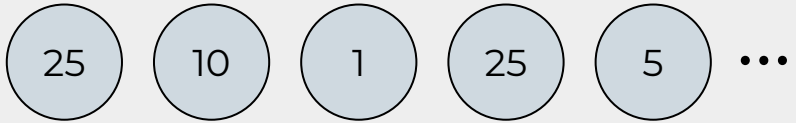
Investigating Student Usage of Metacognitive Problem-Solving Strategies in Algorithm Design Problems

Jonathan Liu
Dissertation Defense, University of Chicago
7/24/2026

Investigating Student Usage of Metacognitive Problem-Solving Strategies in Algorithm Design Problems

Example Problem: Coin Row

You have n coins in a row, with values c_1, c_2, ..., c_n. You want to pick up coins to maximize your total value, but you can not pick up two adjacent coins.



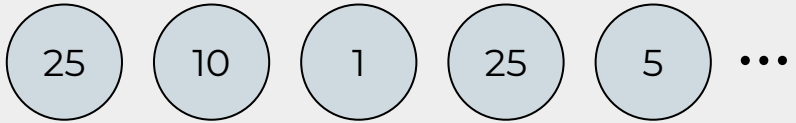
Example Solution

```
def max_sum(C): # Suppose C denotes a list of n coin values.
    # Design Step 1: Select a reasonable subproblem structure.
    coin_sum = 1D array with length n

    # Design Step 4: Initialize base case(s) for cost.
    coin_sum[0] = C[0]
    coin_sum[1] = max(C[0], C[1])

    # Design Step 3: Select the iteration order over the subproblems.
    for i in 2...n-1:

        # Design Step 2: Use a recurrence relationship to fill in
        # subproblem values.
        coin_sum[i] = max(coin_sum[i-1],
                          coin_sum[i-2] + C[i])

    # Step 5: Return a solution.
    return coin_sum[n-1]
```

Investigating Student Usage of Metacognitive Problem-Solving Strategies in Algorithm Design Problems

Example Problem: Coin Row

You have n coins in a row, with values c_1, c_2, ..., c_n. You want to pick up coins to maximize your total value, but you can not pick up two adjacent coins.



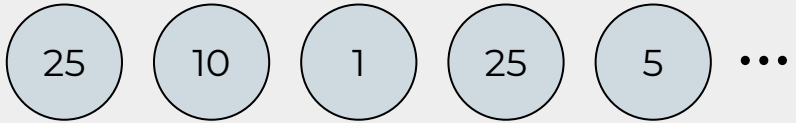
Why should we study algorithms?

- Critical skill in the undergraduate CS degree, with practical implications especially for software engineering
- Algorithms is often the first CS course applying math to CS topics; both math and CS have known equity gaps
  - Recent work suggests this is true in algorithms too [Braught et al., In Submission]

Investigating Student Usage of Metacognitive Problem-Solving Strategies in Algorithm Design Problems

Example Problem: Coin Row

You have n coins in a row, with values c_1, c_2, ..., c_n. You want to pick up coins to maximize your total value, but you can not pick up two adjacent coins.



What are Metacognitive Strategies?

Metacognition: The thinking that guides your thinking.

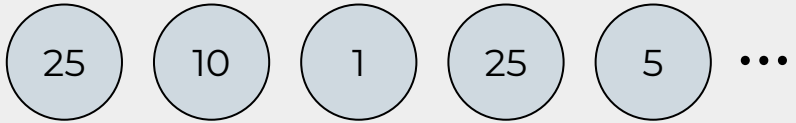
- How do you start a problem?
- What do you do when you're stuck?
- How do you use what you've seen before to help you?

In K-12 and CS1, explicit metacognitive instruction has shown to boost performance and equity

Investigating Student Usage of Metacognitive Problem-Solving Strategies in Algorithm Design Problems

Example Problem: Coin Row

You have n coins in a row, with values c_1, c_2, ..., c_n. You want to pick up coins to maximize your total value, but you can not pick up two adjacent coins.



Why study student usage?

- Metacognitive skills tend to be absent from algorithms instruction, and students are expected to develop it naturally as they progress through the course
- Algorithms is notoriously difficult, and some prior work hints that metacognitive skills are not being properly developed during the course

“*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? (1977)*

Overarching Goal

Establish foundational knowledge about teaching and learning metacognitive strategies in the algorithm design context.

Dissertation Overview

Literature

What do we know about learning and teaching algorithm design? [TOCE ‘25]

Baseline

What metacognitive strategies are students adopting on their own? [SIGCSE ‘25]

Potential

How does metacognitive strategy use relate to algorithm design outcomes? [SIGCSE ‘26]

Intervention

How do we effectively introduce metacognitive strategies to students? [In Submission]

Dissertation Overview

Literature

What do we know about learning and teaching algorithm design? [TOCE ‘25]

Baseline

What metacognitive strategies are students adopting on their own? [SIGCSE ‘25]

Potential

How does metacognitive strategy use relate to algorithm design outcomes? [SIGCSE ‘26]

Intervention

How do we effectively introduce metacognitive strategies to students? [In Submission]

Literature: What do we know about learning and teaching algorithm design?

Teaching Algorithm Design: A Literature Review

Jonathan Liu, Seth Poulsen, Erica Goodwin, Hongxuan Chen, Grace Williams, Yael Gertner, Diana Franklin

TOCE 2025

Methodology

We conducted a systematic literature review across all SIGCSE-sponsored conferences and major journals (TOCE, CSE) in CS Education.

We found 102 papers doing research on undergrad. algorithms.

- For scale, SIGCSE TS alone has published ~4707 papers.

We tagged each paper by topic and methodology.

We synthesized the results of each paper, sorted into major categories.

Findings: Algorithm Design Skills

After the algorithms course, many students don't successfully apply learned strategies

- Abstraction [Ginat and Blau '17, Izu et al. '17]
- Greedy [Ginat '03]
- Recursion [Ginat '04]
- Reduction [Armoni et al. '09]

Takeaway: Students are learning algorithms content, but studies repeatedly show that students aren't applying them well later in unfamiliar situations, hinting at a gap in metacognitive skills.

Dissertation Overview

Literature

The existing research on algorithm design strategies hint at a gap in metacognitive skills.

Baseline

What metacognitive strategies are students adopting on their own? [SIGCSE '25]

Potential

How does metacognitive strategy use relate to algorithm design outcomes? [SIGCSE '26]

Intervention

How do we effectively introduce metacognitive strategies to students? [In Submission]

Dissertation Overview

Literature

The existing research on algorithm design strategies hint at a gap in metacognitive skills.

Baseline

What metacognitive strategies are students adopting on their own? [SIGCSE '25]

Potential

How does metacognitive strategy use relate to algorithm design outcomes? [SIGCSE '26]

Intervention

How do we effectively introduce metacognitive strategies to students? [In Submission]

Baseline: What metacognitive strategies are students adopting on their own?

Student Utilization of Metacognitive Strategies in Solving Dynamic Programming Problems

Jonathan Liu, Erica Goodwin, Diana Franklin

SIGCSE TS 2025

Motivation

Why study the baseline at all?

- We have evidence *hinting* at poor metacognitive development, but no direct measurement or characterization.
- Novices do not use metacognitive strategies well, but to what extent are algorithms students novices?

Why study Dynamic Programming (DP)?

- Often cited as one of the most difficult paradigms
- At UChicago, taught near the end of the course

Theory: Process Steps

Process Steps are a classic metacognitive strategy consisting of a generic “roadmap” for how to approach a type of problem.

For example, in CS1, this may look like: [Loksa et al. ‘16]

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

Research Questions

RQ1: Strategy Development

What metacognitive approaches do students utilize to solve DP problems?

RQ2: Effect of Guidance

To what extent does guidance about metacognitive strategies benefit students solving while DP problems?

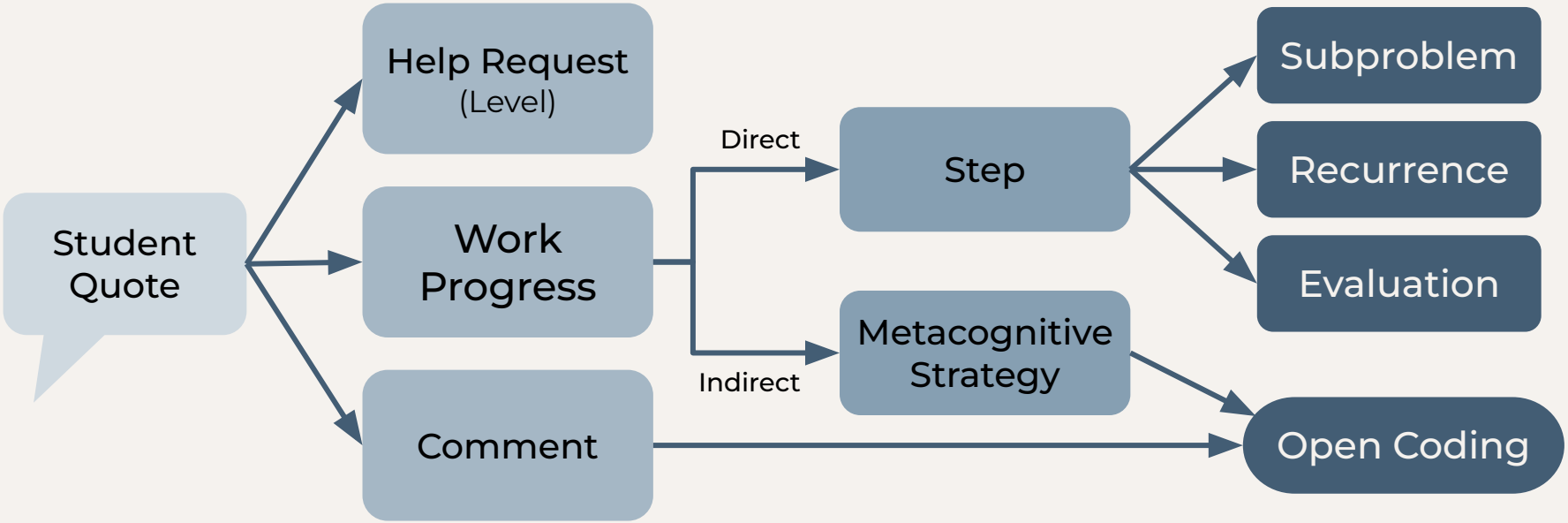
RQ3: Obstacles to Usage

What obstacles do students face when using metacognitive strategies in DP problems?

Methods

- 30 think-aloud interviews with students from the algorithms course here at UChicago, solving DP problems
- Gave scripted guidance for two metacognitive strategies (one of which was Process Steps) when appropriate
- Inductively coded transcripts for **metacognitive strategy usage**: instances where students said something about their approach (rather than about the problem itself)

Methodology: Interview Coding



Major Findings

RQ1: Strategy Development

Students are picking up these strategies on their own, but can be incorrect or inconsistent in their understanding and usage.

Major Findings

RQ1: Strategy Development

Students are picking up these strategies on their own, but can be incorrect or inconsistent in their understanding and usage.

Case Study: Process Steps

Students could often name some keywords/steps to use, but struggled with incomplete understandings of the step names.

> "When I think dynamic programming I'm thinking like splitting it into two *subproblems* and then like maximizing those *subproblems*." (Conflation with Divide-and-Conquer)

Major Findings

RQ1: Strategy Development

Students are picking up these strategies on their own, but can be incorrect or inconsistent in their understanding and usage.

Case Study: Referring to Previously-Seen Problems

14/30 students used this strategy, but <u>many relied on insights from prior problems that were inapplicable to the new problem.</u>

> "I think I've solved more 2D problems... [so] I ended up defaulting to 2D... even though I realized I didn't need to."

Major Findings

RQ1: Strategy Development

Students are picking up these strategies on their own, but can be incorrect or inconsistent in their understanding and usage.

RQ2: Effect of Guidance

Short-term guidance is helpful, but not enough to replace content knowledge, ensure correct usage, or promote long-term adoption.

Major Findings

RQ1: Strategy Development

Students are picking up these strategies on their own, but can be incorrect or inconsistent in their understanding and usage.

RQ2: Effect of Guidance

Short-term guidance is helpful, but not enough to replace content knowledge, ensure correct usage, or promote long-term adoption.

RQ3: Obstacles to Usage

Metacognitive strategy usage and obstacles inherently vary by participant, necessitating longer-term support for adoption.

Dissertation Overview

Literature

The existing research on algorithm design strategies hint at a gap in metacognitive skills.

Baseline

Students are adopting metacognitive strategies, but face barriers to productive use.

Potential

How does metacognitive strategy use relate to algorithm design outcomes? [SIGCSE '26]

Intervention

How do we effectively introduce metacognitive strategies to students? [In Submission]

Dissertation Overview

Literature

The existing research on algorithm design strategies hint at a gap in metacognitive skills.

Baseline

Students are adopting metacognitive strategies, but face barriers to productive use.

Potential

How does metacognitive strategy use relate to algorithm design outcomes? [SIGCSE '26]

Intervention

How do we effectively introduce metacognitive strategies to students? [In Submission]

Potential: How does metacognitive strategy use relate to algorithm design outcomes?

Analogical Reasoning in Undergraduate Algorithms

Jonathan Liu, Erica Goodwin, Diana Franklin

SIGCSE TS 2026

Background: Analogical Reasoning

Analogical reasoning: drawing meaningful connections (*analogies*) to previously-seen problems to inform the development of a solution to a new problem. [Gick and Holyoak '80]

Novices (in any subject) struggle with analogical reasoning.

- In general problem-solving [Gick and Holyoak '80, Ross '84]
- In CS1 [Izu and Mirolo '21, Perkins and Fay '85, Teague and Lister '14]
- In algorithms [Izu et al. '17]

Novices tend to mentally categorize situations based on surface features (related to context), while experts categorize situations based on their structural features (related to structure or solution). [Catrambone '02, Chi et al., '81]

Study Motivation

It is worth investigating the relationship (if any!) between analogical reasoning and success in algorithm design.

What we know:

- Post-algorithms students still often fail to do analogical reasoning while attempting algorithms problems [Ginat ‘04, Armoni ‘09]
- The example-heavy structure of algorithms *implies* the importance of analogical reasoning.

What we don’t know:

- At a broader scale, to what extent is analogical reasoning linked to algorithm design success?

Methods: Data Collection

Course Setting

- Algorithms at UChicago, 50 consenting students
- Instructors integrated some analogical reasoning examples into coursework

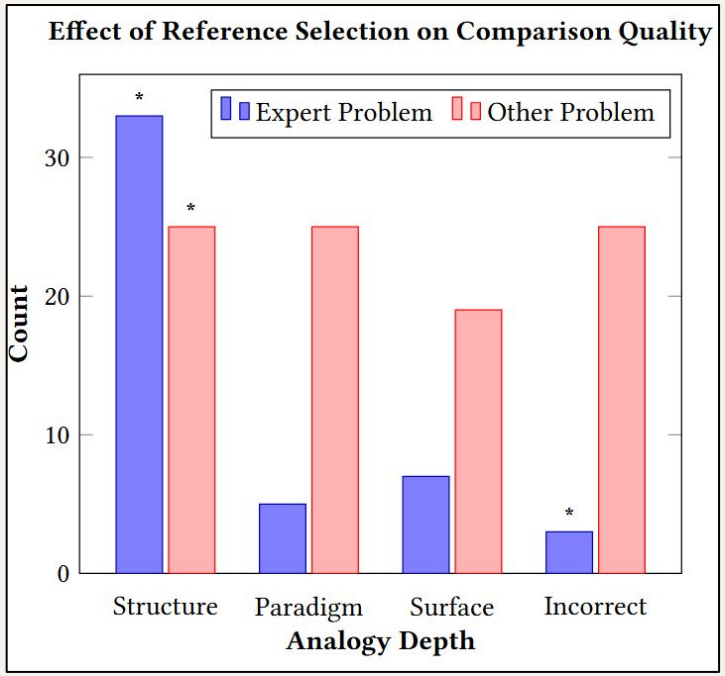
Data

3 algorithm design exam questions, with added reflection:

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.

Analogies were tagged *Structural* if they contained specific knowledge about the problem structure.

Major Finding: Problem Selection (RQ3)

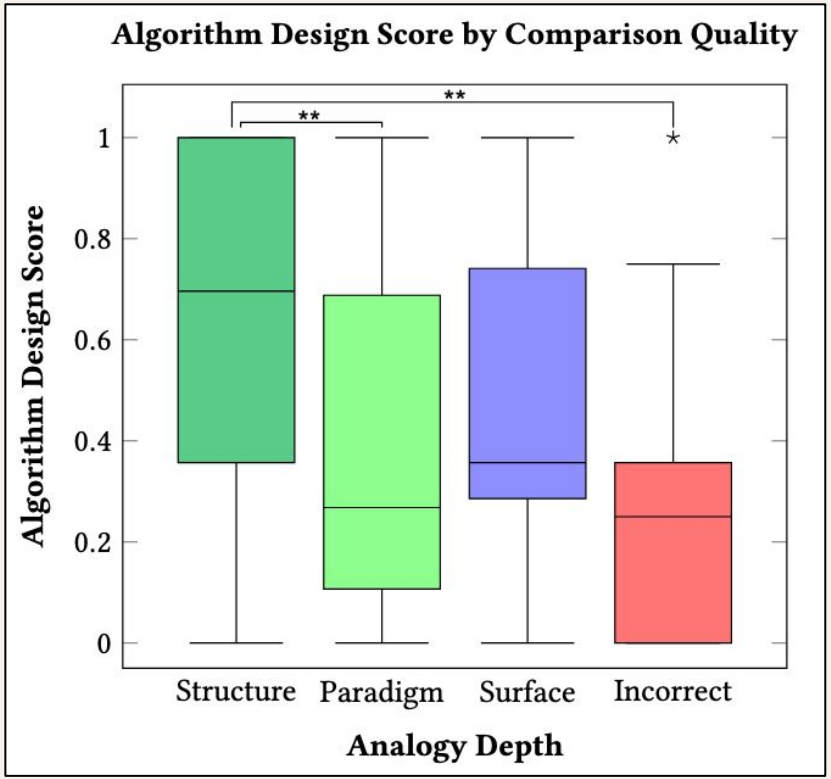


Course staff released solutions, which included example analogies.

Takeaway: When the student aligned with staff on problem selection, the analogy was far more likely to be structural.

*: Critical Residual for Chi-Squared Test of Independence

Major Finding: Depth vs. Score (RQ4)



Kruskal-Wallis Test, $p<0.00001$

Each bar represents scores for a different analogy depth. *Structure* is the left-most.

Takeaway:
Structure-level analogies were associated with significantly higher scores.

Takeaways

RQ1/2: Analogy Quality

Analogy depth was split fairly evenly.

RQ3: Problem Selection

Students whose selections aligned with TAs were significantly more likely to make a structure-level comparison.

RQ4: Does Depth Matter?

Students who made structure-level comparisons performed significantly better in designing the algorithm.

Overall Takeaway: Analogical reasoning is meaningfully tied to success in algorithms courses.

Dissertation Overview

Literature

The existing research on algorithm design strategies hint at a gap in metacognitive skills.

Baseline

Students are adopting metacognitive strategies, but face barriers to productive use.

Potential

Metacognitive strategy development is closely tied to success in algorithm design.

Intervention

How do we effectively introduce metacognitive strategies to students? [In Submission]

Dissertation Overview

Literature

The existing research on algorithm design strategies hint at a gap in metacognitive skills.

Baseline

Students are adopting metacognitive strategies, but face barriers to productive use.

Potential

Metacognitive strategy development is closely tied to success in algorithm design.

Intervention

How do we effectively introduce metacognitive strategies to students? [In Submission]

Intervention: How do we effectively introduce metacognitive strategies to students?

Design-level and Implementation-level Scaffolding for Dynamic Programming Problems

Jonathan Liu, Hongxuan Chen, Michael Shindler, Yael Gertner, Jeff Erickson, Diana Franklin. In Submission.

Theory: Process Steps, Revisited

As a reminder, here is an example of CS1 process steps: [Loksa et al. '16]

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

Explicitly providing novices with guidance around process steps boosts self-orientation and self-efficacy. This, in turn, increases productivity, engagement, help-seeking outcomes, and learning.
[Donker et al. '14, Loksa et al. '16, Prather et al. '19]

Process Steps for Dynamic Programming

Example Problem: Coin Row

You have n coins in a row, with values c_1, c_2, ..., c_n. You want to pick up coins to maximize your total value, but you can not pick up two adjacent coins.

25 10 1 25 5 ...

RQ: Does scaffolding these steps help adoption?

DP 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.

Process Steps for Dynamic Programming

Example Solution: Coin Row

```
def max_sum(C): # Suppose C denotes a list of n coin values.
    # Design Step 1: Select a reasonable subproblem structure.
    coin_sum = 1D array with length n

    # Design Step 4: Initialize base case(s) for cost.
    coin_sum[0] = C[0]
    coin_sum[1] = max(C[0], C[1])

    # Design Step 3: Select the iteration order over the subproblems.
    for i in 2...n-1:

        # Design Step 2: Use a recurrence relationship to fill in
        # subproblem values.
        coin_sum[i] = max(coin_sum[i-1],
                          coin_sum[i-2] + C[i])

    # Step 5: Return a solution.
    return coin_sum[n-1]
```

DP 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.

Process Steps for Dynamic Programming

Example Solution: Coin Row

```
def max_sum(C): # Suppose C denotes a list of n coin values.
    # Design Step 1: Select a reasonable subproblem structure.
    coin_sum = 1D array with length n

    # Design Step 4: Initialize base case(s) for cost.
    coin_sum[0] = C[0]
    coin_sum[1] = max(C[0], C[1])

    # Design Step 3: Select the iteration order over the subproblems.
    for i in 2...n-1:

        # Design Step 2: Use a recurrence relationship to fill in
        # subproblem values.
        coin_sum[i] = max(coin_sum[i-1],
                          coin_sum[i-2] + C[i])

    # Step 5: Return a solution.
    return coin_sum[n-1]
```

DP 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.

Implementation from Design Steps

```
def max_sum(C): # Suppose C denotes a list of n coin values.
    # Design Step 1: Select a reasonable subproblem structure.
    coin_sum = 1D array with length n

    # Design Step 4: Initialize base case(s) for cost.
    coin_sum[0] = C[0]
    coin_sum[1] = max(C[0], C[1])

    # Design Step 3: Select the iteration order over the subproblems.
    for i in 2...n-1:

        # Design Step 2: Use a recurrence relationship to fill in
        # subproblem values.
        coin_sum[i] = max(coin_sum[i-1],
                          coin_sum[i-2] + C[i])

    # Step 5: Return a solution.
    return coin_sum[n-1]
```

The steps are not implemented in the same order as they are designed!

Prior work shows that students face barriers during both design and implementation. [Shindler et al. ‘22]

RQ: Should implementation be scaffolded too?

Overarching Research Question

How do different levels of scaffolding for dynamic programming process steps affect how students adopt the steps and how they develop dynamic programming algorithms?

Metrics for Comparison

RQ1: Steps Recall

To what extent does process step scaffolding impact **process step recall** for DP algorithm design problems?

RQ2: Performance

To what extent does process step scaffolding impact **performance** on DP algorithm design problems?

RQ3: Practice Pace

To what extent does process step scaffolding impact **practice pace** for DP algorithm design problems?

Methods: Intervention Structure

66 students learning Dynamic Programming were recruited to participate in a practice problem session.

Problem Session Structure (Randomized Control Trial)

1. **Practice Problems:** Students are given the steps as they work through 3 algorithm design problems. Scaffolding within the problems varies by condition:
   - **Control** (none)
   - **Design** Scaffolding
   - **Design+Implementation (D+I)** Scaffolding
2. **Knowledge Check**

Condition 1: Control

▶ **Click for Full Problem Statement**

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 you can pick up from coin 0 to coin n, following the rules described by the problem.

Remember that a Dynamic Programming algorithm can be designed with the following steps:

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

You may use pseudocode or simply describe your algorithm in English, but regardless, a reader with coding ability should be able to implement the algorithm directly from your answer. The box below supports markdown and LaTeX, but plain text is completely fine too.

If you are unable to completely solve the problem, write in as much as you figured out, separated by the design steps above if desired. After 10 minutes (or some time), if you are stumped, you can submit your partial work and receive the solution with explanation. You can use that to help you solve the next problem.

coin-row.md

1

Preview

Problem Statement + List of Steps

Implementation Box

Detailed Solution

Condition 2: Design Scaffolding

▼ **Click for Full Problem Statement**

Suppose you are given n coins in a row whose values are positive integers $c_0, \ldots, c_{n-1}$. You may pick up coins from this row, but you can not pick up two adjacent coins.

For example, given a row of coins $\{10, 4, 2, 8, 3\}$, one cannot pick up both 10 and 4 since they are adjacent. One possible way to pick up is $\{10, 2, 3\}$ with $sum = 15$, whereas the best possible way (resulting in maximum sum) is to pick $\{10, 8\}$, which gives you 18 in total.

We are walking through the following steps to design a Dynamic Programming algorithm:

- **Step 1:** Select a reasonable subproblem structure.
  - We use subproblems of the form $Coin\_Sum[i]$, where each subproblem is the maximum sum of values from coin 0 up to coin i, because these subproblems are a smaller version of the original problem.
- **Step 2:** Use a recurrence relationship to fill in subproblem values.
- **Step 3:** Select the iteration order over the subproblems.
- **Step 4:** Initialize base case(s).
- **Step 5:** Return a solution.

Given our selection of subproblem structure for step 1, which of the following recurrences properly relates each subproblem to other subproblems?

- ○ (a) $Coin\_Sum[i] = \max(Coin\_Sum[i-1] + c_i, Coin\_Sum[i-2] + c_i)$
- ○ (b) $Coin\_Sum[i] = \max(Coin\_Sum[i-1], Coin\_Sum[i-2] + c_i)$
- ○ (c) $Coin\_Sum[i] = \max(Coin\_Sum[i-1], Coin\_Sum[i-2])$
- ○ (d) $Coin\_Sum[i] = \max(Coin\_Sum[i-1] + c_i, Coin\_Sum[i-2])$

Problem Statement + List of Steps

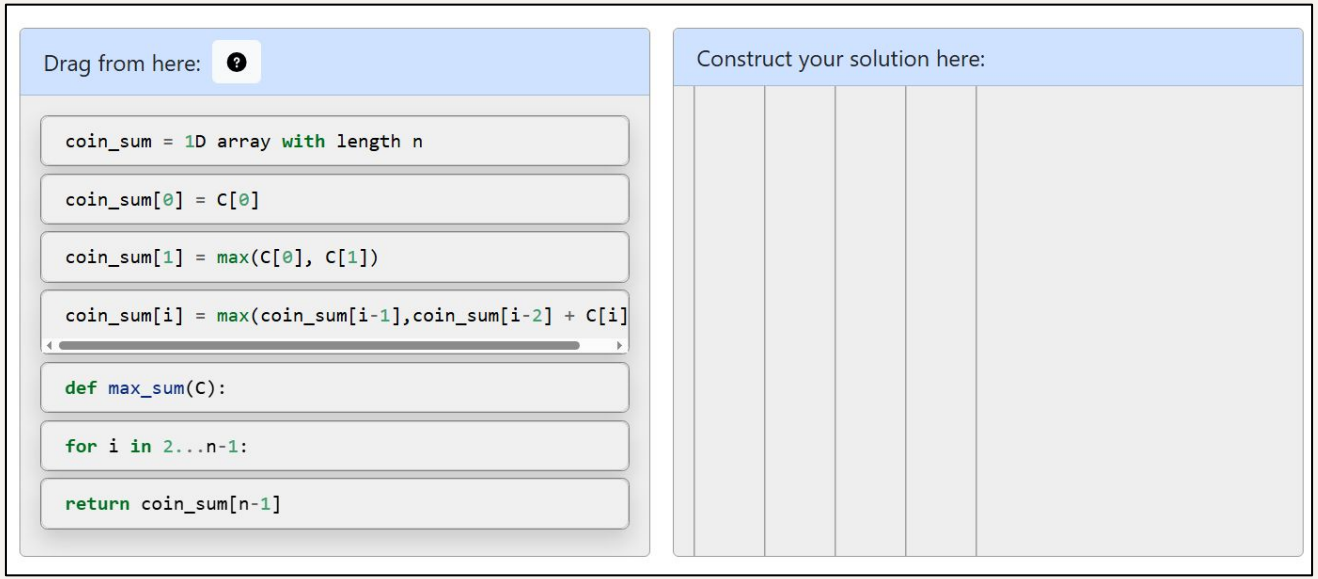
MCQ for each step

Implementation Box

Detailed Solution

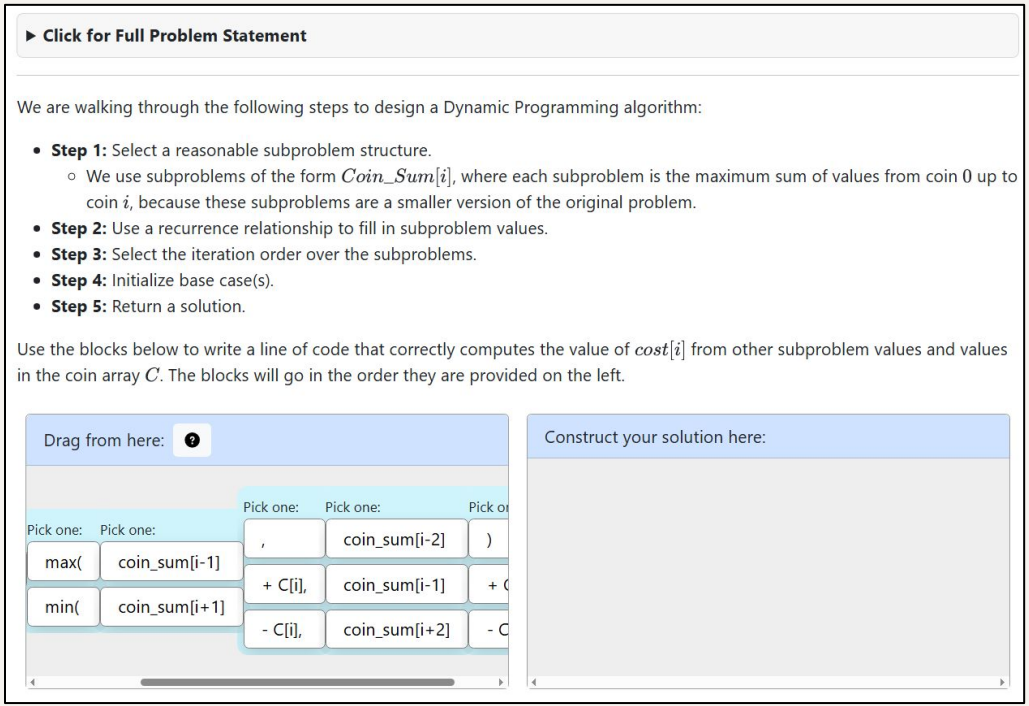
Condition 3: Design + Implementation

Parsons Problems (PP) are a problem style where students must place code blocks in the correct order.



In many CS contexts, they have shown to be more engaging, equally effective for learning, and more efficient than FRQs. [Ericson et al. ‘22, Wu et al. ‘23/’24, Poulsen et al. ‘23]

Condition 3: Design + Implementation



Problem Statement + List of Steps

MCQ for each step

Parson for each step

Implementation PP

Detailed Solution

Condition 3: Faded Scaffolding

Question 1	Question 2	Question 3
Problem Statement + List of Steps	Problem Statement + List of Steps	Problem Statement + List of Steps
MCQ for each step	MCQ for each step	MCQ for each step
PP for each step	PP for each step	PP for each step
Implementation PP	Implementation PP	Implementation **Box**
Detailed Solution	Detailed Solution	Detailed Solution

Intervention Structure

1. **Practice Problems:** 3 DP Problems, solutions provided
   - **Control**: Steps provided, no scaffolding
   - **Design**: Design steps are scaffolded with multiple-choice questions
   - **Design+Implementation (D+I)**: Design and Implementation are both scaffolded using Parsons Problems, with faded scaffolding
2. **Knowledge Check:**
   - **Q1 – Steps Recall:** “List the steps to solve a DP problem.”
   - **Q2 – DP Problem with Partial Scaffolding:** Multiple-choice question was used to scaffold the *Subproblem* and *Recurrence* step, then students were asked to implement as usual.
   - **Q3 – DP Problem with No Scaffolding:** Regular DP problem.

Data Analysis: Performance (RQ2)

Rubrics were designed with two major goals:

1. Evaluate *design* and *implementation* independently, to best measure the results of the intervention.
2. Mimic a reasonable algorithms course grading rubric.

Design (10 pts)

Subproblem: 1 pts
Recurrence: 3 pts
Iteration Order: 2 pts
Base Case: 2 pts
Return Value: 2 pts

+

Implementation (5 pts)

Correct: 5 pts
Minor Issue: 4 pts
Recursive: 3 pts
Mixed: 2 pts
None/Other: 0 pts

=

15 possible points per problem

Data Analysis: RQ1 and RQ3

RQ1: Steps Recall

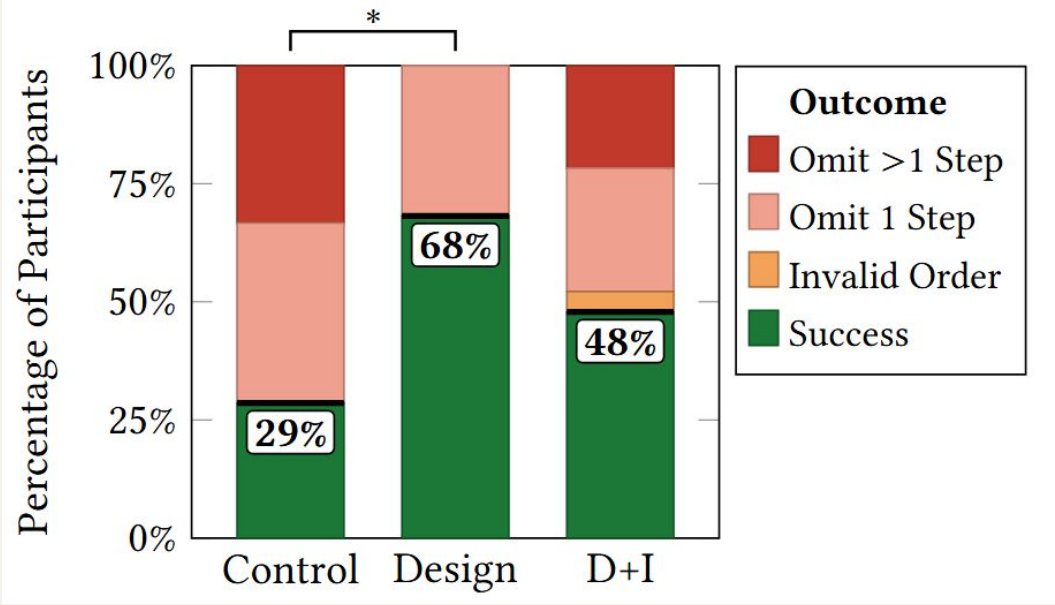
- Responses were coded for added or removed steps
- If response differed from ours, also coded for validity

RQ1 and RQ2 were coded by two authors, each rubric reaching substantial IRR (Cohen's kappa between 0.761-0.954)

RQ3: Pace

- Time spent on practice problems was tracked

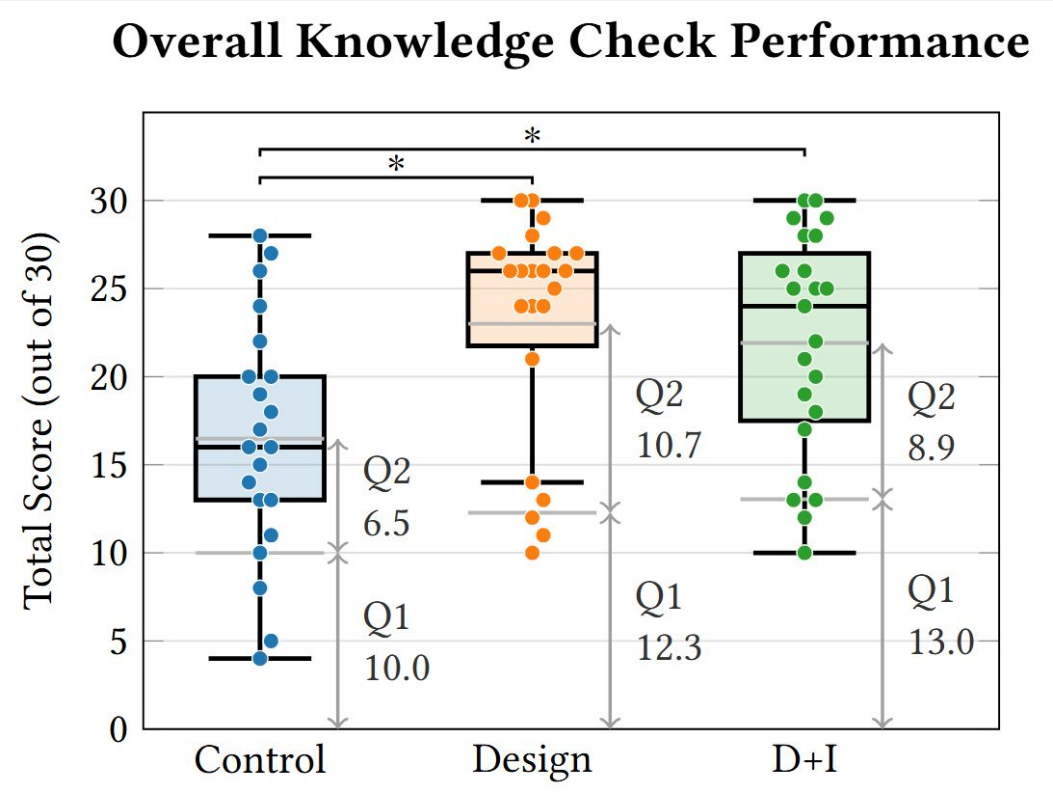
Results: Steps Recall (RQ1)



Students were asked to name the steps.

Takeaway: Students in scaffolded conditions were more likely to name a valid list of steps, but the only significant difference was between the Design and Control conditions.

Results: Performance (RQ2)

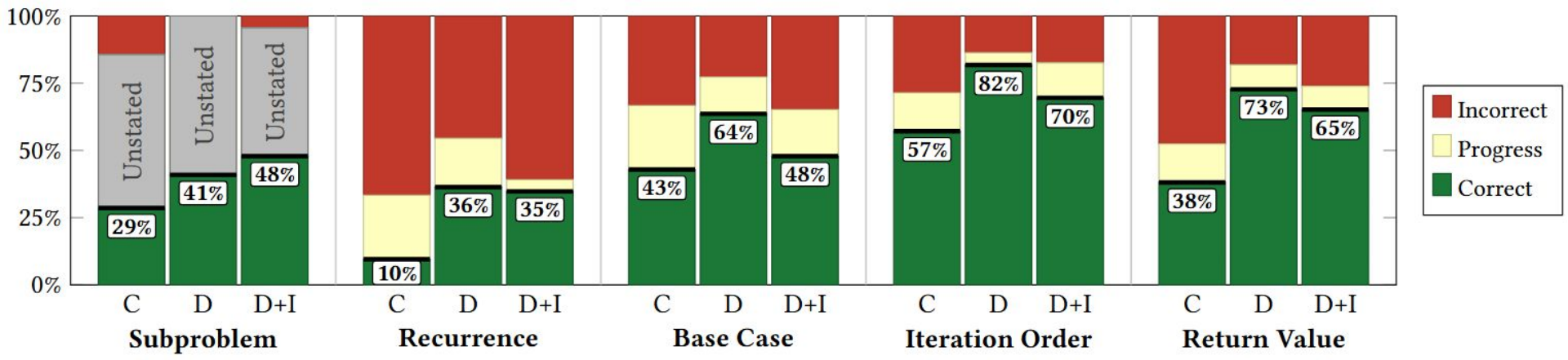


Sum of participant scores on the two knowledge check problems, split by condition.

Takeaway: Participants in scaffolded conditions scored significantly higher than control participants.

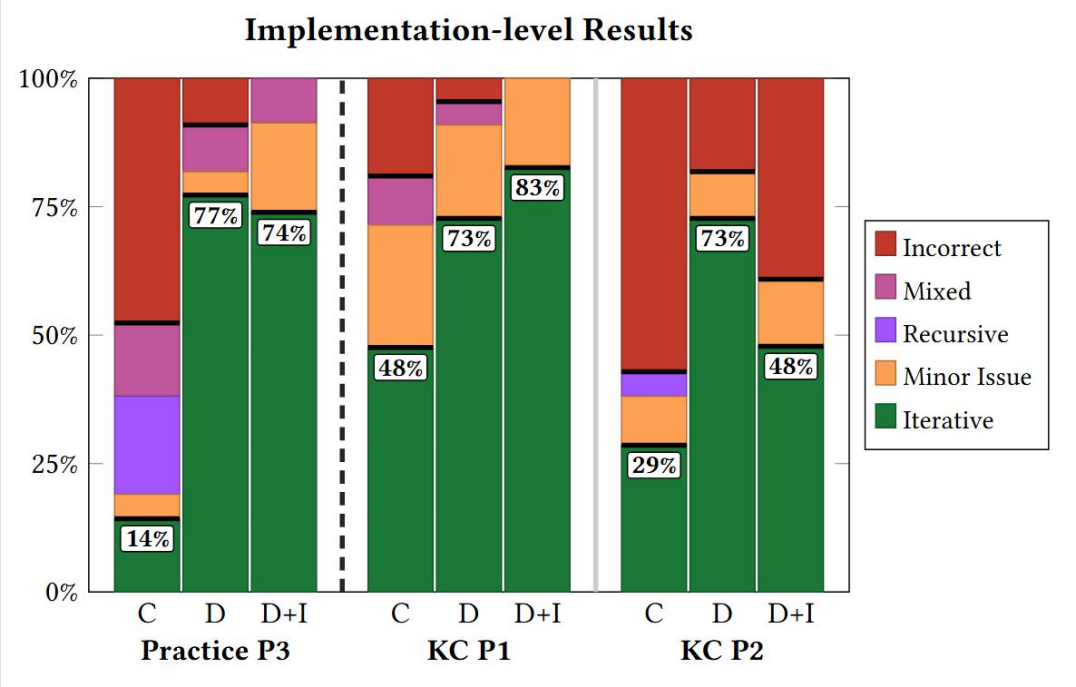
Results: Performance (RQ2) – Design

Design scores from KC P2, split by condition and by step.



Takeaway: Participants in scaffolded conditions consistently perform better than unscaffolded, but the ordering between scaffolded conditions is inconsistent.

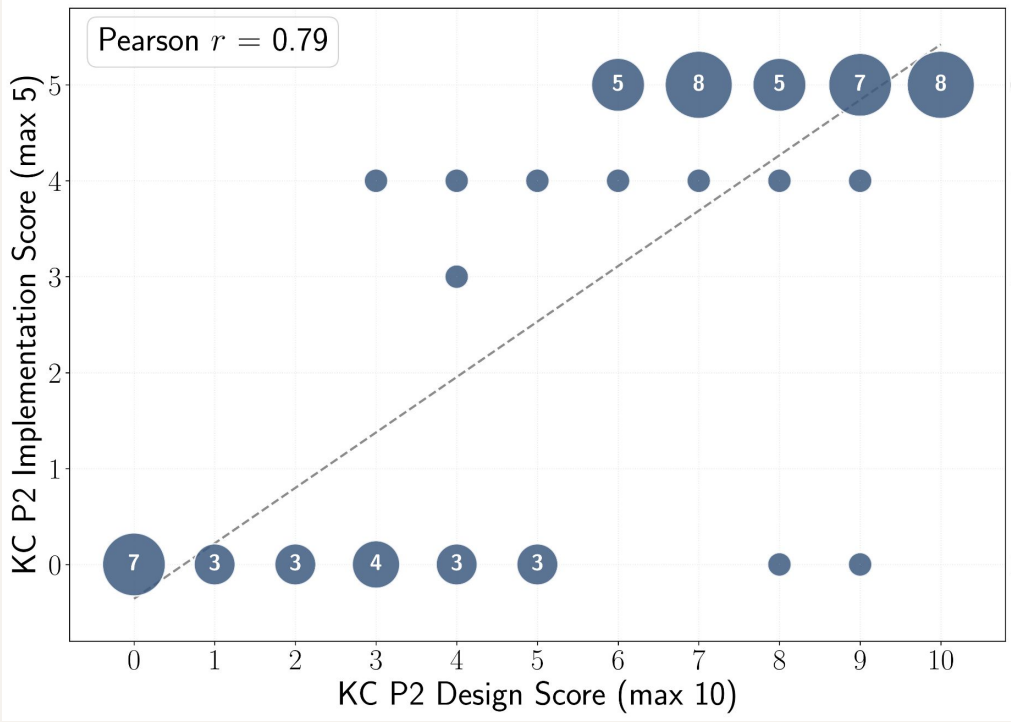
Results: Performance (RQ2) – Implementation



Implementation quality per problem, by condition.

Takeaway: Participants in scaffolded conditions consistently perform far better, and do not make recursive implementations. Again, results are inconsistent between scaffolded conditions.

Results: Performance (RQ2) – D vs. I

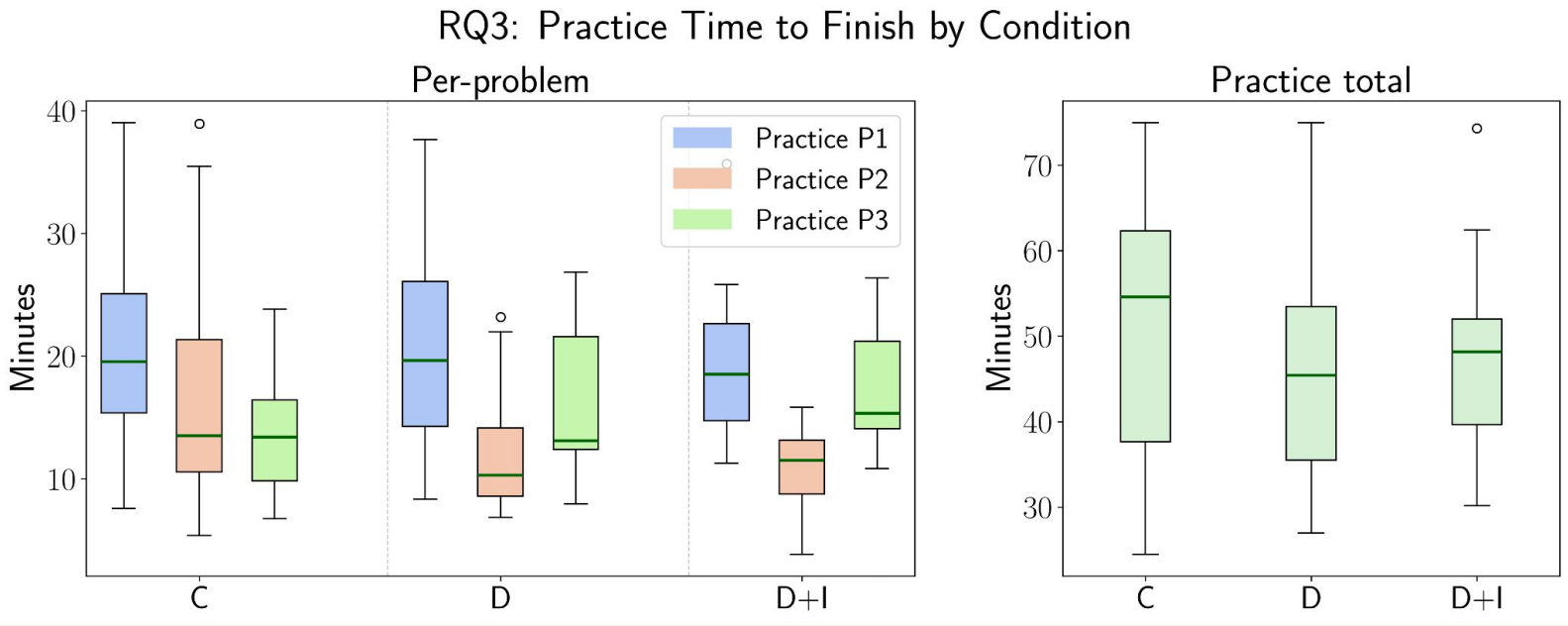


Relationship between KC P2 Design/Implementation scores for all participants.

Takeaway: Performance on these dimensions is closely correlated. Explains, at least partially, why implementation-level scaffolding showed no benefit.

Results: Practice Pace (RQ3)

Average time participants spent on each practice problem, split by problem.



Takeaway: No significant differences between conditions.

Results Overview

Design-level Scaffolding

We find clear evidence that design-level scaffolding is helpful.

- Improved ability to recall process steps
- Improved performance on both design and implementation aspects in knowledge check

Implementation-level Scaffolding

We find **no** evidence that implementation-level scaffolding provides any additional benefit.

- No significant differences in any metric
- Design and Implementation results are also closely correlated

Limitations and Conclusions

Limitations

Free-response questions are imperfect measurements of step-wise ability, though a more precise instrument would introduce unwanted scaffolding.

Steps recall is only a short-term measurement of strategy adoption. Future work should investigate longer-term strategy use.

Conclusions

Practice with integrated scaffolding of the steps led to improved recall and performance, compared with only providing the steps and worked solutions.

Implementation mistakes occur, but scores demonstrate that it is tied to design success, so **implementation scaffolding may be unnecessary**.

Concluding Takeaway

The development of algorithms pedagogy that explicitly teaches metacognitive skills can be more effective and foster learning that is more applicable beyond the classroom.

Thank you!

Literature

The existing research on algorithm design strategies hint at a gap in metacognitive skills.

Baseline

Students are adopting metacognitive strategies, but face barriers to productive use.

Potential

Metacognitive strategy development is closely tied to success in algorithm design.

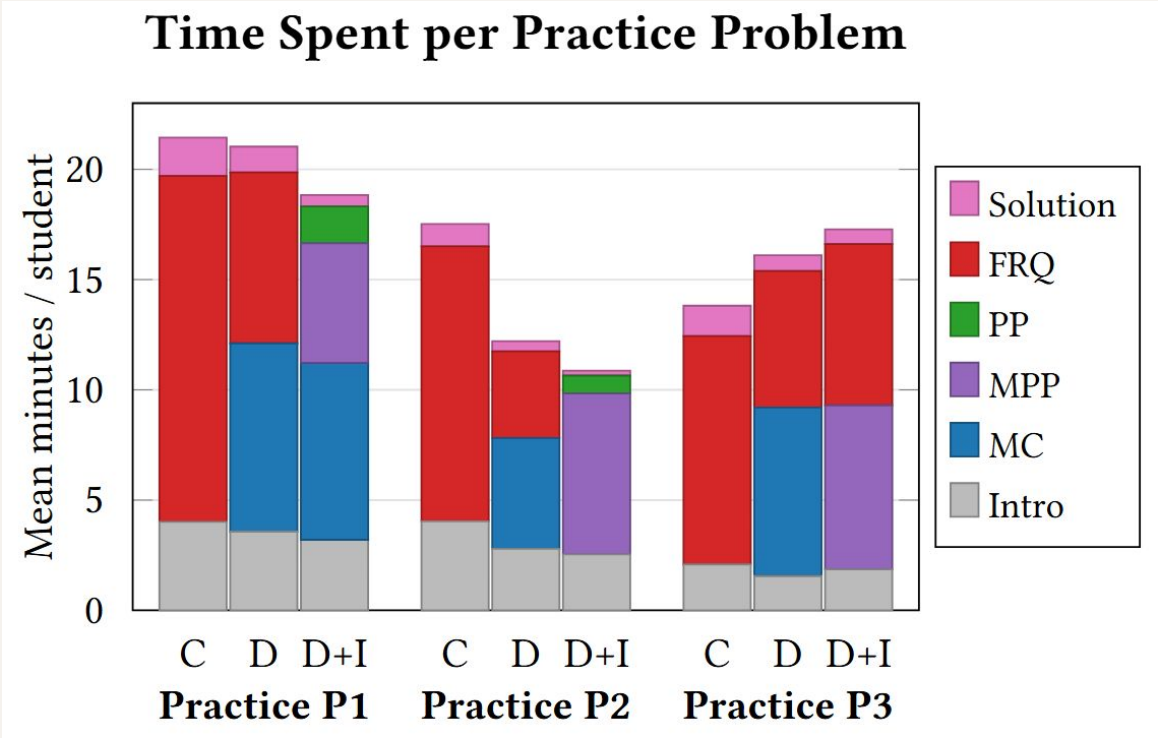
Intervention

Metacognitive instruction with strategy-tailored scaffolding induces better adoption.

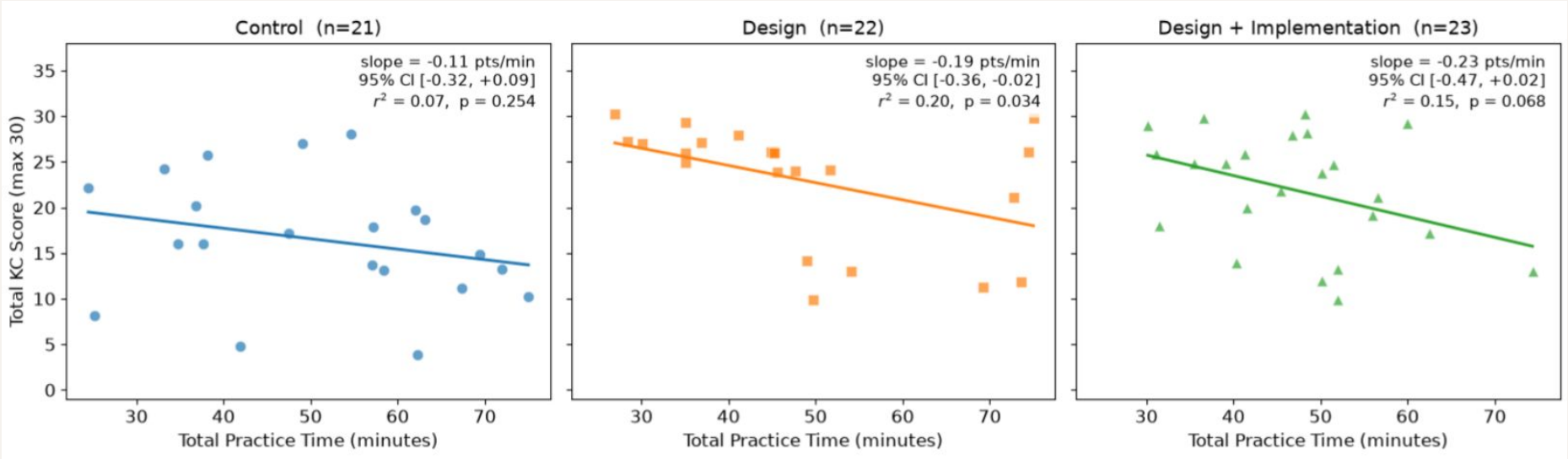
Thanks!

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, infographics & images by **Freepik**

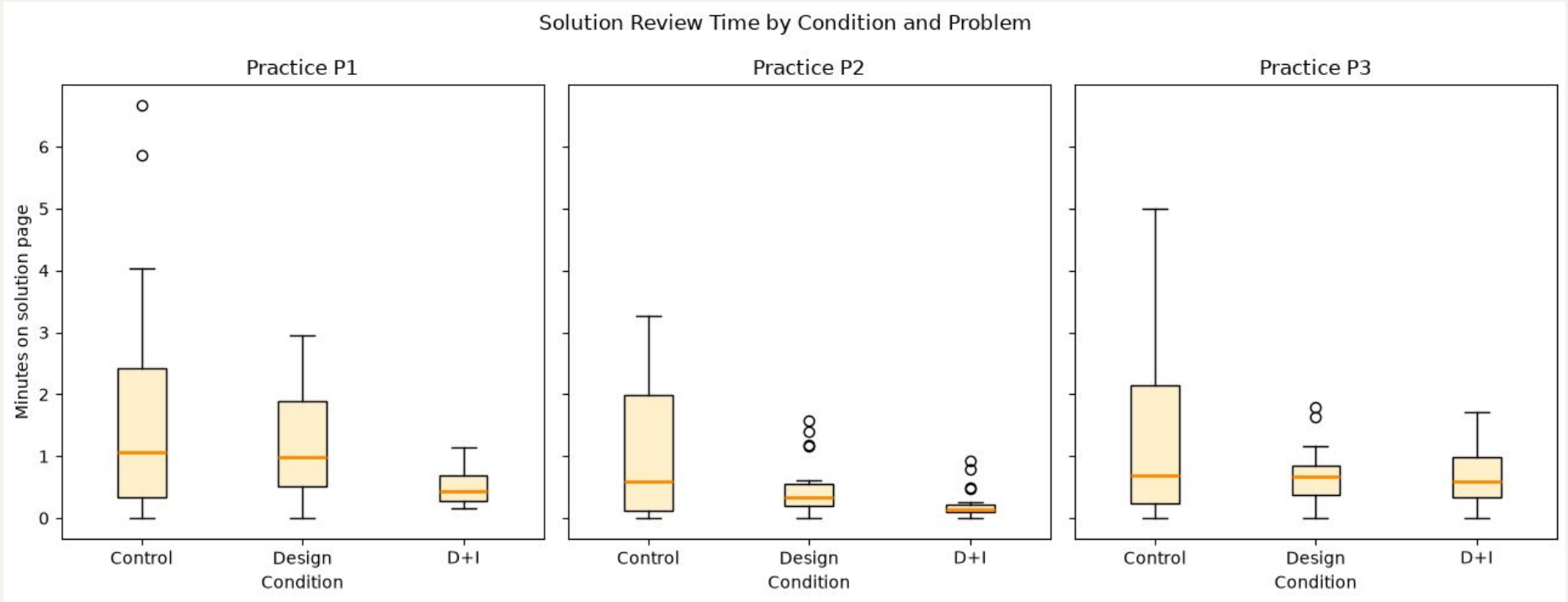
Auxiliary Results: Practice Pace (RQ3)



Auxiliary Results: Pace vs. Performance



Auxiliary Results: Solution Review Time



Auxiliary Results: Solution Review Time (Control)