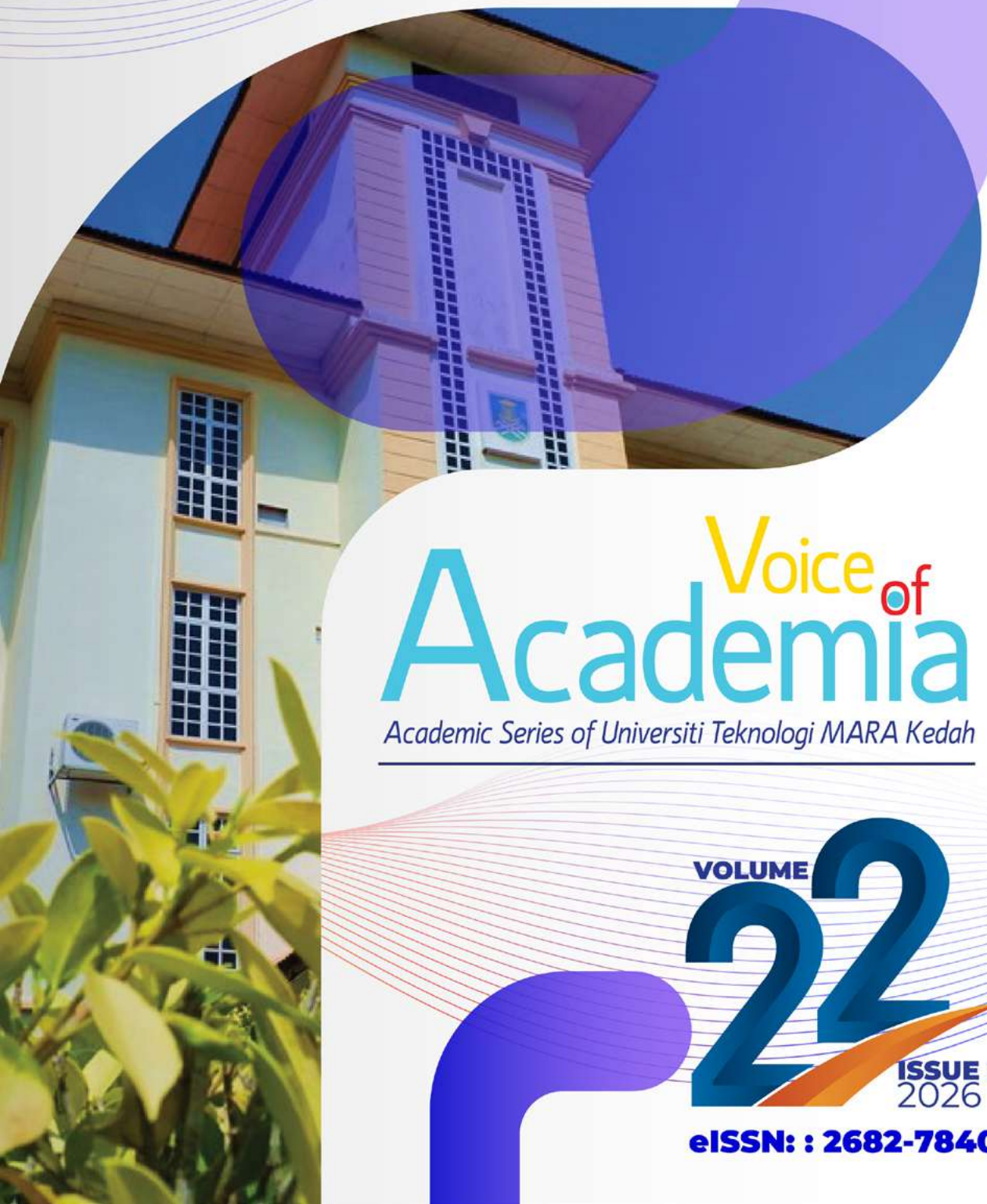

Voice of Academia

Academic Series of Universiti Teknologi MARA Kedah

VOLUME 22
ISSUE 2
2026
eISSN: : 2682-7840

e-ISSN:2682-7840

© Voice of Academia is jointly published by the Universiti Teknologi MARA Caawangan Kedah, Malaysia and Penerbit UiTM (UiTM Press), Universiti Teknologi MARA Malaysia, Shah Alam, Selangor.

TABLE of CONTENTS

Voice of Academia
e-ISSN: 2682-7840

Available online at
https://voa.uitm.edu.my/v1/
https://journal.uitm.edu.my/index.php/VOA/issue/archive

Voice of Academia 22(2) 2026, Bujang Khedif et al.

Voice of Academia

Algorithm Design Patterns: A Conceptual Framework for Simplifying Complex Problems in Beginner Computer Science Education

Lenny Yusrina Bujang Khedif [1], Sulastri Putit [2]

[1,2] *Faculty of Computer and Mathematical Sciences, Universiti Teknologi MARA Cawangan Sarawak, Kampus Samarahan 2*

ARTICLE INFO

Article history:

Received Feb 2026
Accepted June 2026
Published July 2026

Keywords:

Algorithm Design Patterns
Algorithmic Thinking
Beginner Computer Science Education
Instructional Scaffolding

Corresponding Author:
lennykhedif@uitm.edu.my

DOI:
https://doi.org/10.24191/VoA.v22i2.13442

ABSTRACT

Teaching algorithmic problem-solving remains a major challenge in beginner computer science education because novice programmers often struggle to identify appropriate solution strategies beyond programming syntax. Existing instructional approaches generally present algorithm paradigms as isolated topics, making it difficult for learners to recognise problem characteristics and transfer knowledge across different computational tasks. This study proposes the Algorithm Design Patterns (ADP) framework, a conceptual instructional model that organises five fundamental algorithm paradigms into reusable learning patterns for beginner computer science education. A conceptual research approach was adopted by synthesising literature on algorithmic thinking, programming pedagogy, instructional scaffolding, and algorithm design. The proposed framework integrates Divide and Conquer, Greedy Algorithms, Dynamic Programming, Backtracking, and Recursive Patterns into a progressive instructional structure supported by representative algorithms, classroom activities, and common learning pitfalls. By emphasising problem analysis before algorithm selection, the framework aims to strengthen algorithmic thinking and support more systematic learning. The proposed framework offers a structured pedagogical reference for introductory algorithm instruction and provides a foundation for future empirical validation in computer science education.

1. Introduction

Programming is widely recognised as one of the most challenging disciplines for novice computer science students because it requires more than learning a programming language. Students are expected to analyse problems, identify appropriate solution strategies, apply logical reasoning, and translate abstract ideas into executable algorithms. Consequently, the ability to think algorithmically has become a fundamental learning outcome in introductory programming courses, where learners are expected to solve computational problems systematically before implementing program code. Recent studies have also

1* Corresponding author: lennykhedif@uitm.edu.my

emphasised that effective programming education should prioritise conceptual reasoning and problem-solving skills rather than focusing solely on programming syntax (Tikva & Tambouris, 2021; Liu et al., 2025).

Despite its recognised importance, developing algorithmic thinking remains difficult for many beginner programmers. Rather than analysing a problem before attempting a solution, novice learners often begin writing code immediately and rely on repeated trial-and-error to modify the program until it produces the expected output. Although this approach may occasionally lead to a working solution, it rarely develops a deeper understanding of problem structure or improves learners' ability to solve unfamiliar computational problems. Previous studies have consistently reported that such practices contribute to misconceptions, low confidence, and weak transfer of algorithmic knowledge across different programming contexts (McCauley et al., 2015; Dogan, 2020; Song et al., 2021; Liu et al., 2025). These difficulties become increasingly apparent when students encounter algorithmic paradigms that require higher levels of abstraction, including Dynamic Programming, Backtracking, and Recursive algorithms.

Algorithmic thinking extends beyond understanding how individual algorithms operate. It involves recognising the characteristics of a computational problem, evaluating alternative solution strategies, and selecting an appropriate algorithmic approach before implementation begins. Learners who develop this capability are better prepared to transfer their knowledge to unfamiliar problems because they understand the underlying reasoning that connects different computational tasks. Research has shown that structured learning experiences and appropriate instructional support can significantly strengthen students' algorithmic thinking and overall problem-solving ability (Angeli, 2022; Yusuf & Noor, 2024).

In response to these challenges, programming education has increasingly shifted towards instructional approaches that emphasise conceptual understanding and guided learning. Rather than treating programming as the memorisation of syntax or isolated algorithms, contemporary pedagogical approaches encourage learners to develop coherent mental models that explain why particular solution strategies are appropriate for different classes of problems. Instructional scaffolding has been widely adopted to support this objective by helping students organise knowledge progressively while reducing unnecessary cognitive load during learning (Tikva & Tambouris, 2021; Duran et al., 2022; Bjursten et al., 2023; Chen et al., 2023).

One promising way to support this learning process is to organise algorithm paradigms as reusable instructional patterns rather than presenting them as isolated topics. The concept is inspired by design patterns, which describe recurring solutions to recurring problems within a specific context. Applied to programming education, such an approach enables students to focus first on identifying problem characteristics before considering implementation techniques. As learners repeatedly encounter similar problem structures, they gradually develop reusable cognitive models that support more effective algorithm selection and stronger problem-solving skills.

Although substantial research has examined algorithmic thinking, computational thinking, programming pedagogy, and instructional scaffolding, these perspectives are typically discussed independently. Existing instructional models often explain the operation of individual algorithms but provide limited guidance on how novice learners should determine which algorithmic paradigm is most appropriate for a particular problem. As a result, many students continue to perceive algorithm paradigms as unrelated concepts rather than complementary problem-solving strategies, making knowledge transfer unnecessarily difficult.

To address this limitation, this study proposes the Algorithm Design Patterns (ADP) framework, a conceptual instructional model that organises five fundamental algorithm paradigms—Divide and Conquer, Greedy Algorithms, Dynamic Programming, Backtracking, and Recursive Patterns—into a structured pedagogical framework for beginner computer science education. By integrating these paradigms into a single instructional model, the framework seeks to help learners analyse problem characteristics

1* Corresponding author: lennykhedif@uitm.edu.my

systematically, select appropriate solution strategies with greater confidence, and develop transferable algorithmic thinking skills applicable across diverse computational contexts.

Accordingly, the objective of this study is to develop a conceptual framework grounded in Algorithm Design Patterns that supports the teaching and learning of algorithmic thinking in introductory computer science education. The remainder of this paper is organised as follows. Section 2 reviews the relevant literature on algorithmic thinking, beginner programming education, design patterns, and instructional scaffolding. Section 3 describes the conceptual framework development process; Section 4 presents the proposed Algorithm Design Patterns framework; Section 5 discusses its educational implications, and Section 6 concludes the paper with recommendations for future research.

2. Literature Review

2.1 Algorithmic Thinking in Computer Science Education

Algorithmic thinking is widely regarded as a fundamental competency in computer science education because it enables learners to analyse computational problems systematically before developing solutions in code. Unlike programming syntax, which differs across programming languages, algorithmic thinking represents a transferable cognitive capability that supports abstraction, logical reasoning, decomposition, and solution planning regardless of the implementation environment. As a result, many introductory programming curricula recognise algorithmic thinking as a core learning outcome rather than a supplementary programming skill (Angeli & Giannakos, 2020; Liu et al., 2024).

The concept of algorithmic thinking has evolved considerably over the past two decades. Early work described it primarily as the ability to formulate precise sequences of instructions for solving computational problems through logical reasoning and abstraction (Futschek, 2006). More recent studies extend this perspective by emphasising decomposition, pattern recognition, abstraction, and algorithm construction as interconnected cognitive processes that underpin effective computational problem-solving (Wing, 2006; Grover & Pea, 2013). Rather than viewing algorithmic thinking as a procedural skill, contemporary research increasingly recognises it as a higher-order reasoning process that supports the transfer of knowledge across diverse programming tasks.

Empirical studies consistently demonstrate a positive relationship between algorithmic thinking and programming performance. Students who can analyse problem structures before implementation generally produce more efficient solutions and demonstrate greater confidence when solving unfamiliar programming tasks (Dogan, 2020; Angeli, 2022). Furthermore, recent investigations indicate that algorithmic thinking can be strengthened through carefully designed instructional interventions that provide structured guidance and progressively reduce learner dependency as competence develops (Yusuf & Noor, 2024). These findings suggest that algorithmic thinking should be intentionally cultivated rather than assumed to emerge naturally through programming practice alone.

Despite broad agreement regarding its importance, developing algorithmic thinking remains a persistent challenge in introductory programming education. Many students continue to focus on producing executable code without first examining the problem's underlying characteristics. As a result, learners often struggle to select suitable solution strategies when confronted with unfamiliar computational tasks. This limitation indicates that programming instruction should extend beyond explaining how algorithms operate to explicitly supporting learners in recognising when and why particular algorithmic approaches should be applied. Process-based analyses of novice programmers similarly demonstrate that successful problem-solving depends not only on coding ability but also on learners' reasoning and self-regulation during the problem-solving process (Song et al., 2021).

1* Corresponding author: lennykhedif@uitm.edu.my

2.2 Challenges in Beginner Programming Education

Teaching programming to novice learners continues to present considerable educational challenges because programming requires students to integrate logical reasoning, abstraction, problem analysis, and technical implementation simultaneously. Beginners must not only understand programming syntax but also construct mental models that explain why a particular solution strategy is appropriate for a given computational problem. Consequently, many students experience substantial cognitive overload during the early stages of programming education, particularly when confronted with problems that demand abstract reasoning and algorithm selection (Song et al., 2021; Hao et al., 2023).

One of the most frequently reported difficulties is learners' reliance on trial-and-error programming. Instead of analysing problem requirements before implementation, novice programmers often begin coding immediately and repeatedly modify their programs until acceptable outputs are obtained. Although this approach may produce functional solutions, it rarely encourages conceptual understanding or supports the development of transferable problem-solving skills (McCauley et al., 2015; Song et al., 2021). As a consequence, students frequently encounter similar difficulties when required to solve unfamiliar computational problems.

Another significant challenge concerns the selection of appropriate algorithmic strategies. Many students understand the syntax of a programming language but remain uncertain about whether a problem should be approached using Divide and Conquer, Greedy Algorithms, Dynamic Programming, Backtracking, or another algorithmic paradigm. This difficulty is often attributed to instructional practices that introduce algorithmic paradigms independently, providing limited opportunities for learners to compare alternative problem-solving strategies or recognise recurring problem characteristics (Liu et al., 2025).

Knowledge transfer represents an additional concern within beginner programming education. Rather than identifying common structural characteristics across computational problems, novice programmers frequently memorise algorithmic procedures for specific examples. As a result, previously acquired knowledge is not easily transferred to new programming contexts, limiting students' ability to solve unfamiliar problems independently (Song et al., 2021; Chen et al., 2023). These observations suggest that programming instruction should provide learners with instructional support that promotes conceptual understanding and encourages systematic reasoning before implementation begins.

2.3 Design Patterns in Computer Science Education

The concept of design patterns originated in architecture and was later adopted in software engineering to describe reusable solutions for recurring design problems. Rather than prescribing fixed implementations, design patterns provide general solution templates that can be adapted to different contexts while preserving fundamental design principles. Their widespread adoption in software engineering demonstrates the value of organising knowledge into reusable structures that support both learning and problem-solving (Fojtík & Jirásková, 2014).

Beyond software engineering, researchers have increasingly explored the educational potential of design patterns as pedagogical tools. In teaching and learning, design patterns help organise instructional knowledge into structured, reusable components, allowing learners to recognise similarities across different learning situations rather than treating concepts as isolated topics. This approach encourages learners to develop broader conceptual understanding while improving their ability to transfer knowledge across related domains.

The same principle can be applied to algorithm education. Fundamental algorithmic paradigms such as Divide and Conquer, Greedy Algorithms, Dynamic Programming, Backtracking, and Recursive Patterns recur across different categories of computational problems. Instead of teaching these paradigms

1* Corresponding author: lennykhedif@uitm.edu.my

independently, organising them as instructional patterns allows learners to associate specific problem characteristics with suitable solution strategies. Consequently, students are encouraged to focus on understanding the reasoning behind algorithm selection rather than memorising individual algorithms.

From a pedagogical perspective, this organisation supports the development of transferable knowledge. As learners repeatedly encounter similar problem structures, they gradually construct cognitive models that can be applied across multiple programming contexts. Such an approach aligns closely with the broader objective of algorithmic thinking, which emphasises recognising patterns and selecting appropriate strategies rather than recalling isolated procedures. Despite these educational advantages, the application of design patterns as a dedicated instructional framework for teaching algorithmic thinking remains relatively limited in current computer science education literature. This limitation further motivates the framework proposed in the present study.

2.4 Instructional Scaffolding

In education, pedagogy refers to the theories, principles, and instructional practices used to facilitate learning. In computer science education, pedagogical approaches influence how complex concepts such as algorithmic thinking are organised, explained, and scaffolded to novice learners. Consequently, selecting an appropriate pedagogical strategy is essential for reducing cognitive load while promoting meaningful learning.

Instructional scaffolding has become one of the most widely adopted pedagogical approaches for supporting learners in acquiring complex knowledge and problem-solving skills. Rooted in constructivist learning theory, scaffolding involves providing temporary instructional support that enables learners to perform tasks beyond their current level of competency before gradually reducing assistance as understanding develops. Within programming education, this approach is particularly valuable because beginners often struggle to manage multiple cognitive processes simultaneously, including problem analysis, algorithm selection, and program implementation (Duran et al., 2022; Shin et al., 2024).

Various forms of scaffolding have been applied successfully in programming education, including worked examples, guided exercises, visual representations, collaborative learning activities, and structured problem-solving tasks. These instructional strategies reduce unnecessary cognitive load while helping learners organise new knowledge into coherent conceptual structures (Hou et al., 2022; Pechorina et al., 2023). Rather than encouraging students to memorise programming syntax, scaffolding supports the gradual development of reasoning skills that can later be applied independently.

The effectiveness of instructional scaffolding depends not only on the amount of support provided but also on how well instructional guidance corresponds to learners' cognitive needs. Constructivist perspectives suggest that meaningful learning occurs when students actively relate new concepts to their existing knowledge and progressively refine their understanding through guided practice. Consequently, instructional activities should encourage learners to analyse problem characteristics before implementing algorithmic solutions, thereby strengthening both conceptual understanding and long-term knowledge transfer.

The proposed Algorithm Design Patterns (ADP) framework adopts these principles by integrating conceptual explanations, representative algorithms, classroom learning activities, and common misconceptions into a structured instructional model. Rather than requiring students to memorise isolated algorithms, the framework guides learners to recognise recurring problem characteristics before selecting appropriate algorithmic strategies. In doing so, it aims to reduce cognitive overload while promoting deeper algorithmic reasoning and more meaningful learning experiences in introductory programming education.

1* Corresponding author: lennykhedif@uitm.edu.my

2.5 Research Gap

Although algorithmic thinking has received considerable attention in computer science education, existing studies have primarily focused on improving programming performance, computational thinking, or evaluating specific instructional interventions. Compared with other areas, attention has been given to developing comprehensive pedagogical frameworks that explicitly organise algorithm paradigms into structured learning models for novice programmers. Recent reviews similarly indicate that greater emphasis should be placed on supporting learners' decision-making processes during algorithm selection rather than concentrating solely on algorithm implementation (Liu et al., 2025).

A second limitation concerns the way algorithm paradigms are typically presented in introductory programming courses. Divide and Conquer, Greedy Algorithms, Dynamic Programming, Backtracking, and Recursive Patterns are commonly taught as separate instructional units. While this approach introduces students to individual techniques, it provides limited opportunities for learners to identify relationships among different paradigms or recognise recurring characteristics shared by computational problems. Consequently, many novice programmers struggle to transfer knowledge from previously encountered problems to unfamiliar situations.

Furthermore, although the concept of design patterns has been widely adopted in software engineering and has demonstrated educational value in organising reusable knowledge structures, its application as a dedicated instructional framework for teaching algorithmic thinking remains relatively underexplored. Existing studies rarely integrate algorithm design, instructional scaffolding, and pedagogical design patterns within a unified conceptual model specifically intended for beginner programming education.

To address these limitations, this study proposes the Algorithm Design Patterns (ADP) framework, which organises five fundamental algorithm paradigms into reusable instructional patterns that support systematic problem analysis and algorithm selection. By integrating concepts from algorithmic thinking, design patterns, and instructional scaffolding, the framework seeks to provide educators with a coherent pedagogical model that encourages transferable algorithmic reasoning and supports more effective learning in introductory computer science courses.

Table 1. Summary of Research Gap

Existing Studies	Limitation	Contribution of This Study
Focus on programming performance	Limited emphasis on structured algorithmic reasoning	Introduces a conceptual framework based on algorithm design patterns
Algorithm paradigms taught independently	Lack of integration among paradigms	Organises five paradigms into a unified instructional framework
Design patterns mainly used in software engineering	Limited application in algorithm instruction	Applies design patterns as pedagogical support for beginner programming
Existing instructional approaches focus on implementation	Limited guidance for strategy selection	Supports learners in recognising problem characteristics before implementation

The literature indicates that algorithmic thinking plays a central role in beginner programming education, yet novice programmers continue to struggle to select appropriate problem-solving strategies. Existing studies have explored algorithmic thinking, programming pedagogy, instructional scaffolding, and design patterns independently, but limited research has integrated these perspectives into a unified conceptual

1* Corresponding author: lennykhedif@uitm.edu.my

framework for algorithm instruction. This gap provides the foundation for the proposed Algorithm Design Patterns framework, which is described in the following section.

3. Methodology

3.1 Framework Development Process

This study adopted a conceptual research approach to develop the proposed Algorithm Design Patterns framework. Rather than conducting an empirical investigation, the study synthesised existing literature to establish a structured pedagogical framework that supports algorithmic thinking in beginner computer science education.

The framework was developed through four systematic phases, as illustrated in Figure 1.

Phase 1: Literature Review

The first phase involved reviewing relevant literature related to algorithmic thinking, beginner programming education, instructional scaffolding, computational thinking, and algorithm design paradigms. Journal articles, conference papers, books, and review studies published between 2020 and 2025 were prioritised to ensure that the framework reflected current educational practices while retaining seminal references that remain fundamental in computer science education.

The literature review served two primary purposes. First, it identified the challenges novice programmers face in developing algorithmic thinking. Second, it examined existing instructional approaches used to teach algorithm design and problem-solving in introductory programming courses.

Phase 2: Identification of Algorithm Design Paradigms

Based on the literature review, five algorithm paradigms that are consistently introduced in beginner programming and introductory algorithm courses were identified.

These paradigms include:

- Divide and Conquer
- Greedy Algorithms
- Dynamic Programming
- Backtracking
- Recursive Patterns

The selection was based on their educational significance, widespread use in undergraduate computer science curricula, and their ability to represent different computational problem-solving strategies. Together, these paradigms provide a progressive learning pathway that exposes students to increasingly sophisticated forms of algorithmic reasoning. Their curricular relevance and representative problem-solving roles are supported by established algorithm literature and recent reviews of algorithm-design education (Cormen et al., 2009; Liu et al., 2025).

The progression begins with Divide and Conquer, which introduces decomposition through recursive reasoning. Greedy Algorithms subsequently strengthen local optimisation strategies before learners encounter Dynamic Programming, where overlapping subproblems and optimal substructure require more advanced reasoning. Backtracking extends learners' understanding through systematic search under constraints, while Recursive Patterns consolidate recursive abstraction across multiple problem domains. This sequence is intended to support gradual cognitive development rather than prescribe a rigid teaching order.

1* Corresponding author: lennykhedif@uitm.edu.my

Phase 3: Pedagogical Analysis

Each algorithm paradigm was subsequently analysed from an educational perspective to determine how it could be presented more effectively to novice programmers.

The analysis focused on four instructional components:

- conceptual understanding;
- representative algorithm examples;
- classroom learning activities; and
- common misconceptions experienced by beginner programmers.

Rather than emphasising algorithm efficiency or mathematical analysis, the pedagogical analysis focused on how each paradigm could facilitate the development of algorithmic thinking in introductory programming instruction.

Phase 4: Framework Development

The findings obtained from the previous phases were synthesised into the proposed Algorithm Design Patterns framework.

The framework was organised into a structured instructional model comprising five algorithm paradigms that are intended to assist educators in presenting algorithm concepts systematically while supporting learners in recognising recurring problem characteristics before implementing program solutions.

Unlike conventional instruction, where algorithm paradigms are often introduced independently, the proposed framework provides a conceptual organisation that highlights the relationships among different problem-solving strategies and encourages transferable algorithmic thinking.

3.2 Proposed Methodology Framework

Figure 1 illustrates the overall process for developing the proposed conceptual framework. The framework development began with a comprehensive review of relevant literature, followed by the identification of fundamental algorithm paradigms commonly taught in introductory programming courses. Each paradigm was then analysed from a pedagogical perspective before it was integrated into a unified conceptual framework that supports algorithmic thinking among beginner programmers.

1* Corresponding author: lennykhedif@uitm.edu.my

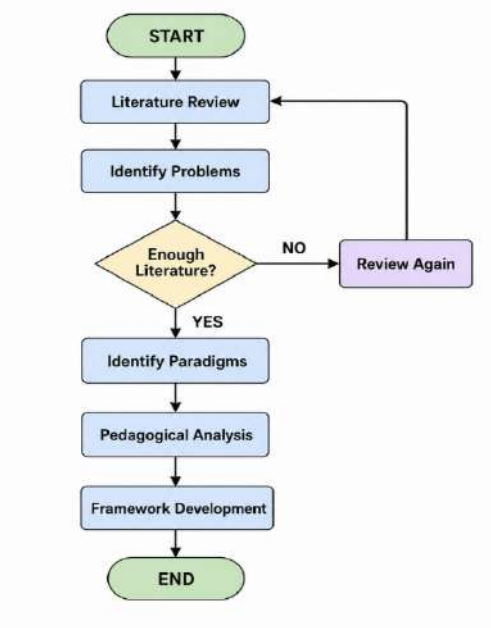

Figure 1. Framework Development Process

The methodology adopted in this study emphasises systematic knowledge synthesis rather than experimental validation. As a result, the proposed framework should be viewed as a conceptual instructional model that provides a foundation for future classroom implementation and empirical evaluation in beginner computer science education.

4. Proposed Algorithm Design Patterns Framework

4.1 Framework Overview

The proposed Algorithm Design Patterns (ADP) framework was developed to support the teaching and learning of algorithmic thinking in beginner computer science education. The framework organises fundamental algorithmic paradigms into structured instructional patterns. These patterns assist novice programmers in analysing computational problems and selecting appropriate solution strategies before implementing code.

Unlike conventional approaches that present algorithm paradigms as independent topics, the proposed framework emphasises the relationships among different problem-solving strategies. By recognising recurring problem characteristics, learners are encouraged to develop transferable algorithmic thinking rather than memorising individual algorithms. This approach is intended to help beginner programmers understand when and why a particular algorithmic strategy should be applied.

The framework consists of five fundamental algorithm paradigms commonly introduced in introductory programming and algorithm courses: Divide and Conquer, Greedy Algorithms, Dynamic Programming, Backtracking, and Recursive Patterns. These paradigms were selected because they represent distinct approaches to computational problem-solving and collectively cover a broad range of algorithmic concepts introduced at the undergraduate level. These paradigms are well established in algorithm curricula and collectively illustrate decomposition, local choice, reuse of subproblem solutions, systematic search, and recursive abstraction (Cormen et al., 2009; Liu et al., 2025).
Figure 2 presents the overall structure of the proposed Algorithm Design Patterns framework.

1* Corresponding author: lennykhedif@uitm.edu.my

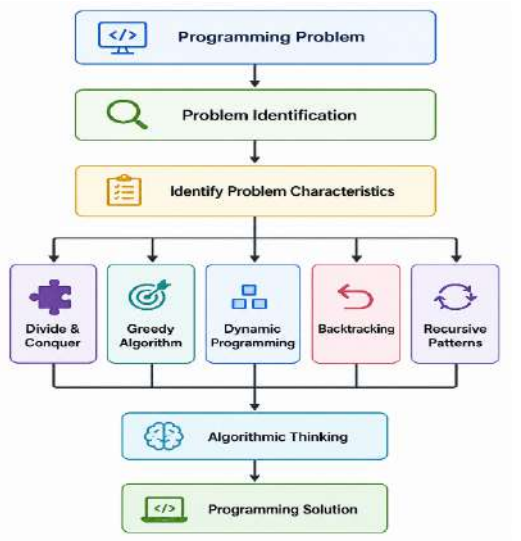

Figure 2. Proposed Algorithm Design Patterns Framework

The framework begins with problem identification, in which learners analyse the characteristics of a programming problem before selecting an appropriate algorithmic strategy. After identifying the problem characteristics, learners determine which algorithm paradigm best matches the problem. Finally, the selected strategy guides algorithm development before program implementation.

4.2 Algorithm Design Patterns

The proposed framework consists of five algorithm design patterns that collectively represent different approaches to computational problem solving. Representative algorithms accompany each pattern, along with suggested classroom activities and common misconceptions frequently encountered by beginner programmers. These instructional elements are intended to assist educators in designing learning activities and, at the same time, help students recognise recurring problem characteristics.

Divide and Conquer

Divide and Conquer introduces learners to the concept of decomposing a complex problem into smaller independent subproblems that can be solved separately before combining their solutions. This paradigm strengthens learners' understanding of recursive decomposition and systematic problem analysis. Representative algorithms include Merge Sort, Quick Sort, and Binary Search. Classroom activities may involve tracing recursion trees and analysing recursive execution, while common misconceptions include incorrect base-case identification and misunderstanding recursive decomposition.

Greedy Algorithms

Greedy Algorithms solve optimisation problems by selecting the locally optimal choice at each decision stage. This paradigm encourages learners to evaluate alternative solutions in the context of immediate decision-making, while understanding that local optimisation does not always guarantee a globally optimal solution. Typical examples include Activity Selection and Huffman Coding. Classroom activities may involve scheduling simulations and algorithm comparisons, whereas common misconceptions arise when students incorrectly assume that every optimisation problem can be solved using a greedy approach. Active-learning research on greedy algorithms also shows the importance of requiring students to justify why a greedy choice is valid rather than applying the strategy mechanically (Velázquez-Iturbide, 2013).

1* Corresponding author: lennykhedif@uitm.edu.my

Dynamic Programming

Dynamic Programming focuses on solving optimisation problems by storing previously computed results to avoid recomputation. Students are introduced to concepts such as overlapping subproblems and optimal substructure through representative algorithms, including Fibonacci, Knapsack, and Longest Common Subsequence. Suggested classroom activities include constructing dynamic programming tables and comparing recursive and dynamic programming solutions. Common misconceptions include confusing memorisation with tabulation and applying dynamic programming to problems that lack overlapping subproblems. The need to identify overlapping subproblems and optimal substructure makes this paradigm particularly demanding for novice learners (Liu et al., 2025).

Backtracking

Backtracking introduces systematic search techniques that explore multiple solution paths while eliminating invalid alternatives. Representative problems include the N-Queens problem, Sudoku, and maze-solving algorithms. Classroom activities may involve drawing state-space trees and tracing search paths. Beginner programmers commonly struggle to distinguish between backtracking and brute-force search and often neglect pruning strategies that improve search efficiency. Its pedagogical value lies in making decision points, constraints, and the consequences of reversing choices visible to learners (Fojtík & Jirásková, 2014).

Recursive Patterns

Recursive Patterns strengthen learners' understanding of recursive reasoning by solving problems through smaller instances of the same problem. Representative algorithms include Factorial, Tower of Hanoi, and Tree Traversal. Classroom activities focus on tracing recursive function calls and analysing recursion trees. Common misconceptions include incorrect base-case definitions, infinite recursion, and misunderstanding recursive return values. These difficulties are consistent with research showing that learners commonly struggle to form accurate mental models of recursive execution and call-stack behaviour (McCauley et al., 2015).

Table 2. Summary of the Proposed Algorithm Design Patterns

Algorithm Design Pattern	Representative Algorithms	Suggested Classroom Activity (CSC126)	Common Pitfall
Divide and Conquer	• Merge Sort • Quick Sort • Binary Search	Trace recursion tree	Incorrect base case and recursive decomposition
Greedy Algorithms	• Activity Selection • Huffman Coding	Scheduling and optimisation exercises	Assuming greedy solutions are always optimal
Dynamic Programming	• Fibonacci • Knapsack • Longest Common Subsequence	Construct dynamic programming tables	Confusing memoisation and tabulation
Backtracking	• N-Queens • Sudoku • Maze Solving	Draw state-space tree	Treating backtracking as brute-force search
Recursive Patterns	• Factorial • Tower of Hanoi • Tree Traversal	Trace recursive function calls	Infinite recursion and incorrect return values

1* Corresponding author: lennykhedif@uitm.edu.my

4.3 Framework Summary

The proposed Algorithm Design Patterns framework provides a structured approach for introducing algorithm paradigms in beginner computer science education. Instead of presenting each paradigm independently, the framework organises them into a coherent instructional model that assists learners in recognising problem characteristics before selecting appropriate solution strategies. This organisation supports the development of algorithmic thinking by encouraging students to analyse problems systematically rather than relying solely on programming syntax or memorised algorithms.

The inclusion of representative examples, classroom activities, and common misconceptions further assists educators in planning instructional activities while helping novice programmers develop a deeper conceptual understanding of algorithm design. Consequently, the framework serves as a practical reference for teaching introductory algorithm concepts. It provides a foundation for future implementation in beginner programming courses such as CSC126: Fundamentals of Algorithm and Computer Problem Solving.

5. Discussion

The proposed Algorithm Design Patterns (ADP) framework provides a structured instructional approach for teaching algorithmic thinking in beginner computer science education. Unlike conventional instruction, which introduces algorithm paradigms independently, the framework organises five fundamental paradigms into a coherent conceptual model that emphasises problem analysis before algorithm implementation. By encouraging learners to identify problem characteristics before selecting an appropriate solution strategy, the framework promotes transferable algorithmic thinking and reduces reliance on trial-and-error programming. This emphasis is consistent with recent studies highlighting the importance of structured algorithm-design reasoning and instructional support in programming education (Liu et al., 2025; Song et al., 2021; Duran et al., 2022).

The framework also provides practical guidance for educators who teach introductory programming and algorithm courses. The inclusion of representative algorithms, suggested classroom activities, and common misconceptions offers a structured reference for lesson planning and classroom implementation. Rather than preparing instructional materials independently for each algorithm paradigm, educators may use the proposed framework as a guide for organising learning activities that progressively develop students' algorithmic reasoning. The inclusion of classroom activities aligned with CSC126: Fundamentals of Algorithm and Computer Problem Solving further demonstrates the framework's potential applicability within undergraduate computer science curricula. Evidence from scaffolded and collaborative programming activities indicates that carefully designed supports can strengthen computational thinking, self-efficacy, knowledge transfer, and motivation (Wei et al., 2021; Angeli, 2022; Shin et al., 2024).

Another important contribution of the framework is its ability to connect different algorithm paradigms within a single instructional structure. Conventional programming instruction often presents Divide and Conquer, Greedy Algorithms, Dynamic Programming, Backtracking, and Recursive Patterns as separate topics. Consequently, students often perceive each paradigm as unrelated knowledge rather than recognising their shared purpose as problem-solving strategies. The proposed framework addresses this limitation by encouraging learners to identify common problem characteristics before selecting suitable algorithmic approaches, thereby supporting the development of transferable algorithmic thinking. The use of recurring pedagogical structures also reflects broader recommendations for coherent teaching strategies in programming education (Bjursten et al., 2023; Liu et al., 2025).

Although the proposed framework provides a structured conceptual model for teaching algorithmic thinking, this study is limited to conceptual framework development and does not include classroom implementation or empirical evaluation. Therefore, the framework's educational effectiveness has not yet

1* Corresponding author: lennykhedif@uitm.edu.my

been measured. Future studies should investigate its implementation in introductory programming courses to examine its impact on students' algorithmic thinking, programming performance, and learning experiences. Such investigations provide empirical evidence of the framework's usefulness across different educational contexts. Overall, the proposed Algorithm Design Patterns framework complements rather than replaces existing programming pedagogy by providing educators with a structured reference for teaching algorithmic thinking through recurring algorithm-design strategies.

6. Conclusion

Algorithmic thinking is a fundamental competency in beginner computer science education because it enables learners to analyse computational problems systematically before developing program solutions. However, novice programmers often struggle to select appropriate algorithmic strategies when algorithm paradigms are introduced as isolated topics.

This study proposed the Algorithm Design Patterns (ADP) framework, a conceptual instructional model that organises five fundamental algorithm paradigms into reusable learning patterns for beginner computer science education. Developed through a conceptual synthesis of the literature, the framework integrates algorithmic thinking, instructional scaffolding, and design-pattern principles to support systematic problem analysis and algorithm selection.

Rather than focusing solely on algorithm implementation, the framework encourages learners to recognise problem characteristics before choosing an appropriate solution strategy. The inclusion of representative algorithms, classroom activities, and common misconceptions provides educators with practical guidance for teaching introductory algorithms.

Although the framework has not yet been empirically validated, it provides a foundation for future classroom implementation and evaluation, particularly within introductory courses such as CSC126: Fundamentals of Algorithm and Computer Problem Solving. Overall, the proposed Algorithm Design Patterns framework offers a structured pedagogical reference that supports algorithmic thinking, systematic problem solving, and future empirical research in beginner computer science education.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments and valuable suggestions, which significantly improved the quality of this manuscript.

Funding Details

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Authors Contributions

All authors contributed to developing ideas and the submission process.

Conflict of Interest

There is no conflict of interest associated with this publication.

1* Corresponding author: lennykhedif@uitm.edu.my

References

Angeli, C. (2022). The effects of scaffolded programming scripts on pre-service teachers' computational thinking: Developing algorithmic thinking through programming robots. *International Journal of Child-Computer Interaction, 31,* 100329. https://doi.org/10.1016/j.ijcci.2021.100329

Angeli, C., & Giannakos, M. (2020). Computational thinking education: Issues and challenges. *Computers in Human Behavior, 105,* 106185. https://doi.org/10.1016/j.chb.2019.106185

Bjursten, E.-L., Nilsson, T., & Gumaelius, L. (2023). Computer programming in primary schools: Swedish technology teachers' pedagogical strategies. *International Journal of Technology and Design Education, 33*(4), 1345–1368. https://doi.org/10.1007/s10798-022-09786-7

Chen, H. E., Sun, D., Hsu, T.-C., Yang, Y., & Sun, J. (2023). Visualising trends in computational thinking research from 2012 to 2021: A bibliometric analysis. *Thinking Skills and Creativity, 47*, 101224. https://doi.org/10.1016/j.tsc.2022.101224

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

Dogan, A. (2020). Algorithmic thinking in primary education. *International Journal of Progressive Education, 16*(4), 286–301. https://doi.org/10.29329/ijpe.2020.268.18

Duran, R., Zavgorodniaia, A., & Sorva, J. (2022). Cognitive load theory in computing education research: A review. *ACM Transactions on Computing Education, 22*(4), Article 40, 1–27. https://doi.org/10.1145/3483843

Fojtík, R., & Jirásková, D. (2014). Design patterns in the teaching of programming. *Procedia - Social and Behavioral Sciences, 143,* 352–357. https://doi.org/10.1016/j.sbspro.2014.07.575

Futschek, G. (2006). Algorithmic thinking: The key for understanding computer science. In R. T. Mittermeir (Ed.), Informatics education - The bridge between using and understanding computers (pp. 159–168). Springer. https://doi.org/10.1007/11915355_15

Grover, S., & Pea, R. (2013). Computational thinking in K–12: A review of the state of the field. *Educational Researcher, 42*(1), 38–43. https://doi.org/10.3102/0013189X12463051

Hao, X., Xu, Z., Guo, M., Hu, Y., & Geng, F. (2023). The effect of embedded structures on cognitive load for novice learners during block-based code comprehension. *International Journal of STEM Education, 10*, Article 42. https://doi.org/10.1186/s40594-023-00432-9

Hou, X., Ericson, B. J., & Wang, X. (2022). Using adaptive Parsons problems to scaffold write-code problems. In Proceedings of the 2022 ACM Conference on International Computing Education Research (Vol. 1). Association for Computing Machinery. https://doi.org/10.1145/3501385.3543977

Liu, J., Poulsen, S., Goodwin, E., Chen, H., Williams, G., Gertner, Y., & Franklin, D. (2025). Teaching algorithm design: A literature review. *ACM Transactions on Computing Education, 25*(2), Article 17, 1–20. https://doi.org/10.1145/3727987

1* Corresponding author: lennykhedif@uitm.edu.my

Liu, Z., Gearty, Z., Richard, E., Orrill, C. H., Kayumova, S., & Balasubramanian, R. (2024). Bringing computational thinking into classrooms: A systematic review on supporting teachers in integrating computational thinking into K–12 classrooms. *International Journal of STEM Education, 11*, Article 51. https://doi.org/10.1186/s40594-024-00510-6

McCauley, R., Grissom, S., Fitzgerald, S., & Murphy, L. (2015). Teaching and learning recursive programming: A review of the research literature. *Computer Science Education, 25*(1), 37–66. https://doi.org/10.1080/08993408.2015.1033205

Pechorina, Y., Anderson, K., & Denny, P. (2023). Metacodenition: Scaffolding the problem-solving process for novice programmers. In Proceedings of the 25th Australasian Computing Education Conference (pp. 59–68). Association for Computing Machinery. https://doi.org/10.1145/3576123.3576130

Shin, Y., Jung, J., & Lee, H. J. (2024). Exploring the impact of concept-oriented faded worked-out examples and metacognitive scaffolding on learners' transfer performance and motivation in programming education. *Metacognition and Learning, 19*(1), 147–168. https://doi.org/10.1007/s11409-023-09362-x

Song, D., Hong, H., & Oh, E. Y. (2021). Applying computational analysis of novice learners' computer programming patterns to reveal self-regulated learning, computational thinking, and learning performance. *Computers in Human Behavior, 120*, 106746. https://doi.org/10.1016/j.chb.2021.106746

Tikva, C., & Tambouris, E. (2021). A systematic mapping study on teaching and learning computational thinking through programming in higher education. *Thinking Skills and Creativity, 41*, 100849. https://doi.org/10.1016/j.tsc.2021.100849

Velázquez-Iturbide, J. Á. (2013). An experimental method for the active learning of greedy algorithms. *ACM Transactions on Computing Education, 13*(4), Article 18, 1–23. https://doi.org/10.1145/2534972

Wei, X., Lin, L., Meng, N., Tan, W., Kong, S.-C., & Kinshuk. (2021). The effectiveness of partial pair programming on elementary school students' computational thinking skills and self-efficacy. *Computers & Education, 160,* 104023. https://doi.org/10.1016/j.compedu.2020.104023

Wing, J. M. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33–35. https://doi.org/10.1145/1118178.1118215

Yusuf, A., & Noor, N. M. (2024). Modelling students' algorithmic thinking growth trajectories in different programming environments: An experimental test of the Matthew and compensatory hypothesis. *Smart Learning Environments, 11*, Article 38. https://doi.org/10.1186/s40561-024-00324-7

1* Corresponding author: lennykhedif@uitm.edu.my

UNIVERSITI TEKNOLOGI MARA

eISSN: : 2682-7840