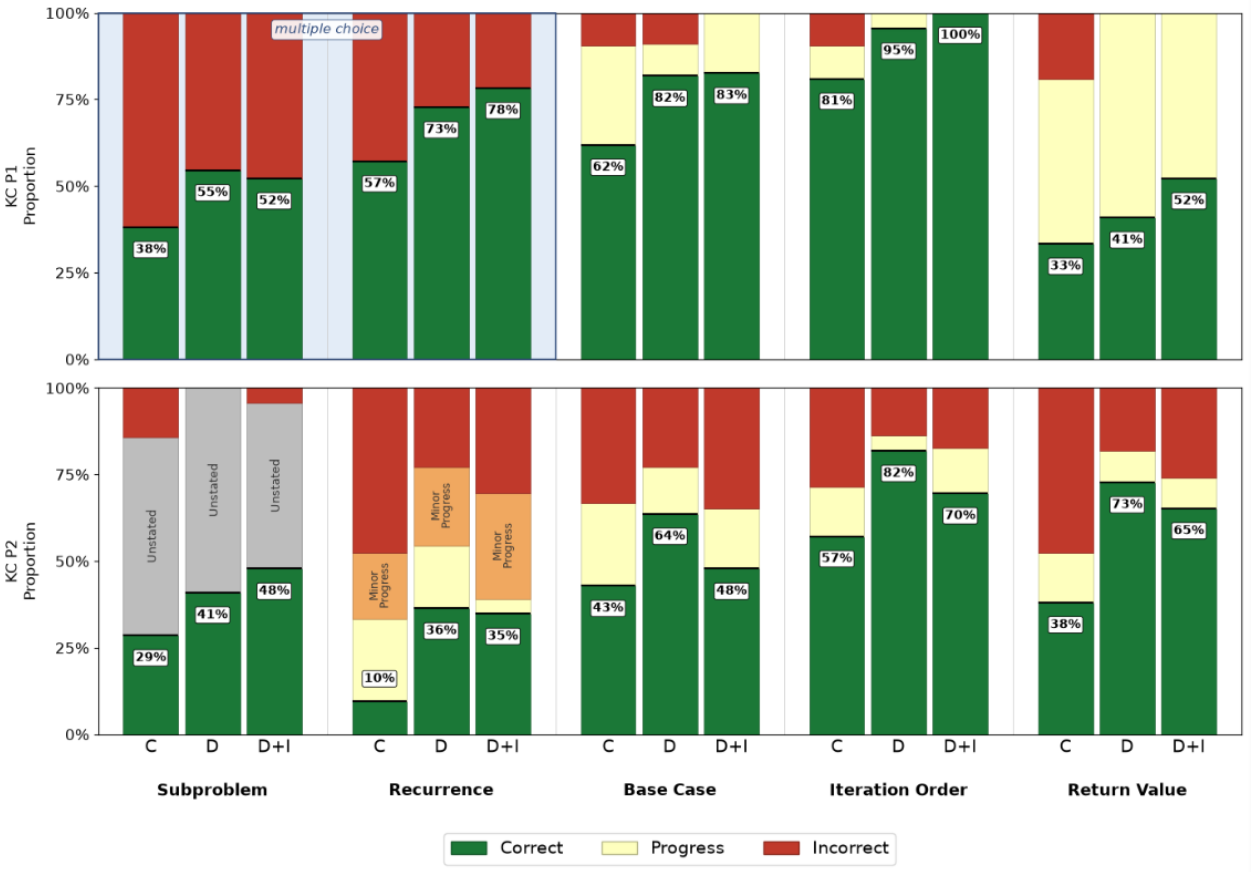


Figure 5.7: Per-Step Design-Level Results from Knowledge Check Problem 2, with Correct Percentage Shown. See Section 5.3.4 for rubric details. In all steps, we see that *Control* performs the worst.

thick lines within each bar.

Participants in the *Control* condition struggled with implementation compared to the scaffolded conditions: correctness percentages are much lower, and they also make a wider range of implementation errors, with more *Recursive:* ∼ and *Other:* ✗ tags in every problem. On the other hand, participants in the *D+I* condition do not perform any better on implementation than those in the *D* condition, despite receiving implementation-level scaffolding.

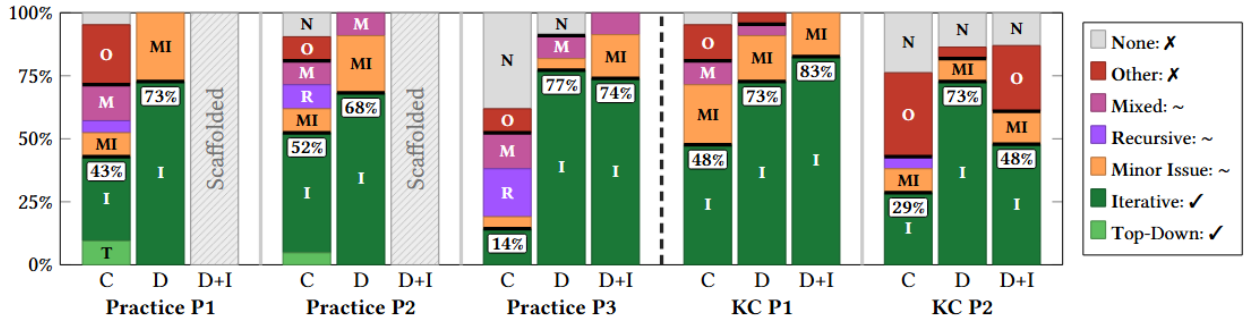


Figure 5.8: Percentage of students achieving each level of implementation performance, split by problem and then by condition. See Table 5.2 for rubric details. *C* performs the worst on every problem and consistently has a wider variety of errors. The other two conditions are comparable despite *D+I* receiving implementation-level scaffolding.

5.4.3 RQ3: Condition vs. Practice Time

Total Practice Time

Prior work asserts that Parsons-like problems reduce practice time while producing equivalent learning [Ericson et al. (2022)]. See Figure 5.9 for the total practice time breakdown, split by condition and task. The total practice times do not differ significantly, by a Kruskal-Wallis test: $\chi^2(df = 2) = 0.84, p = 0.657$. In P3, *Control* participants perform the fastest, but they also perform much worse on implementation (*C*:14% correct, *D*:77%, *D+I*:74%; see Figure 5.8), so their time on task may not reflect time to *complete* the problem.

Time Breakdown by Task

Figure 5.9 also reveals that the implementation scaffolding does not seem to provide a substantial practice time benefit. On one hand, the Parsons problem (PP) is much faster than the FRQ for the implementation task, especially in P1 where *Design* participants took an average 7.75 minutes to complete the implementation compared to 1.30 minutes for the PP. However, because participants are also asked to first implement the design steps through MPPs, the difference vanishes to less than a minute in P1 and P2.

Another clear effect of the conditions on practice time can be seen in the amount of

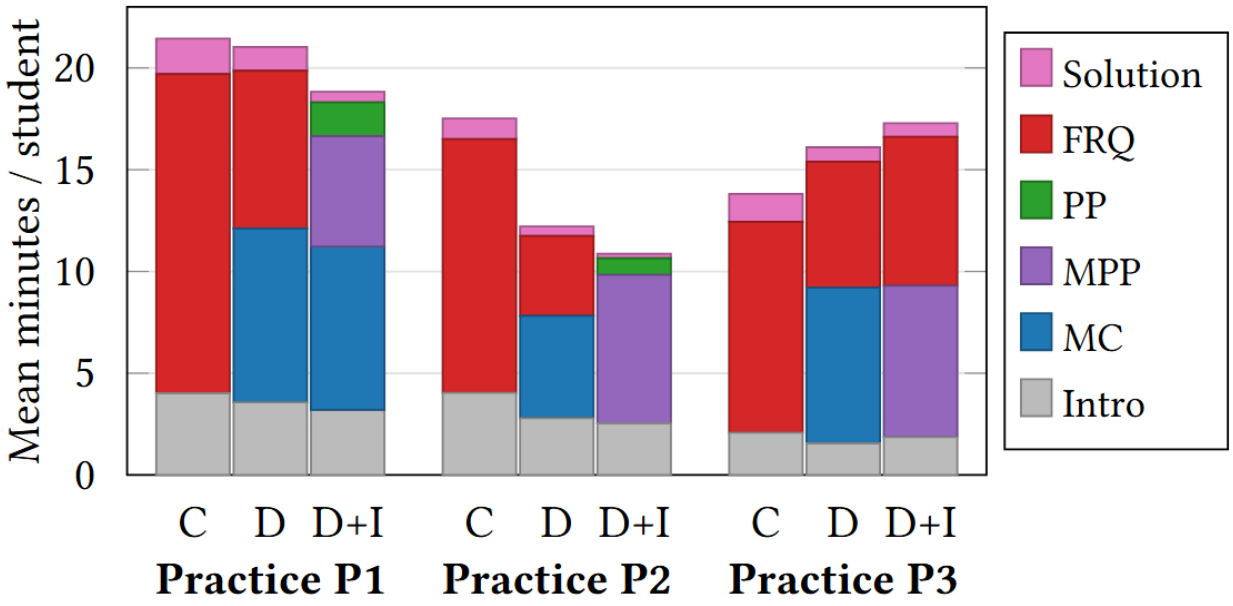


Figure 5.9: Average Time Spent on Different Parts of Practice Problems. We find no significant differences in time spent.

time students spend reading the provided solutions. As shown in Figure 5.10 below, the distribution of practice time spent reading the solution varies between conditions.

Across the practice session, *Control* participants average four minutes of practice time reading the solutions, as opposed to 2.31 minutes for *Design* and 1.42 minutes for *D+I*. These outcomes align with expectations, as participants who received the most scaffolding will have the least new information in the solutions. That said, it is visually clear that the difference is only distributional. For example, in Practice P3, all three medians are within 10 seconds of each other, but the IQR for *Control* is 114.7 seconds while the IQR for *Design* 28.9 seconds. We conclude that many students are breezing past the solution without much engagement, but students in the *Control* condition were more likely to take more time on them.

In Appendix A.3, we conduct exploratory analysis on the extent to which the measurements and findings in this section relate to performance.

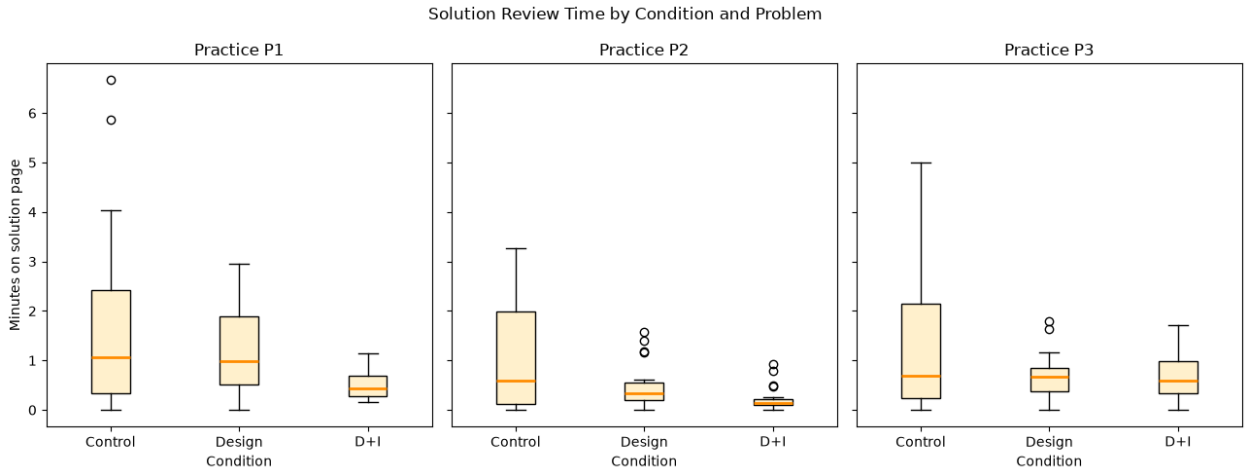


Figure 5.10: Average Time Spent Reading the Provided Solutions to Practice Problems. Distributions vary widely by condition.

5.5 Discussion

5.5.1 The Value of Design Scaffolding

Our results provide a clear demonstration that design process scaffolding is beneficial for student learning. Participants in scaffolded conditions recall the design steps significantly better (RQ1), which can provide long-term metacognitive benefits as they continue to learn and practice DP problems [Loksa et al. (2016); Prather et al. (2019)]). Participants in scaffolded conditions also applied the steps more successfully (RQ2) in every design, implementation, and total metric. In addition, when *Control* participants had the subproblem and recurrence scaffolded (KC P1), 73% of them gave an implementation that was either iterative or had a minor issue, much higher than on all other problems. Providing steps-level scaffolding with feedback significantly improves student engagement with the steps and student learning.

5.5.2 The Value of Implementation Scaffolding

Implementation-level scaffolding was introduced as a condition in response to implementation barriers reported by prior work. However, our data does not show a benefit from

implementation-level scaffolding beyond that of *design*-level scaffolding. In fact, 73% of *D* condition participants were already successful implementing in Practice P1, and by the knowledge check there was no consistent difference in implementation between the two scaffolded conditions. In Practice P3, where both conditions have all design steps scaffolded, the successful implementation rate was nearly identical.

One major benefit of Parsons problems from prior work is that they take less time for students to complete, but not in our study. The Parsons problem itself takes less time than the FRQ for implementation, but the added time for the design steps and line-by-line implementation (MPPs) produces roughly equivalent total times.

However, the gap in implementation success between *C* participants and the scaffolded conditions demonstrates that the task is not inherently easy, as corroborated by prior misconception-finding work. Instead, our results suggest that it may be tractable enough for students who successfully complete the design steps that implementation-level scaffolding provides little marginal benefit.

Across all RQs, we also find many results where participants in the *D+I* condition perform worse than *Design*: participants in the *D+I* condition have a worse steps recall success rate (RQ1), lower knowledge check performance (RQ2), and higher total practice time (RQ3). Though none of these results are statistically significant, their consistency may hint at the potential for over-scaffolding. If implementation is an approachable task once the design process is scaffolded, then providing the Parsons Problem instead of an FRQ may only result in participants decreasing the depth with which they engage with the material (and the steps).

5.6 Limitations

FRQs can only indirectly measure step-wise ability for a couple of reasons: it creates potential dependencies between steps (e.g., a student who is unable to select a subproblem will likely

not write any return values) and does not require an explicit subproblem definition (as described in Section 5.3.4). Measuring steps independently would require us to prompt students to solve each step separately, which would have created more scaffolding. As such, the presented step-based results are not a direct reflection of step-wise ability, though the comparisons between conditions are still valid.

Similarly, the ability to recall the process steps is a short-term and indirect measure of metacognitive strategy adoption. Future work could use longer time scales to measure more lasting effects of this scaffolding on student metacognition.

The scaffolded conditions provided more immediate feedback during practice, which confounds the mechanism by which participants improved. On the other hand, the control condition should not be taken to be business-as-usual, but instead best practices with limited support for the process steps. In particular, traditional practice (e.g. homework) may not give students solutions to reference after each problem, nor would it list process steps at all, so performance in the control condition may be higher than otherwise. Both of these limitations may change some of the results' mechanisms but do no change the implications of the findings.

Finally, problem selection, rubric design, sampling bias, and coder bias may all impact results, though we hope that the randomized and controlled structure mitigates some of these effects.

5.7 Conclusion

While introducing generic process steps has proven to help novices navigate complex problem-solving, practical considerations around the extent to which they need to be scaffolded for students has not previously been explored. In this work, we bolster prior findings that the mere introduction of these steps may not suffice for student adoption, but that scaffolded guidance along these steps leads to significant increases in student ability to both recall the

steps and use them, reflecting short-term and potentially longer-term metacognitive benefits to this scaffolding. Our results also suggest that students who can successfully work through the design steps are able to succeed at the implementation step on their own, implying that implementation-level scaffolding seems to provide minimal marginal gains. As a whole, our work reinforces the idea that metacognitive instruction is valuable but must be coupled with sufficient scaffolding to allow for proper student adoption.

CHAPTER 6

CONCLUSION

Algorithm design is one of the most broadly applicable skills in computer science: it allows us to utilize the inherent structure of data to develop efficient solutions to all kinds of real-world computational problems. However, in order to realize this value, students must learn to reason beyond lecture content and develop more broadly applicable problem-solving strategies. The explicit instruction of metacognitive strategies is a potent and well-established pedagogical approach that instructors and researchers should leverage to promote the development of these strategies.

In this work, we construct a foundation upon which we hope the development of these pedagogical strategies will continue. We talked to algorithms students, finding not only that students are only partially developing these strategies independently in classes but also that these partial understandings can hinder their progress in solving problems. We also found that the existing literature, while sparse, corroborates the claim that students' are learning course content but not necessarily transferrable problem-solving skills. We added a short metacognitive task to algorithms exams, finding strong evidence that metacognitive reasoning is closely tied with algorithm design ability, reinforcing the potency of this approach. Finally, we took a first step towards intervention design, finding that proper scaffolding alongside explicit metacognitive instruction is crucial for adoption and for learning, though there is a potential risk of overscaffolding.

At a broader level, we hope this work serves as a starting point for the evolution of algorithms pedagogy. We hope that algorithms classes and instructors will be equipped with pedagogical practices that more effectively foster students' metacognitive strategies, and that this leads not only to more equitable classrooms but also to students better-equipped to apply their learnings to the real-world problems they will face.

APPENDIX A

CHAPTER 5 SUPPLEMENTARY MATERIAL

A.1 Problem Statements

Table A.1: Chapter 5 Practice Questions. All students were presented the same three questions, but they were scaffolded differently. Students had around 75 minutes to complete these questions.

Problem Title and Source	**Problem Statement**
Practice P1: **Frog Jump** [GeeksforGeeks (2025a)]	A frog is trying to use a sequence of n stones to jump across a river. The frog is starting at stone 0 and wants to end at stone $n-1$. The stones have heights $h_0, h_1, \ldots, h_{n-1}$. At each stone i, the frog may only jump to either stone $i+1$ or stone $i+2$. If the frog jumps from stone i to stone j, the cost is $(h_i - h_j)^2$. Design a Dynamic Programming algorithm that takes as input an array H of positive integers, where $H[i] = h_i$, and outputs the cost of the minimum cost sequence that gets from stone 0 to stone n, as described by the problem.
Practice P2: **Coin Row** [Zehra et al. (2018), Chapter 3]	Suppose you are given n coins in a row whose values are positive integers $c_0, ..., c_{n-1}$. You may pick up coins from this row, but you can not pick up two adjacent coins. Design a Dynamic Programming algorithm that takes as input an array C of positive integers denoting coin values, where $C[i] = c_i$, and outputs the maximum sum of values, following the rules described by the problem.
Practice P3: **Make Change** [Dasgupta et al. (2006) #6.1]	Given a list of denomination of coins with unlimited supplies $C = \{c_1, c_2, ...c_n\}$, where each c_i denotes one denomination, we wish to make change for a value v; that is, we wish to find a combination of C whose values add up to exactly v. Design a Dynamic Programming algorithm that takes as input an array C of positive integers and a target sum v, and outputs whether (True or False) you can make change of v using only coins from C (repetitions allowed).

Table A.2: Chapter 5 Knowledge Check Questions. All students were presented the exact same questions. Students had around 50 minutes to complete these questions.

Problem Title and Source	**Problem Statement**
KC P1: **Contiguous Sum** [Erickson (2019) #3.1]	A contiguous subsequence of a list $S = s_1, s_2, ..., s_n$ is a subsequence made up of consecutive elements of S. For instance, if S is $$5, 15, -30, 10, -5, 40, 10,$$ then $15, -30, 10$ is a contiguous subsequence but $5, 15, 40$ is not. Subsequences can be of length 1, but not of length 0. Design a Dynamic Programming algorithm that takes as input an array $S = s_1, s_2, ..., s_n$ of integers and outputs the largest contiguous sum.
KC P2: **Travel Plan** [GeeksforGeeks (2026b)]	You are planning a road trip to visit all of your friends, but you don't have a car! Thankfully, your friends are willing to drive you around. You have n friends, labeled 0 to $n-1$ in the order that you plan to visit them in. For each friend i, denote by d_i the maximum number of stops they are willing to drive you, where d_i is a positive integer. For example, if $d_2 = 4$, that means you can ask friend 2 to drive you for any number of stops up to friend 6. Design a Dynamic Programming algorithm that takes as input an array $D = d_0, d_1, ..., d_{n-1}$ of positive integers and outputs the least number of drivers needed, as described by the problem above.

A.2 Free-Response Evaluation Codebook

In this section, we present our full free-response evaluation codebooks. These codebooks were developed deductively, with all authors providing input. As described in Sections 5.3.4 and 5.3.4, each rubric item reached "substantial" inter-rater reliability, and overall scores also had "excellent" intraclass correlation.

A.3 Exploratory Results: Practice Time vs. Performance

In this section, we present further investigation into the relationship between practice time and performance, aiming to understand and explain the differences (and lack thereof) between the practice time of participants in different conditions found in Section 5.4.3. Our

Table A.3: Design Step Codebooks, Steps 1 and 2

Code	Description	Example	Pts.
		Subproblem	
Explicitly stated, correct	Explicitly states a subproblem definition that is correct. This will typically be in a comment.	Initialize Cost is a list where each Cost[i] is the minimum cost to get to the ith stone.	1
Unstated	No subproblem definition is given.	for i from 2 to n-1: sum[i] = max(c_i + sum[i-2], sum[i-1])	0
Explicitly stated, incorrect	Explicitly gives a subproblem definition that does not work for this problem.	The subproblem strcuture that made the most sense to me was the knap-sack style subproblem structure, which compares items and weights to try and produce a minimal sum with the most number of objects. [Author's Note: This was P1: **Frog Jump.**]	0
		Recurrence	
Correct	Gives a recurrence that is correct. Off-by-one issues involving indexing are acceptable.	Practice P2: cost[i] = max(cost[i-1], cost[i-2] + C[n-i]	3
Major Progress	Writes a recurrence that would work in some cases, or makes minor logical issue.	Practice P3: CanMake[i] ^= CanMake[i - c_j] [Author's Note: Used ^= (XOR) instead of |= (OR).]	2
Minor Progress	Writes something that uses prior subproblems to compute a new one	KC P2: for i = 1 ... n-1 ask = MinAsk[i-1] + 1, not_ask = MinAsk[i-1] MinAsk[i] = min(ask, not_ask)	1
Incorrect	Does not write a recurrence of any kind. Could be computing subproblems without using prior ones.	KC P2: for i=1 to len(D): M[i] = max()	0

hypothesis was that students in the *Control* condition would take the longest to complete the practice sessions.

In our analysis, we find that *Control* participants are performing worse in the knowledge check, but seem to spend the same amount of time in practice as the other conditions. One hypothesized explanation for the lack of practice time difference is that participants in the

Table A.4: Design Step Codebooks, Steps 3-5. Note that the three codebooks are identical except for the *Progress* condition.

Code	Description	Pts.
Correct	Implementation uses the correct [Step].	2
Progress	**Step 3 – Iteration Order:** Says to iterate (either in English or introduces a loop), but does not give an order or gives an incorrect one.	1
	Step 4 – Base Case: Names at least one correct base case to be computed, but does not correctly compute every necessary base case.	
	Step 5 – Return Value: Returns something in all cases. It is not enough to simply return something for base cases.	
Missing	Does not describe [step], or provides something that does not meet the threshold for Progress.	0

Control conditions who perform poorly are spending less time on each problem, and simply giving up. In Figure A.1, we plot participants' practice time against their knowledge check performance.

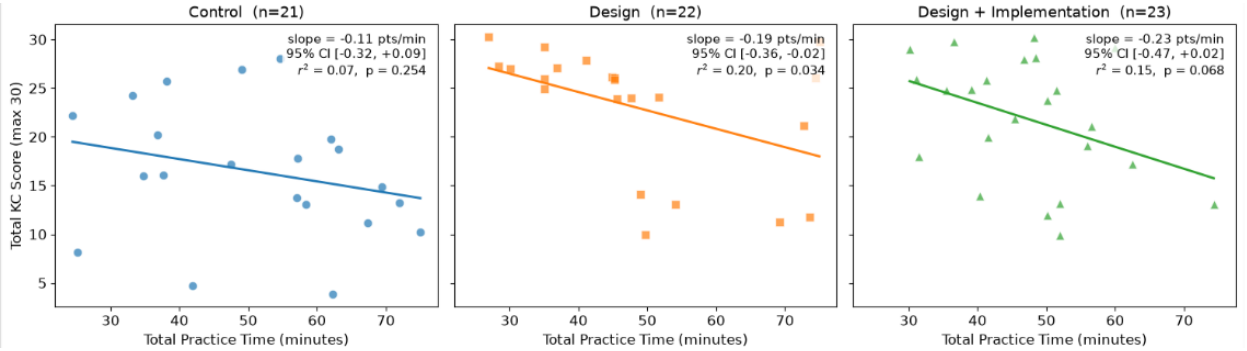


Figure A.1: Ordinary Least Squares Regressions between Practice Time and Knowledge Check Performance. The scaffolded conditions have a somewhat negative correlation, but the *Control* does not.

Contrary to the proposed explanation, in the *Control* condition we find no correlation between practice time and performance. Instead, we find the opposite correlation among the scaffolded conditions: we find negative correlations between practice time and knowledge check performance. This may imply that the scaffolding may function instead as a filter for students: students who understand the material will move through the scaffolded version

more quickly than the unscaffolded version, whereas students with less understanding of the material may spend more time working through the scaffolded problems. In our mind, this negative correlation is a good outcome: students with stronger understanding should be able to spend less time practicing, whereas students with weaker understanding are doing longer, more deliberate practice.

In Section 5.4.3, we found that *Control* participants had a much wider distribution in the amount of time they spent reading the provided solutions. Our natural question was whether solution-reading time correlated with performance on the problem – was the longer solution-reading time a result of students self-identifying that they were struggling with the material? In Figure A.2, we display the correlation between *Control* students' total score on practice problems and their corresponding solution review time for the problem.

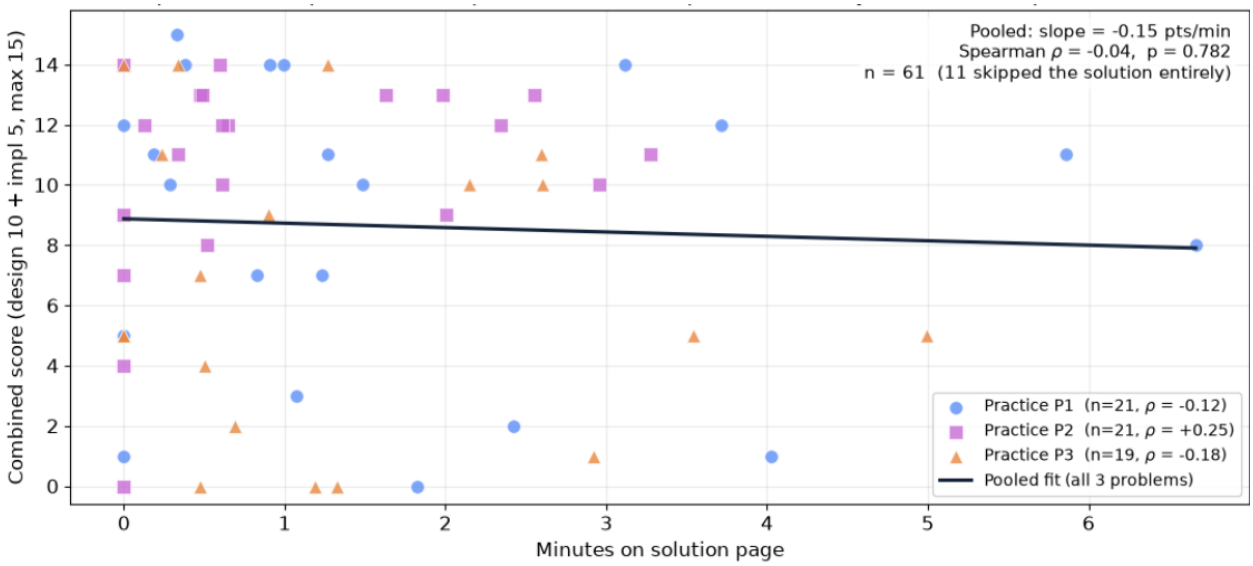


Figure A.2: Ordinary Least Squares Regressions between Practice Problem Performance and Solution Reading Time for *Control* Participants. No significant correlation is found.

Again, we find no correlation – though students in the *Control* condition are viewing the solutions for longer than other conditions, practice performance does not predict their solution view time at all.

REFERENCES

Louis Alfieri, Timothy J. Nokes-Malach, and Christian D. Schunn. 2013. Learning Through Case Comparisons: A Meta-Analytic Review. *Educational Psychologist* 48, 2 (2013), 87–113. doi:10.1080/00461520.2013.775712

Richard Anderson, Ruth Anderson, K. M. Davis, Natalie Linnell, Craig Prince, and Valentin Razmov. 2007. Supporting active learning and example based instruction with classroom technology. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (Covington, Kentucky, USA) *(SIGCSE '07)*. Association for Computing Machinery, New York, NY, USA, 69–73. doi:10.1145/1227310.1227338

Michal Armoni. 2009. Reduction in CS: A (Mostly) Quantitative Analysis of Reductive Solutions to Algorithmic Problems. *J. Educ. Resour. Comput.* 8, 4, Article 11 (jan 2009), 30 pages.

Paulo Eduardo Battistella, Christiane Gresse von Wangenheim, and Jean Everson Martina. 2017. Design and Large-scale Evaluation of Educational Games for Teaching Sorting Algorithms. *Informatics Educ.* 16 (2017), 141–164. https://api.semanticscholar.org/CorpusID:54604571

Brett A. Becker and Keith Quille. 2019. 50 Years of CS1 at SIGCSE: A Review of the Evolution of Introductory Programming Education Research. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 338–344. doi:10.1145/3287324.3287432

Mahnaz Behroozi, Chris Parnin, and Titus Barik. 2019. Hiring is broken: What do developers say about technical interviews?. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 1–9.

Patrice Belleville, Steven A. Wolfman, Susanne Bradley, and Cinda Heeren. 2020. Inverted Two-Stage Exams for Prospective Learning: Using an Initial Group Stage to Incentivize Anticipation of Transfer. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. ACM, New York, NY, USA, 720–738.

Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* 57, 1 (1995), 289–300. http://www.jstor.org/stable/2346101

Jens Bennedsen and Michael E. Caspersen. 2008. Optimists have more fun, but do they learn better? On the influence of emotional and social factors on learning introductory computer science. *Computer Science Education* 18, 1 (2008), 1–16. arXiv:https://doi.org/10.1080/08993400701791133 doi:10.1080/08993400701791133

Jens Bennedssen and Michael E. Caspersen. 2008. Abstraction Ability as an Indicator of Success for Learning Computing Science?. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. ACM, New York, NY, USA, 15–26.

Steven Bird and James R. Curran. 2006. Building a Search Engine to Drive Problem-Based Learning. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITICSE '06)*. ACM, New York, NY, USA, 153–157.

Florent Bouchez-Tichadou. 2018. Problem solving to teach advanced algorithms in heterogeneous groups. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018)*. Association for Computing Machinery, New York, NY, USA, 200–205. doi:10.1145/3197091.3197147

Noelle Brown, Koriann South, and Eliane S. Wiese. 2022. The Shortest Path to Ethics in AI: An Integrated Assignment Where Human Concerns Guide Technical Decisions. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1* (Lugano and Virtual Event, Switzerland) *(ICER '22)*. ACM, New York, NY, USA, 344–355.

Shannon Butler and Dewan Tanvir Ahmed. 2016. Gamification to Engage and Motivate Students to Achieve Computer Science Learning Goals. In *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*. 237–240. doi:10.1109/CSCI.2016.0053

Serena Caraco, Nelson Lojo, and Armando Fox. 2025. Fading Strategies for Parsons Problems in Intermediate Classrooms. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Nijmegen, Netherlands) *(ITiCSE 2025)*. Association for Computing Machinery, New York, NY, USA, 521–527. doi:10.1145/3724363.3729062

Richard Catrambone. 2002. The Effects of Surface and Structural Feature Matches on the Access of Story Analogs. *Journal of experimental psychology. Learning, memory, and cognition* 28 (04 2002), 318–34. doi:10.1037//0278-7393.28.2.318

Michelene T. H. Chi, Paul J. Feltovich, and Robert Glaser. 1981. Categorization and Representation of Physics Problems by Experts and Novices. *Cognitive Science* 5, 2 (1981), 121–152. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog0502_2 doi:10.1207/s15516709cog0502_2

Domenic Cicchetti. 1994. Guidelines, Criteria, and Rules of Thumb for Evaluating Normed and Standardized Assessment Instruments in Psychology. *Psychological Assessment* 6 (12 1994), 284–290. doi:10.1037/1040-3590.6.4.284

Richard Clark, David Feldon, Jeroen J. G. Van Merri enboer, Kenneth Yates, and Sean Early. 2008. Cognitive task analysis. *Handbook of Research on Educational Communications and Technology* (01 2008), 577–593.

John W. Coffey. 2013. Integrating Theoretical and Empirical Computer Science in a Data Structures Course. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. ACM, New York, NY, USA, 23–28.

Christian Collberg, Stephen G. Kobourov, and Suzanne Westbrook. 2004. AlgoVista: An Algorithmic Search Tool in an Educational Setting. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (Norfolk, Virginia, USA) *(SIGCSE '04)*. ACM, New York, NY, USA, 462–466.

Harris M. Cooper. 1988. Organizing knowledge syntheses: A taxonomy of literature reviews. *Knowledge in Society* (1988). doi:10.1007/BF03177550

Daniel Coore and Daniel Fokum. 2019. Facilitating Course Assessment with a Competitive Programming Platform. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. ACM, New York, NY, USA, 449–455.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms* (4 ed.). The MIT Press, Cambridge, Massachusetts.

Pierluigi Crescenzi, Emma Enström, and Viggo Kann. 2013. From Theory to Practice: NP-Completeness for Every CS Student. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education* (Canterbury, England, UK) *(ITiCSE '13)*. ACM, New York, NY, USA, 16–21.

Holger Danielsiek, Wolfgang Paul, and Jan Vahrenhold. 2012. Detecting and Understanding Students' Misconceptions Related to Algorithms and Data Structures. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12)*. ACM, New York, NY, USA, 21–26.

Holger Danielsiek, Laura Toma, and Jan Vahrenhold. 2017. An Instrument to Assess Self-Efficacy in Introductory Algorithms Courses. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. ACM, New York, NY, USA, 217–225.

Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. 2006. *Algorithms* (1 ed.). McGraw-Hill, Inc., USA.

Debzani Deb, Muztaba Fuad, James Etim, and Clay Gloster. 2018. MRS: Automated Assessment of Interactive Classroom Exercises. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. ACM, New York, NY, USA, 290–295.

Debzani Deb, Mohammad Muztaba Fuad, and Mallek Kanan. 2017. Creating Engaging Exercises With Mobile Response System (MRS). In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. ACM, New York, NY, USA, 147–152.

Katherine Deibel. 2005. Team formation methods for increasing interaction during in-class group work. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Caparica, Portugal) *(ITiCSE '05)*. Association for Computing Machinery, New York, NY, USA, 291–295. doi:10.1145/1067445.1067525

Alexis Dinno. 2015. Nonparametric Pairwise Multiple Comparisons in Independent Groups using Dunn's Test. *The Stata Journal* 15, 1 (2015), 292–300. arXiv:https://doi.org/10.1177/1536867X1501500117 doi:10.1177/1536867X1501500117

Daniel L Dinsmore, Patricia A Alexander, and Sandra M Loughlin. 2008. Focusing the conceptual lens on metacognition, self-regulation, and self-regulated learning. *Educational psychology review* 20 (2008), 391–409.

A.S. Donker, H. de Boer, D. Kostons, C.C. Dignath van Ewijk, and M.P.C. van der Werf. 2014. Effectiveness of learning strategy instruction on academic performance: A meta-analysis. *Educational Research Review* 11 (2014), 1–26. doi:10.1016/j.edurev.2013.11.002

James W. Drisko and Tina Maschi. 2016. *Content Analysis*. Oxford University Press.

Yuemeng Du, Andrew Luxton-Reilly, and Paul Denny. 2020. A Review of Research on Parsons Problems. In *Proceedings of the Twenty-Second Australasian Computing Education Conference* (Melbourne, VIC, Australia) *(ACE'20)*. Association for Computing Machinery, New York, NY, USA, 195–202. doi:10.1145/3373165.3373187

Emma Enström and Viggo Kann. 2017. Iteratively Intervening with the "Most Difficult" Topics of an Algorithms and Complexity Course. *ACM Trans. Comput. Educ.* 17, 1, Article 4 (jan 2017), 38 pages.

Jeff Erickson. 2019. *Algorithms* (1 ed.). self-pub.

Barbara J. Ericson, Paul Denny, James Prather, Rodrigo Duran, Arto Hellas, Juho Leinonen, Craig S. Miller, Briana B. Morrison, Janice L. Pearce, and Susan H. Rodger. 2022. Parsons Problems and Beyond: Systematic Literature Review and Empirical Study Designs. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education* (Dublin, Ireland) *(ITiCSE-WGR '22)*. Association for Computing Machinery, New York, NY, USA, 191–234. doi:10.1145/3571785.3574127

Barbara J. Ericson, James D. Foley, and Jochen Rick. 2018. Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. Association for Computing Machinery, New York, NY, USA, 60–68. doi:10.1145/3230977.3231000

Ali Erkan. 2018. The Educational Insights and Opportunities Afforded by the Nuances of Prim's and Kruskal's MST Algorithms. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018)*. ACM, New York, NY, USA, 129–134.

Ali Erkan and Diyan Gochev. 2008. An Image Background Detection Project for a Visual Exploration of DFS and BFS. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '08)*. ACM, New York, NY, USA, 483–487.

Mohammed F. Farghally, Kyu Han Koh, Hossameldin Shahin, and Clifford A. Shaffer. 2017. Evaluating the Effectiveness of Algorithm Analysis Visualizations. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 201–206. doi:10.1145/3017680.3017698

Tommy Färnqvist and Fredrik Heintz. 2016. Competition and Feedback through Automated Assessment in a Data Structures and Algorithms Course. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (Arequipa, Peru) *(ITiCSE '16)*. Association for Computing Machinery, New York, NY, USA, 130–135. doi:10.1145/2899415.2899454

J. L. Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological Bulletin* 76 (1971). Issue 5. doi:10.1037/h0031619

Association for Computing Machinery. 2024. ACM Digital Library. https://dl.acm.org.

Diana Franklin, Phillip Conrad, Bryce Boe, Katy Nilsen, Charlotte Hill, Michelle Len, Greg Dreschler, Gerardo Aldana, Paulo Almeida-Tanaka, Brynn Kiefer, Chelsea Laird, Felicia Lopez, Christine Pham, Jessica Suarez, and Robert Waite. 2013. Assessment of computer science learning in a scratch-based outreach program. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 371–376. doi:10.1145/2445196.2445304

Diana Franklin, Jean Salac, Zachary Crenshaw, Saranya Turimella, Zipporah Klain, Marco Anaya, and Cathy Thomas. 2020. Exploring Student Behavior Using the TIPP&SEE Learning Strategy. In *Proceedings of the 2020 ACM Conference on International Computing Education Research* (Virtual Event, New Zealand) *(ICER '20)*. Association for Computing Machinery, New York, NY, USA, 91–101. doi:10.1145/3372782.3406257

Iris Gaber, Amir Kirsh, and David Statter. 2023. Studied Questions in Data Structures and Algorithms Assessments. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (Turku, Finland) *(ITiCSE 2023)*. ACM, New York, NY, USA, 250–256.

Judith Gal-Ezer and Mark Trakhtenbrot. 2016. Identification and addressing reduction-related misconceptions. *Computer Science Education* 26, 2-3 (2016), 89–103. arXiv:https://doi.org/10.1080/08993408.2016.1171470 doi:10.1080/08993408.2016.1171470

Ginés Gárcia-Mateos and José Luis Fernández-Alemán. 2009. A Course on Algorithms and Data Structures Using On-Line Judging. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education* (Paris, France) *(ITiCSE '09)*. ACM, New York, NY, USA, 45–49.

GeeksforGeeks. 2025a. *Frog Jump – Climbing Stairs with Cost*. https://www.geeksforgeeks.org/dsa/minimum-cost-for-hopping-frog-to-reach-stair-n/

GeeksforGeeks. 2026b. *Jump Game – Minimum Jumps to Reach End*. https://www.geeksforgeeks.org/dsa/minimum-number-of-jumps-to-reach-end-of-a-given-array/

Edward F. Gehringer and Carolyn S. Miller. 2009. Student-generated active-learning exercises. *SIGCSE Bull.* 41, 1 (mar 2009), 81–85. doi:10.1145/1539024.1508897

Dedre Gentner, Jeffrey Loewenstein, and Leigh Thompson. 2003. Learning and Transfer: A General Role for Analogical Encoding. *J Educ Psychol* 95 (06 2003), 393–408. doi:10.1037/0022-0663.95.2.393

Dedre Gentner and Linsey A. Smith. 2013. Analogical Learning and Reasoning. In *The Oxford Handbook on Cognitive Psychology* (online ed.), Daniel Reisberg (Ed.). Oxford Academic, Oxford Library of Psychology, 668–681. doi:10.1093/oxfordhb/9780195376746.013.0042

Mary L. Gick and Keith J. Holyoak. 1980. Analogical Problem Solving. *Cognitive Psychology* 12, 3 (1980), 306–355. doi:10.1016/0010-0285(80)90013-4

Mary L. Gick and Keith J. Holyoak. 1983. Schema induction and analogical transfer. *Cognitive Psychology* 15, 1 (1983), 1–38. doi:10.1016/0010-0285(83)90002-6

David Ginat. 2001. Metacognitive Awareness Utilized for Learning Control Elements in Algorithmic Problem Solving. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education* (Canterbury, United Kingdom) *(ITiCSE '01)*. ACM, New York, NY, USA, 81–84.

David Ginat. 2003. The greedy trap and learning from mistakes. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (Reno, Navada, USA) *(SIGCSE '03)*. Association for Computing Machinery, New York, NY, USA, 11–15. doi:10.1145/611892.611920

David Ginat. 2004. Do senior CS students capitalize on recursion?. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Leeds, United Kingdom) *(ITiCSE '04)*. Association for Computing Machinery, New York, NY, USA, 82–86. doi:10.1145/1007996.1008020

David Ginat and Yoav Blau. 2017. Multiple Levels of Abstraction in Algorithmic Problem Solving. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. ACM, New York, NY, USA, 237–242.

Lasse Hakulinen. 2011. Using Serious Games in Computer Science Education. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '11)*. ACM, New York, NY, USA, 83–88.

Yeajin Ham and Brandon Myers. 2021. Learning from Team Quizzes in CS2. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 362–368. doi:10.1145/3408877.3432412

Stuart Hansen, Karen Tuinstra, Jason Pisano, and Lester I. McCann. 2003. Graph Magic: A Visual Graph Package for Students. *Computer Science Education* 13, 1 (2003), 53–66. arXiv:https://doi.org/10.1076/csed.13.1.53.13541 doi:10.1076/csed.13.1.53.13541

Carl C. Haynes and Barbara J. Ericson. 2021. Problem-Solving Efficiency and Cognitive Load for Adaptive Parsons Problems vs. Writing the Equivalent Code. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 60, 15 pages. doi:10.1145/3411764.3445292

Sarah Heckman, Jeffrey C. Carver, Mark Sherriff, and Ahmed Al-zubidy. 2021. A Systematic Literature Review of Empiricism and Norms of Reporting in Computing Education Research Literature. *ACM Trans. Comput. Educ.* 22, 1, Article 3 (oct 2021), 46 pages. doi:10.1145/3470652

Jason J Holdsworth and Siu Man Lui. 2009. GPS-Enabled Mobiles for Learning Shortest Paths: A Pilot Study. In *Proceedings of the 4th International Conference on Foundations of Digital Games* (Orlando, Florida) *(FDG '09)*. ACM, New York, NY, USA, 86–90.

Hadi Hosseini, Maxwell Hartt, and Mehrnaz Mostafapour. 2019. Learning IS Child's Play: Game-Based Learning in Computer Science Education. *ACM Trans. Comput. Educ.* 19, 3, Article 22 (jan 2019), 18 pages.

Hadi Hosseini and Laurel Perweiler. 2019. Are You Game?. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. ACM, New York, NY, USA, 866–872.

Christopher D Hundhausen, Sarah A Douglas, and John T Stasko. 2002. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing* 13, 3 (2002), 259–290.

Cruz Izu and Brad Alexander. 2018. Using unstructured practice plus reflection to develop programming/problem-solving fluency. In *Proceedings of the 20th Australasian Computing Education Conference* (Brisbane, Queensland, Australia) *(ACE '18)*. Association for Computing Machinery, New York, NY, USA, 25–34. doi:10.1145/3160489.3160496

Cruz Izu and Claudio Mirolo. 2021. Learning Transfer in Novice Programmers: A Preliminary Study. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 178–184. doi:10.1145/3430665.3456336

Cruz Izu, Cheryl Pope, and Amali Weerasinghe. 2017. On the Ability to Reason About Program Behaviour: A Think-Aloud Study. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '17)*. ACM, New York, NY, USA, 305–310.

Duane J. Jarc, Michael B. Feldman, and Rachelle S. Heller. 2000. Assessing the benefits of interactive prediction using Web-based algorithm animation courseware. In *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education* (Austin, Texas, USA) *(SIGCSE '00)*. Association for Computing Machinery, New York, NY, USA, 377–381. doi:10.1145/330908.331889

Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, New York, NY, USA.

Maria Kallia, Quintin Cutts, and Nicola Looker. 2022. When Rhetorical Logic Meets Programming: Collective Argumentative Reasoning in Problem-Solving in Programming. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1* (Lugano and Virtual Event, Switzerland) *(ICER '22)*. ACM, New York, NY, USA, 120–134.

Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Technical Report EBSE-2007-01. Software Engineering Group, Keele University and Department of Computer Science, University of Durham, Salt Lake City, UT.

Amruth N. Kumar, Rajendra K. Raj, Sherif G. Aly, Monica D. Anderson, Brett A. Becker, Richard L. Blumenthal, Eric Eaton, Susan L. Epstein, Michael Goldweber, Pankaj Jalote, Douglas Lea, Michael Oudshoorn, Marcelo Pias, Susan Reiser, Christian Servin, Rahul Simha, Titus Winters, and Qiao Xiang. 2024. *Computer Science Curricula 2023*. Association for Computing Machinery, New York, NY, USA.

Barry L. Kurtz, James B. Fenwick, Rahman Tashakkori, Ahmad Esmail, and Stephen R. Tate. 2014. Active Learning during Lecture Using Tablets. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (Atlanta, Georgia, USA) *(SIGCSE '14)*. ACM, New York, NY, USA, 121–126.

J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (1977), 159–174. http://www.jstor.org/stable/2529310

Ming-Han Lee and Guido Rößling. 2011. Toward Replicating Handmade Algorithm Visualization Behaviors in a Digital Environment: A Pre-Study. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (Darmstadt, Germany) *(ITiCSE '11)*. ACM, New York, NY, USA, 198–202.

Abe Leite and Saúl A. Blanco. 2020. Effects of Human vs. Automatic Feedback on Students' Understanding of AI Concepts and Programming Style. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 44–50. doi:10.1145/3328778.3366921

Ran Libeskind-Hadas. 2013. A derivation-first approach to teaching algorithms. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 573–578. doi:10.1145/2445196.2445366

Shu Lin, Na Meng, Dennis Kafura, and Wenxin Li. 2021. PDL: Scaffolding Problem Solving in Programming Courses. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. ACM, New York, NY, USA, 185–191.

Jonathan Liu, Erica Goodwin, and Diana Franklin. 2025a. Student Utilization of Metacognitive Strategies in Solving Dynamic Programming Problems. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1* (Pittsburgh, PA, USA) *(SIGCSE TS 2025)*. Association for Computing Machinery, New York, NY, USA, 687–693. doi:10.1145/3641554.3701888

Jonathan Liu, Erica Goodwin, and Diana Franklin. 2026. Analogical Reasoning in Undergraduate Algorithms. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1* (USA) *(SIGCSE TS 2026)*. Association for Computing Machinery, New York, NY, USA, 673–679. doi:10.1145/3770762.3772624

Jonathan Liu, Seth Poulsen, Erica Goodwin, Hongxuan Chen, Grace Williams, Yael Gertner, and Diana Franklin. [n. d.]. Dataset for "Teaching Algorithm Design: A Literature Review". https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/KTR2ZH.

Jonathan Liu, Seth Poulsen, Erica Goodwin, Hongxuan Chen, Grace Williams, Yael Gertner, and Diana Franklin. 2025b. Teaching Algorithm Design: A Literature Review. *ACM Trans. Comput. Educ.* 25, 2, Article 17 (May 2025), 20 pages. doi:10.1145/3727987

Dastyni Loksa, Amy J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects

of Explicit Guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. Association for Computing Machinery, New York, NY, USA, 1449–1461. doi:10.1145/2858036.2858252

Dastyni Loksa, Lauren Margulieux, Brett A. Becker, Michelle Craig, Paul Denny, Raymond Pettit, and James Prather. 2022. Metacognition and Self-Regulation in Programming Education: Theories and Exemplars of Use. *ACM Trans. Comput. Educ.* 22, 4, Article 39 (sep 2022), 31 pages. doi:10.1145/3487050

Joan M. Lucas. 2015. Illustrating the Interaction of Algorithms and Data Structures Using the Matching Problem. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 247–252. doi:10.1145/2676723.2677212

Michael Luu, Matthew Ferland, Varun Nagaraj Rao, Arushi Arora, Randy Huynh, Frederick Reiber, Jennifer Wong-Ma, and Michael Shindler. 2023. What is an Algorithms Course? Survey Results of Introductory Undergraduate Algorithms Courses in the U.S.. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. ACM, New York, NY, USA, 284–290.

Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, et al. 2018. Introductory programming: a systematic literature review. In *Proceedings companion of the 23rd annual ACM conference on innovation and technology in computer science education*. Association for Computing Machinery, New York, NY, USA, 55–106.

Lauri Malmi, Judy Sheard, Päivi Kinnunen, Simon, and Jane Sinclair. 2019. Computing Education Theories: What Are They and How Are They Used?. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. ACM, New York, NY, USA, 187–197.

Byron B. Marshall, Hsinchun Chen, Rao Shen, and Edward A. Fox. 2006. Moving Digital Libraries into the Student Learning Space: The GetSmart Experience. *J. Educ. Resour. Comput.* 6, 1 (mar 2006), 2–es.

Lester I. McCann. 2004. Contemplate Sorting with Columnsort. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (Norfolk, Virginia, USA) *(SIGCSE '04)*. ACM, New York, NY, USA, 166–169.

Dinesh Mehta, Tina Kouri, and Irene Polycarpou. 2012. Forming project groups while learning about matching and network flows in algorithms. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (Haifa, Israel) *(ITiCSE '12)*. Association for Computing Machinery, New York, NY, USA, 40–45. doi:10.1145/2325296.2325310

Kane Meissel and Esther Yao. 2024. Using Cliff's Delta as a Non-Parametric Effect Size Measure: An Accessible Web App and R Tutorial. *Practical Assessment* 29 (01 2024). doi:10.7275/pare.1977

Patricia H. Miller, Frank S. Kessel, and John H. Flavell. 1970. Thinking about People Thinking about People Thinking about...: A Study of Social Cognitive Development. *Child Development* 41, 3 (1970), 613–623. http://www.jstor.org/stable/1127211

Diba Mirza, Phillip T. Conrad, Christian Lloyd, Ziad Matni, and Arthur Gatin. 2019. Undergraduate Teaching Assistants in Computer Science: A Systematic Literature Review. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. Association for Computing Machinery, New York, NY, USA, 31–40. doi:10.1145/3291279.3339422

Orna Muller. 2005. Pattern oriented instruction and the enhancement of analogical reasoning. In *Proceedings of the First International Workshop on Computing Education Research* (Seattle, WA, USA) *(ICER '05)*. Association for Computing Machinery, New York, NY, USA, 57–67. doi:10.1145/1089786.1089792

Orna Muller and Bruria Haberman. 2008. Supporting abstraction processes in problem solving through pattern-oriented instruction. *Computer Science Education* 18, 3 (2008), 187–212. arXiv:https://doi.org/10.1080/08993400802332548 doi:10.1080/08993400802332548

Thomas L. Naps, James R. Eagan, and Laura L. Norton. 2000. JHAVÉ—an Environment to Actively Engage Students in Web-Based Algorithm Visualizations. In *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education* (Austin, Texas, USA) *(SIGCSE '00)*. ACM, New York, NY, USA, 109–113.

Alannah Oleson, Benjamin Xie, Jean Salac, Jayne Everson, F. Megumi Kivuva, and Amy J. Ko. 2022. A Decade of Demographics in Computing Education Research: A Critical Review of Trends in Collection, Reporting, and Use. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1* (Lugano and Virtual Event, Switzerland) *(ICER '22)*. ACM, New York, NY, USA, 323–343.

Taylor & Francis Online. 2024. Computer Science Education. https://www.tandfonline.com/journals/ncse20.

Eva Ostertagova, Oskar Ostertag, and Jozef Kováč. 2014. Methodology and Application of the Kruskal-Wallis Test. *Applied Mechanics and Materials* 611 (08 2014), 115–120.

Roy P. Pargas. 2006. Reducing Lecture and Increasing Student Activity in Large Computer Science Courses. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITICSE '06)*. ACM, New York, NY, USA, 3–7.

Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (Hobart, Australia) *(ACE '06)*. Australian Computer Society, Inc., AUS, 157–163.

Wolfgang Paul and Jan Vahrenhold. 2013. Hunting High and Low: Instruments to Detect Misconceptions Related to Algorithms and Data Structures. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. ACM, New York, NY, USA, 29–34.

David Pengelley, Inna Pivkina, Desh Ranjan, and Karen Villaverde. 2006. A Project in Algorithms Based on a Primary Historical Source about Catalan Numbers. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (Houston, Texas, USA) *(SIGCSE '06)*. ACM, New York, NY, USA, 318–322.

David Perkins and Martin Fay. 1985. *Fragile Knowledge and Neglected Strategies in Novice Programmers*. Technical Report. Educational Technology Center, Harvard Graduate School of Education.

Vinhthuy Phan and Eric Hicks. 2018. Code4Brownies: An Active Learning Solution for Teaching Programming and Problem Solving in the Classroom. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018)*. ACM, New York, NY, USA, 153–158.

George Polya and John H. Conway. 2004. *How to Solve It: A New Aspect of Mathematical Method*. Princeton University Press. `https://books.google.com/books?id=z_hsbu9kyQQC`

Seth Poulsen, Yael Gertner, Benjamin Cosman, Matthew West, and Geoffrey L. Herman. 2023. Efficiency of Learning from Proof Blocks Versus Writing Proofs. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 472–478. doi:`10.1145/3545945.3569797`

Seth Poulsen, Mahesh Viswanathan, Geoffrey L. Herman, and Matthew West. 2022. Proof Blocks: Autogradable Scaffolding Activities for Learning to Write Proofs. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1* (Dublin, Ireland) *(ITiCSE '22)*. Association for Computing Machinery, New York, NY, USA, 428–434. doi:`10.1145/3502718.3524774`

James Prather, Brett A Becker, Michelle Craig, Paul Denny, Dastyni Loksa, and Lauren Margulieux. 2020. What do we think we think we are doing? Metacognition and self-regulation in programming. In *Proceedings of the 2020 ACM conference on international computing education research*. Association for Computing Machinery, New York, NY, USA, 2–13.

James Prather, Raymond Pettit, Brett A. Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. 2019. First Things First: Providing Metacognitive Scaffolding for Interpreting Problem Prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 531–537. doi:10.1145/3287324.3287374

James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. Association for Computing Machinery, New York, NY, USA, 41–50. doi:10.1145/3230977.3230981

Justus Randolph. 2009. A Guide to Writing the Dissertation Literature Review. *Practical Assessment, Research, and Evaluation* 14 (2009). Issue 1.

Richard Rasala, Viera K. Proulx, and Harriet J. Fell. 1994. From animation to analysis in introductory computer science. In *Proceedings of the Twenty-Fifth SIGCSE Symposium on Computer Science Education* (Phoenix, Arizona, USA) *(SIGCSE '94)*. Association for Computing Machinery, New York, NY, USA, 61–65. doi:10.1145/191029.191057

Samuel A. Rebelsky. 2005. The New Science Students in Too Much, Too Soon an Abbreviated, Accelerated, Constructivist, Collaborative, Introductory Experience in CS. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (St. Louis, Missouri, USA) *(SIGCSE '05)*. ACM, New York, NY, USA, 312–316.

Robert H. Riffenburgh. 2006. Chapter 20 - Tests on the Distribution Shape of Continuous Data. In *Statistics in Medicine* (2 ed.), Robert H. Riffenburgh (Ed.). Academic Press, Burlington, 369–386. doi:10.1016/B978-012088770-5/50060-5

Brian H. Ross. 1984. Remindings and their effects in learning a cognitive skill. *Cognitive Psychology* 16, 3 (1984), 371–416. doi:10.1016/0010-0285(84)90014-8

Kate Sanders, Judy Sheard, Brett A. Becker, Anna Eckerdal, Sally Hamouda, and Simon. 2019. Inferential Statistics in Computing Education Research: A Methodological Review. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. Association for Computing Machinery, New York, NY, USA, 177–185. doi:10.1145/3291279.3339408

Clifford A. Shaffer, Matthew Cooper, and Stephen H. Edwards. 2007. Algorithm Visualization: A Report on the State of the Field. *SIGCSE Bull.* 39, 1 (mar 2007), 150–154.

Clifford A. Shaffer, Matthew L. Cooper, Alexander Joel D. Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H. Edwards. 2010. Algorithm Visualization: The State of the Field. *ACM Trans. Comput. Educ.* 10, 3, Article 9 (aug 2010), 22 pages.

Donald Sharpe. 2015. Your Chi-Square Test is Statistically Significant: Now What? *Practical Assessment, Research & Evaluation* 20, 8 (2015).

Judy Sheard, S. Simon, Margaret Hamilton, and Jan Lönnberg. 2009. Analysis of research into the teaching and learning of programming. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop* (Berkeley, CA, USA) *(ICER '09)*. Association for Computing Machinery, New York, NY, USA, 93–104. doi:10.1145/1584322.1584334

Michael Shindler, Natalia Pinpin, Mia Markovic, Frederick Reiber, Jee Hoon Kim, Giles Pierre Nunez Carlos, Mine Dogucu, Mark Hong, Michael Luu, Brian Anderson, Aaron Cote, Matthew Ferland, Palak Jain, Tyler LaBonte, Leena Mathur, Ryan Moreno, and Ryan Sakuma. 2022. Student misconceptions of dynamic programming: a replication study. *Computer Science Education* 32, 3 (2022), 288–312. arXiv:https://doi.org/10.1080/08993408.2022.2079865 doi:10.1080/08993408.2022.2079865

Simon. 2007a. A Classification of Recent Australasian Computing Education Publications. *Computer Science Education* 17, 3 (2007), 155–169. arXiv:https://doi.org/10.1080/08993400701538021 doi:10.1080/08993400701538021

Simon. 2007b. Koli Calling comes of age: an analysis. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88* (Koli National Park, Finland) *(Koli Calling '07)*. Australian Computer Society, Inc., AUS, 119–126.

John T. Stasko. 1997. Using Student-Built Algorithm Animations as Learning Aids. In *Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education* (San Jose, California, USA) *(SIGCSE '97)*. ACM, New York, NY, USA, 25–29.

Ahmad Taherkhani, Ari Korhonen, and Lauri Malmi. 2012. Automatic Recognition of Students' Sorting Algorithm Implementations in a Data Structures and Algorithms Course. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '12)*. ACM, New York, NY, USA, 83–92.

David Scot Taylor, Andrei F. Lurie, Cay S. Horstmenn, Menko B. Johnson, Sean K. Sharma, and Edward C. Yin. 2009. Predictive vs. Passive Animation Learning Tools. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (Chattanooga, TN, USA) *(SIGCSE '09)*. ACM, New York, NY, USA, 494–498.

Donna Teague and Raymond Lister. 2014. Manifestations of preoperational reasoning on similar programming tasks. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148* (Auckland, New Zealand) *(ACE '14)*. Australian Computer Society, Inc., AUS, 65–74.

James D. Teresco, Razieh Fathi, Lukasz Ziarek, MariaRose Bamundo, Arjol Pengu, and Clarice F. Tarbay. 2018. Map-Based Algorithm Visualization with METAL Highway Data.

In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. ACM, New York, NY, USA, 550–555.

David R. Thomas. 2006. A General Inductive Approach for Analyzing Qualitative Evaluation Data. *American Journal of Evaluation* 27, 2 (2006), 237–246. arXiv:https://doi.org/10.1177/1098214005283748 doi:10.1177/1098214005283748

Laura Toma and Jan Vahrenhold. 2018. Self-Efficacy, Cognitive Load, and Emotional Reactions in Collaborative Algorithms Labs - A Case Study. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. ACM, New York, NY, USA, 1–10.

Jan Vahrenhold and Wolfgang Paul. 2014. Developing and validating test items for first-year computer science courses. *Computer Science Education* 24, 4 (2014), 304–333. arXiv:https://doi.org/10.1080/08993408.2014.970782 doi:10.1080/08993408.2014.970782

Marten van den Berg, Bastiaan Heeren, and Ebrahim Rahimi. 2024. Parsons Problems for Equivalence Proofs in Logic. In *Proceedings of the 24th Koli Calling International Conference on Computing Education Research (Koli Calling '24)*. Association for Computing Machinery, New York, NY, USA, Article 10, 12 pages. doi:10.1145/3699538.3699551

Tammy VanDeGrift. 2017. POGIL Activities in Data Structures: What do Students Value?. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 597–602. doi:10.1145/3017680.3017697

Sharon Vaughn, Janette K. Klingner, Elizabeth A. Swanson, Alison G. Boardman, Greg Roberts, Sarojani S. Mohammed, and Stephanie J. Stillman-Spisak. 2011. Efficacy of Collaborative Strategic Reading With Middle School Students. *American Educational Research Journal* 48, 4 (2011), 938–964. arXiv:https://doi.org/10.3102/0002831211410305 doi:10.3102/0002831211410305

J. Ángel Velázquez-Iturbide. 2012. Refinement of an Experimental Approach Tocomputer-Based, Active Learning of Greedy Algorithms. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (Haifa, Israel) *(ITiCSE '12)*. ACM, New York, NY, USA, 46–51.

J. Ángel Velázquez-Iturbide. 2013. An Experimental Method for the Active Learning of Greedy Algorithms. *ACM Trans. Comput. Educ.* 13, 4, Article 18 (nov 2013), 23 pages. doi:10.1145/2534972

J. Ángel Velázquez-Iturbide. 2019. Students' Misconceptions of Optimization Algorithms. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE '19)*. ACM, New York, NY, USA, 464–470.

J. Ángel Velázquez-Iturbide, Isidoro Hernán-Losada, and Antonio Pérez-Carrasco. 2016. A "Multiple Executions" Technique of Visualization. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (Arequipa, Peru) *(ITiCSE '16)*. Association for Computing Machinery, New York, NY, USA, 59–64. doi:10.1145/2899415.2899451

J. Ángel Velázquez-Iturbide and Antonio Pérez-Carrasco. 2016. Systematic Development of Dynamic Programming Algorithms Assisted by Interactive Visualization. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (Arequipa, Peru) *(ITiCSE '16)*. ACM, New York, NY, USA, 71–76.

Valdemar Švábenský, Jan Vykopal, and Pavel Čeleda. 2020. What Are Cybersecurity Education Papers About? A Systematic Literature Review of SIGCSE and ITiCSE Conferences. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 2–8. doi:10.1145/3328778.3366816

Robbie Weber. 2023. Using Alternative Grading in a Non-Major Algorithms Course. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. ACM, New York, NY, USA, 638–644.

Matthew West, Geoffrey L. Herman, and Craig B. Zilles. 2015. PrairieLearn: Mastery-based Online Problem Solving with Adaptive Scoring and Recommendations Driven by Machine Learning. https://api.semanticscholar.org/CorpusID:64230337

Titus Winters and Tom Payne. 2006. Closing the Loop on Test Creation: A Question Assessment Mechanism for Instructors. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (Houston, Texas, USA) *(SIGCSE '06)*. ACM, New York, NY, USA, 169–170.

Anthony Wirth and Michael Bertolacci. 2006. New algorithms research for first year students. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITICSE '06)*. Association for Computing Machinery, New York, NY, USA, 128–132. doi:10.1145/1140124.1140160

Zihan Wu, Barbara J. Ericson, and Christopher Brooks. 2023. Using Micro Parsons Problems to Scaffold the Learning of Regular Expressions. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (Turku, Finland) *(ITiCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 457–463. doi:10.1145/3587102.3588853

Zihan Wu and IV Smith, David H. 2024. Evaluating Micro Parsons Problems as Exam Questions. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 674–680. doi:10.1145/3649217.3653583

Jason Xia and Craig Zilles. 2023. Using Context-Free Grammars to Scaffold and Automate Feedback in Precise Mathematical Writing. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 479–485. doi:10.1145/3545945.3569728

Shamama Zehra, Aishwarya Ramanathan, Larry Yueli Zhang, and Daniel Zingaro. 2018. Student Misconceptions of Dynamic Programming. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. ACM, New York, NY, USA, 556–561.

Andreas Zeller. 2000. Making Students Read and Review Code. In *Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSEconference on Innovation and Technology in Computer Science Education* (Helsinki, Finland) *(ITiCSE '00)*. ACM, New York, NY, USA, 89–92.