MATHEMATICAL PROBLEM-SOLVING BY MEANS OF COMPUTATIONAL THINKING AND PROGRAMMING: A USE-MODIFY-CREATE APPROACH

Nils-Kristian Hansen
University of Agder, Institute of Mathematical Sciences, Kristiansand, Norway

Said Hadjerrouit
University of Agder, Institute of Mathematical Sciences, Kristiansand, Norway

ABSTRACT

This paper aims at using a Use-Modify-Create approach to explore students' mathematical problem solving by means of computational thinking (CT) and programming activities. The data collection method is participant observation, in which the researcher also has the role as teacher, guiding the group activities. In our study, two groups of students at the undergraduate level solving a mathematical task. The main finding of the study shows that the progression through the Use-Modify-Create continuum did not work as expected and that the connections between mathematical thinking, computational thinking, and programming proved difficult for the students. Conclusions so far are drawn from the study to promote mathematical problem solving by means of computational thinking and programming in a Use-Modify-Create context.

KEYWORDS

Algorithm, computational thinking (CT), mathematical problem-solving, mathematical thinking (MT), programming, Python, Use-Modify-Create (UMC)

1. INTRODUCTION

The existing body of literature on programming in mathematics education research has primarily been informed more by constructionist and individual perspectives than by conceptual and pedagogical approaches to mathematical problem-solving such as Use-Modify-Create (UMC) or similar frameworks for supporting progression in learning programming. Moreover, empirical data collected in our previous research (Hansen & Hadjerrouit, 2023; Hadjerrouit & Hansen, 2022) do not sufficiently account for the connectedness of mathematical thinking (MT), computational thinking (CT), and programming. Given this background, we argue that a pedagogical approach supporting learning progression may help students to develop an enhanced understanding of mathematical problem solving by means of CT and programming. The objective of the study is to use a Use-Modify-Create (UMC) approach to analyze undergraduate students' ability to solve mathematical problems by means of CT and programming.

The research question of this study is as follows*: How do students use, modify, and create computer programs to solve mathematical problems by means of computational thinking and programming activities?*

The article is structured as follows. Firstly, the theoretical background is outlined. Secondly, the UMC approach, context of the study, participants, and methods are described. Then, the results are reported and analyzed, followed by a discussion. Finally, conclusions, limitations, and future work conclude the article.

2. THEORETICAL BACKGROUND

We start with a brief overview of the existing research literature on the integration of CT and programming activities into undergraduate mathematics education. The purpose is to uncover and delineate the use of CT and programming to support problem solving in mathematics education research during the last few years.

2.1 Brief overview of the literature

The recent trend of integrating programming into the mathematics subject at the undergraduate level suggests the importance of CT as a core notion beyond programming in the context of mathematics education.

CT is more about thinking than computer programming (Li, Schoenfeld, & diSessa, 2020; Misfeldt & Ejsing-Duun, 2017). More specifically, it represents a "universally applicable attitude and skill set everyone, not just computer scientists, (…)" (Wing 2006, p. 33). More specifically, CT is a fundamental skill and shares several communalities with mathematical thinking (MT), e.g., problem solving and logical reasoning. MT involves the application of mathematical skills to solve mathematical problems. CT is "an approach to solving problems in a way that can be implemented using a computer" and goes beyond computer programming (Barr & Stephenson, 2011). Moreover, it is assumed that CT improves both logical and reasoning skills in mathematics education (Martínez-García, 2021). The close connection between MT and CT provides good opportunities for mathematical problem-solving.

Likewise, programming is closely related to CT, but traditionally, programming in educational settings required producing algorithms from scratch without the mediation of CT (Kaufmann & Stenseth, 2020). Recently, it is recommended to pass through CT to be able to design algorithms. Being able to programs and test program codes is the result of being able to think computationally (Li et al., 2020; Shute, Sun, & Asbell-Clarke, 2017; Wing 2006, Wing 2008). Clearly, programming alone without a clear connection to CT may not be sufficient to support students in mathematical problem solving. Thus, CT skills are critical for building efficient algorithms for mathematical problem-solving rather than trial-and-error and getting the program to run (Topallia & Cagiltay, 2018). In other words, CT requires students to be engaged in a problem-solving process until an appropriate algorithm is found that can be translated into a computer program.

Moreover, our previous research shows that integrating CT and programming with mathematics is a challenging task, and it has scarcely been explored in undergraduate education (Hadjerrouit & Hansen, 2022; Hansen & Hadjerrouit, 2023). The main findings of this research are twofold. Firstly, the introduction of CT at the undergraduate level presents many challenges both for students and teachers. Secondly, the mathematical tasks presented were challenging for novice students due to their lack of prior knowledge of CT and programming, varied mathematical problem-solving skills, and the difficulty to connect MT, CT, and programming in an effective way.

Hence, efforts must be put on a more encompassing view of the relationship between CT, MT, and programming. But still, a great challenge remains to make CT consequential and accessible to all students. As a result, it becomes important to broaden the view of CT as a fundamental skill and as a model of thinking that is important for all students. Our hypothesis is that developing and using pedagogical approaches for mathematical problem-solving by means of CT and programming may open new opportunities. At the same time, new challenges may become visible, and each of these challenges will require tremendous effort. The same is the case for specifying CT competencies that serve as a foundation for mathematical problem-solving.

2.2 MT, CT, and programming and their interplays

MT, CT, and programming have a lot of communalities (Weintrop, 2016; Wan-Rou Wu & Kai-Lin Yang, 2022). Firstly, MT consists of solving mathematical problems, (e.g., algebraic equations or functions), and justification for solutions (Shute, Sun & Asbell-Clarke, 2017, p. 145). Secondly, Wing (2008, 2014) pointed out that the main commonality between CT and mathematical thinking is problem-solving and a structured step-by-step construction process. Thirdly, CT and programming constructs such as variables and flow statements (if-then-else, for, while-until, repeat, etc.) are closely connected to mathematical thinking (Lie, Hauge, & Meaney, 2017). The close connection and interplay between MT and CT might provide opportunities for mathematical problem-solving (Figure 1). As one can see the approach is not linear, moving from MT, to CT to programming, instead transitions back are possible to make sense of algorithm and program code.

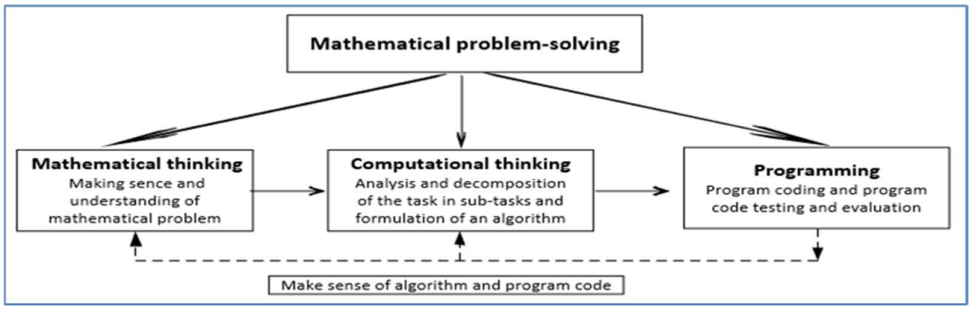

Figure 1: MT, CT, and programming and their interplay (Hadjerrouit & Hansen, 2022)

2.3 Use-Modify-Create (UMC)

The Use-Modify-Create (UMC) approach is a framework for supporting progression in learning programming. (Franklin, 2022; Houchins, 2021; Lee et al., 2011). Learners move along a continuum from where they first use programs made by someone else, e.g., teachers. Then, they modify programs developed by other people (e.g., teachers) so that the modified code becomes "theirs". Finally, in the third step, they create their own programs (Figure 2).

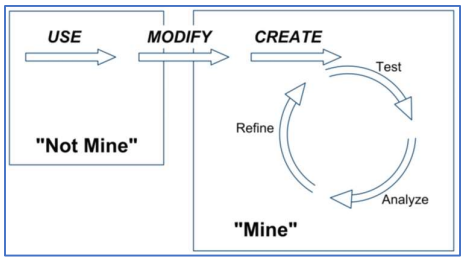

Figure 2: Use-Modify-Create model (Lee et al., 2011)

We think that the UMC approach offers a helpful progression for developing such skills over time. Its greatest advantage is in illustrating the benefits arising from engaging students with progressively more complex tasks and giving them increasing ownership of their learning (Lee et al., 2011). In the Use-phase students are consumers of someone else's creation. For example, testing, debugging, or interpreting a program. In the Modify-phase they begin to modify the program with increasing levels of sophistication, e.g., changing the value of a variable or modifying a loop. In the Create-phase, the students may want to change the behavior of the program in a way that entails developing new pieces of code, through a series of tests, analyses, iterative refinements, and attempts to validate the program against a mathematical problem, which becomes one's own. Within this "Create" phase, key aspects of CT come into play, connecting to MT and programming. Clearly, this phase necessitates MT and CT to a higher degree in comparison with previous phases. Moving through this progression, it is important to maintain a level of challenge difficulty that supports growth while limiting frustration and anxiety, considering that the approach is not fully linear, moving from Use to Modify to Create, instead transitions back and forth (Lee et al. 2021).

3. THE STUDY

Grounded on a qualitative research design this work uses a case study to collect data by observing the work of 2 groups of students in a first-year undergraduate course on programming with applications in industrial mathematics. UMC is used as an analytical framework to evaluate students' activities.

3.1. Data Collection Method and Participants

The data collection method is participant observation based on qualitative research, with a slight modification. The researcher had the role as a teacher (T), guiding the students when they got stuck. The work was performed in one group of three students (S1, S2, and S3) and one group of four (S4, S5, S6, and S7), using the video conferencing tool Zoom with screen sharing. In the Use-phase the teacher shared the programming environment, during the Modify- and Create-phases a student undertook this responsibility. All students had video and sound turned on, so that they were able to see and hear each other as well as to observe the program code. The participants were volunteers from a class of 86 students, having varied background in mathematics, but were about to complete an introductory course in programming. This course introduced basic Python constructs like variables, loops, branches, functions, and classes, but also dealt with structured programming and CT.

3.2. Use-Modify-Create tasks.

The Use-Modify-Create approach is used to progressively introduce students to new programming concepts through three activity phases. In the Use-phase they start in a programming context, trying to abstract a mathematical concept from a Python-function, (figure 3). In the subsequent Modify- and Create-phases, however, they start in an MT context, attempting to solve a mathematical challenge by programming.

```
def x(a):
    """?"""
    for n in range(a-1, 1, -1):
        if a % n == 0:
            return False
    return True
```

Figure 3. Python code used for the Use-phase.

Use-phase: In the Use-phase the students were presented with the Python-function in figure 3 and asked to describe what it performed, without testing it. The function was named "x" to avoid semantical interpretation.

In an MT context the function may be described as checking if its argument is a prime number. In a CT context it may by described as checking the parameter, a, for divisibility by all integers, n, in the range from $a-1$ down to, but not including, 1. The divisibility test is performed by examining if the remainder of a divided by n is zero. If so, a is not a prime, and the function returns *True*. If a remainder of zero never occurs, a is a prime, and the function returns *False*.

The students were also asked to reflect on if reversing the order of the integers, n, i.e., testing from 2 up to $a-1$, is correct, and if the one method is preferable over the other. In a CT context the two methods are identical, but in an MT context checking from 2 and up is preferable, as the probability of detecting divisibility is significantly larger for small numbers.

Modify-phase: Another way of stating that a is divisible by n is to say that n is a factor in a. In the Modify-phase the students were asked to modify the function as to count all factors in the parameter, a. In a programming context this involves introducing an accumulator variable, updating it for each factor found, and ultimately returning its value. MT considerations are that the numbers 1 and a also should be considered factors, and that for each positive factor a corresponding negative factor exists. One algorithmic solution to counting negative factors is extending the range of n to $[-a, a]$. This, however, requires excluding $n = 0$ from the divisibility test. Another solution is to keep the range as $[1, a]$ and add 2 to the accumulator variable for each factor found.

Create-phase: In the Create phase the students were asked to write a Python-function returning a list of all factors in the parameter a. It may be noted that the programming pattern for this closely resembles the pattern in the Use-phase, so modifying the existing code may be a better choice than writing it from scratch. The modifications required are that a composite variable like a list must be used instead of an accumulator variable, and that each factor found must be concatenated to that variable.

3.3. Data analysis

The analysis of the results seeks indications of students' mathematical problem-solving by means CT and MT in an UMC context. It uses a deductive-inductive approach based on the interplay between the theoretical basis of the study and the empirical data (Patton, 2002). Specific questions are addressed when analyzing the data such as: To what extent do students successfully complete the UMC activities? To what extent does UMC support student learning progression? To what degree do students modify the programs? To what extent do students focus on mathematical problem-solving? To what extent do they benefit from CT to develop an algorithm in the Create-phase? To what degree do they use MT and CT to create a program?

4. RESULTS

4.1 Group 1

Use-phase: T presents the Python function, asking the students to take some time to reflect on what it does. The students start in a programming context, engaging in a line-by-line discussion, but are unable to give an overall MT or CT description of the code. Even after T has given an example, a strong hint is required for them to switch to an MT-context:

T: So, we agree on that if *a* is not divisible by any of the smaller numbers, the function returns *False*. Do you follow me on that?

S2: Yes.

T: Can we say that in another way? What property does *a* have, then? Can you find a mathematical concept that describes …

S1: A prime number.

T then demonstrates that the function returns *True* on 11 as an argument and *False* on 10, and eventually S1 can describe the function in an MT-context:

S1: The function checks if a number is a prime.

T now reverses the range to count upwards from 2 and asks if the two methods are equal. S1 responds in a CT-context and concludes that they are. S3 however responds in an MT-context concluding that upwards from 2 is more efficient because one more frequently will find a multiple of the lowest numbers.

Modify-phase: S2 copies the code and shares the programming environment. T requires the students to modify the code so that it counts the factors in the parameter *a*.

The students do not engage in any CT or MT strategy discussion but goes straight to programming. S1 initially instructs S2 on what to write, but eventually S3 takes over the role as instructor. Together they write code that is correct. A test is run, and S3 first claims that the result is incorrect, but when T insists on it being correct, S3 realizes that the function counts all factors, not only prime factors:

S3: Yes. Now you get alle the numbers that can be factors, you do not get the numbers in a prime factorization.

On a question from T on whether the numbers 1 and *a* also should be considered factors in the number *a*, the students are unable to give an answer. But when T states that this is the case, S2 and S3 immediately rewrite the code so that 1 and *a* are included in the divisibility test.

When T asks for negative factors to be included in the count, S3 instantly suggests ranging from $-a$ to a, and S2 modifies the code accordingly. They do, however, not employ MT to foresee that the code will produce a division by zero-error. But when this error occurs, they immediately realize what the problem is:

T: Well, what did go wrong?

S3: Modulo by zero.

S1/3: Yes.

S3: You cannot divide by zero.

S2: No.

However, the students do not find a simple programming solution to this. The suggestions they provide are awkward in a CT setting, involving code duplications. T needs to guide their work in detail.

In contrast, when T asks for another, mathematical way to count the negative factors, S1 immediately employs MT, stating that one may count two factors at a time, as there is an equal number of negative and positive factors. S2 then modifies the code accordingly.

Create-phase: T now asks for a function returning all the factors in a. The students choose to modify the existing function.

S3 sees the need for a data structure to collect the factors and suggests replacing the accumulator variable with a list. Furthermore, S3 appears able to use CT to model the required algorithm mentally, giving S2 detailed coding instructions. S1 spots the problem that only positive factors are included and suggest expanding the range to $[-a, a]$, excluding zero, as done in the Modify-phase. A final test does not reveal any errors.

4.2 Group 2

Use-phase: T presents the Python function, asking the students to take some time to reflect on what it does. After a while S7 describes the workings of the details in the program and concludes that it does a divisibility check. S6 is then able to see this in an MT context and conclude that the function is a primality checker. Then S7 is also able to see the expression $a \% n == 0$ in an MT context and conclude that it is about modulus in integer division.

T now reverses the range to count upwards from 2 and asks if the two methods are equal. S7 reflects both in a programming context and an MT context, concluding that the two methods algorithmically are the same, but that counting upwards is better, as small numbers are more prone to be factors.

Modify-phase: S6 copies the code and shares the programming environment. T requires the students to modify the code so that it counts the factors in the parameter a. Both S6 and S7 immediately sees the need for an accumulator variable. Followingly, however, they struggle with creating the counting mechanism, making several erroneous suggestions. T needs to remind them that the task is about counting factors. S4 then reintroduces the idea of an accumulator variable. The students are however unable to integrate the accumulator variable in the code, discussing if it should be placed inside or outside of the loop, apparently unaware of the algorithmic implications of their choice. Ultimately, they make the correct decision, but based only on what they have seen in previous examples. The discussion then continues along the same lines, again demonstrating the lack of CT. When advised by T, S7 eventually reaches the correct conclusion, though unable to phrase it properly:

S6: Does it matter if it is in the if-loop or the for-loop, T?

T: Yes, it does matter.

S7: It think the for-loop should be allowed to complete before we print out how many.

A test run counts four factors in the number 12. In an MT reflection S4 states that these factors are 2, 3, 4 and 6, but S6 objects that the number 12 itself is not counted. T argues that 1 and the number itself should also be considered factors and prompts the students to modify the code accordingly. S6 then has no difficulty in doing the proper modifications.

T now asks if negative numbers can be considered factors, and S6 and S7 hesitantly agrees to yes. T suggest having the program counting them also. S6 wants to expand the range from $-a$ to a, but S7 employs MT and suggests counting two factors at the time instead, pointing out that this also will avoid a division by zero. S6 writes the correct code.

Create-phase: T asks for a function returning all the factors in a. The students choose to further modify the existing function. S6 comments that they already have touched on the thought and invites S5 and S4 to give input. S4 states the need for a list, and S6 replaces the accumulator variable. S4 replies confirmative to S5s question on if the same loop-range can be used, but S7 realizes that this will exclude negative numbers. S6s algorithmic suggestion for solving the problem is however awkward, involving cloning the list. S6 suggests extending the range to $[-a, a]$ but notes the MT challenge of zero division. Eventually S4 produces the idea of appending $-n$ to the list along with n. S6 modifies the code and runs a test that does not reveal errors.

Next T asks how to get the factors in ascending order. S7 has a correct CT suggestion of adding a sort, but employing the idea in program code proves challenging. Eventually the students come up with a solution, sound both in an MT and a CT context, but involving superfluous code that can be eliminated by using an extended range and testing for zero divisors. However, the students seem unable to write the proper code. S6 and S7 bring forward a range of suggestions, but they are either incorrect or exceedingly complex. T eventually has to dictate the solution.

5. DISCUSSION

The research question "*How do students use, modify, and create computer programs to solve mathematical problems by means of computational thinking and programming activities?"* focuses on the structured activities underlying UMC to ensure a progression in learning associated with an exploration that allows for drawing on the interactions between MT, CT, and programming, and the extent to which the students benefit from the interactions. We draw on the deductive-inductive approach to reflect on the results achieved so far. A summary of our reflections on the groups' work is given below, organized according to the UMC approach.

Use-phase: Both groups managed to do a syntactical analysis of the program, but only group 2 was able to use MT and CT to abstract and describe the functionality on a general level, and to reflect on code details in a mathematical context. Once the functionality was established however, both groups were able to deduct the mathematical implications of a modification to the program.

Modify-phase: Neither group attacked the task in a structured manner, being unable to draw on their experience from the Use-phase. They did not engage in any CT or MT strategy discussion but went straight to programming. Group 1 was able to do minor modifications, but none involving major structural changes. Group 2, in contrast to their success in Use-phase, now were unable to make any connection between the mathematical context and the code. They went about the coding in a trial-and-error fashion, apparently without understanding the algorithmic implications of their trials. Substantial assistance from T was required.

Create-phase: Discovering the similarities between this task and the Modify-task, both groups chose to adapt the existing code instead of writing new from scratch. With the major mechanism required already established in the Modify-phase, group 1 was able do the necessary programming without aid. As for group 2, their trial-and-error approach again impeded their work, and substantial assistance from T was required.

The main finding of this study is that connections between MT, CT and programming proved difficult for the students (see fig. 1). The transition from programming to mathematical idea in the Use-phase however appeared to run smoother than going the other way, solving a mathematical problem with program code in the Modify and Create phase. In the latter phases the students to a very little degree were able to profit from their experience in the Use-phase. Also, even though, as stated by Lie et al. (2017), there is a close connection between CT and programming constructs, the students almost took no advantage of this. This may be an indication of that the connection is not obvious to the students, when not pointed out and exercised.

Based on this background, the challenges that need be overcome are as follows. Firstly, while the transition from programming to mathematical notion in the Use-phase worked relatively well, the results show that modifying a given program, and create a new one for solving mathematical problems are more challenging than originally expected. Consequently, the students were thinking at levels not explicitly related to the various phases of the UMC learning progression.

Secondly, referring to figure1, it appears that the lack of CT in the UMC continuum creates a gap in the flow between MT and programming. This may explain some of the major difficulties occurring when modifications to the program structure were required. However, instructor guided programming seemed to aid bridging the gap.

Thirdly, in accordance with the research literature (Hadjerrouit & Hansen, 2022; Hansen & Hadjerrouit, 2023; Wan-Rou Wu & Kai-Lin Yang, 2022; Martínez-García, 2021), the study shows the crucial role of CT as a fundamental skill and as a model of thinking relevant to mathematical problem-solving. To make effective use of CT in the Create-phase in particular, students should be able to analyze and decompose the mathematical task into smaller sub-tasks by means of CT, and then develop an algorithm before programming. In other words, we expect a progression that encourage students to work creativity and exploratory (Franklin, 2022; Romero et al, 2017).

Finally, in line with our previous research (Hansen & Hadjerrouit, 2023; Hadjerrouit & Hansen, 2022), this study shows again that the role of the instructor is still important to assist students in their progression through the entire UMC continuum, in particular in the Create-phase.

6. CONCLUSIONS, LIMITATIONS, AND FUTURE WORK

The present study is an attempt to contribute towards the advancement of the potential value of CT and programming at the undergraduate level when students engage in mathematical problem-solving, while

uncovering both limitations and challenges encountered by the students when trying to connect MT, CT, and programming. The aim is to use a Use-Modify-Create approach and see if this contributes to a better understanding of mathematical problem-solving.

Limitations of this work include potential population bias and our implementation of the UMC method. Firstly, the number of participating students is low for a generalization. Also, as the participants were volunteers, one may assume that they were higher performing students than average. Secondly, a flaw in the design of the tasks associated with the UMC approach was that the Create-task was so similar to the Modify-task that it in reality consisted of two Modify-tasks. In addition, it should be noted that progression did not work as expected. Indeed, moving from Use to Modify to Create, back and forth as described by Lee et al. (2021) required increasing levels of CT, something the students were unable to employ sufficiently without the guidance of the teacher.

Future work will comprise more student groups to increase the study validity and reliability. Further studies will also be conducted with focus on the type and level of CT required in the process of mathematical problem solving.

REFERENCES

Barr, V., & Stephenson, C. (2011). Bringing computational thinking to K-12: What is involved and what is the role of the computer science education community? *ACM Inroads* 2(1), pp. 111-122.

Franklin, D. (2022). An analysis of Use-Modify-Create pedagogical approach's success in balancing structure and student agency. *ICER '20*, August 10–12, 2020, Virtual Event, New Zealand, pp. 14-24.

Hadjerrouit, S.; Hansen, N. K. (2022). Undergraduate mathematics students engaging in problem-solving through computational thinking and programming: A case study. *Orchestration of Learning Environments in the Digital World.* Springer Nature, pp. 197 - 214.

Hansen, N.K., & Hadjerrouit, S. (2023). Analyzing students' computational thinking and programming skills for mathematical problem solving. In: D. Ifenthaler, D.G. Sampson, & P. Isaías (Eds.) (2023). *Open and Inclusive Educational Practice in the Digital World* (pp. 155-173). Springer Nature Switzerland AG 2023.

Houchins, J. (2021). How Use-Modify-Create brings middle grades students to computational thinking. *International Journal for Designs and Learning, 12*(3), pp. 1-20.

Kaufmann, O. T., & Stenseth, B. (2020). Programming in mathematics education. *International Journal of Mathematical Education in Science and Technology*, 52(7), pp. 1029–1048.

Misfeldt, M., & Ejsing-Duun, S. (2015). Learning mathematics through programming: An instrumental approach to potentials and pitfalls. In K. Krainer, & N. Vondrová (Eds.). *Proceedings of CERME9* (pp. 2524-2530).

Lee, I. et al. (2011). Computational thinking for youth in practice. *ACM Inroads 2*(1), pp.32-37.

Li, Y., Schoenfeld, A. H., diSessa, A. A., et al. (2020). Computational thinking is more about thinking than computing. *Journal for STEM Education Research*, 3, pp. 1–18.

Lie, J., Hauge, I. O., & Meaney, T. J. (2017). Computer programming in the lower secondary classroom: Mathematics learning. *Italian Journal of Educational Technology*, *25*(2), pp. 27-35.

Martínez-García, E. (2021). Computational thinking: The road to success in education. *Academia Letters*, pp.1-7.

Patton, M. Q. (2002). *Qualitative research & evaluation methods*. London: Sage Publications.

Romero, M., Lepage, A., & Lille, B. (2017). Computational thinking development through creative programming in higher Education. *International Journal of Educational Technology in Higher Education* 14, pp. 1-15.

Topalli, D., & Cagiltay, N. E. (2018). Improving programming skills in engineering education through problem-based game projects with Scratch. *Computers & Education*, *120*, pp. 64-74.

Shute, V.J., Sun, C., & Asbell-Clarke, J. (2017). Demystifying computational thinking. *Educational Research Review*, pp. 1-10.

Weintrop, D. et al. (2016). Defining computational thinking for mathematics and science classrooms. *J Sci Educ Technol 25*, pp. 127–147.

Wan-Rou Wu & Kai-Lin Yang (2022) The relationships between computational and mathematical thinking: A review study on tasks, *Cogent Education*, 9:1, pp. 1-19.

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A*, *366*(1881), pp. 3717-3725.

Wing, J. M. (2006). Computational thinking. *Communication of the ACM*, 49(3), pp. 33–35.