24/1 (2026), 29–49
DOI: 10.5485/TMCS.2026.15760
tmcs@science.unideb.hu
https://ojs.lib.unideb.hu/tmcs

Teaching
Mathematics and
Computer Science

Removing the burden of syntax: developing computational thinking and algorithmic skills of STEM students

ÁDÁM GULÁCSI and MÁRIA CSERNOCH

Abstract. In higher education, solving programming exercises using a high-level programming language is a standard approach for developing computational thinking and algorithmic skills. However, this method has its limitations: learning the syntax of a high-level programming language puts an extra cognitive load on students, preventing them from focusing on problem-solving. Furthermore, computational thinking is not limited to programming: STEM students can benefit more from solving problems within their own discipline, in different environments. This practical article proposes a collection of unplugged, semi-unplugged and plugged-in alternatives that can be used to develop the computational thinking and algorithmic skills of students.

Key words and phrases: computational thinking, algorithmic skills, STEM education, unplugged, cognitive load.

MSC Subject Classification: 97P99.

Introduction

In college- and university-level STEM education, computational thinking is frequently part of the curriculum (Takács et al., 2022). Unfortunately, this usually manifests in the form of high-level programming classes that are deemed beginner-friendly. There are multiple problems with this approach.

First, only a small subset of STEM students will become software developers or software engineers. For everyone else, learning the features and syntactic

rules of a particular programming language has little real benefit. Furthermore, learning syntax and language-specific tools consumes valuable time from developing computational thinking that is language- and tool-independent (Wing, 2006). It is a universal skill that is beneficial for anyone, including STEM students. Even if students need to learn to program computers later, having a solid foundation of computer science terms and concepts helps ease the cognitive load of learning a new language (Kahneman, 2011; Kirschner et al., 2006; Lister et al., 2006; Sweller et al., 2011).

Contemporary education research suggests presenting interdisciplinary, real-world problems in STEM education (Wolfram, 2020), which was first proposed as early as the 1980s (Papert, 1980; Soloway, 1993). The problem-solving process should be iterative and algorithmic, with analysis, design, execution and reflection as integral parts (Pólya, 1957; Wolfram, 2020). The problem with using high-level programming languages for this purpose is that there are too many concepts, rules and tools to be learned before students can reach an adequate level of problem-solving (Lister et al., 2006).

This paper serves the practical purpose of presenting an alternative. The following section establishes a theoretical background for the pedagogical toolset presented later. The second section details the problems of using high-level programming languages to teach STEM students at a beginner level. The last section presents a set of activities and methods that can be used to develop computational thinking and algorithmic skills. These are split into three different groups, based on how many computers are needed in the classroom or lecture hall.

Theoretical background

The concept of computational thinking (CT) has been approached from various theoretical perspectives, leading to numerous definitions. This paper draws upon Jeannette Wing's refined definition:

> Computational thinking is the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer – human or machine – can effectively carry out. (Wing, 2017, p. 8)

The presented activities in this paper are grounded in the didactic principles outlined by George Pólya's four-step problem-solving framework:

> First, we have to understand the problem [...] Second, we have [...] to make a plan. [...] Third, we carry out our plan. Fourth, we look back at the completed solution, we review and discuss it. (Pólya, 1957, pp. 6–7)

Wing's definition of CT emphasizes the importance of problem formulation, which corresponds to Step 1 and Step 2 in Pólya's problem-solving algorithm. The process of expressing solutions involves implementation (Step 3), followed by evaluation, debugging, and generalization (Step 4). Additionally, Wing's definition highlights two crucial factors that imply didactic considerations: firstly, the computer executing the solution can be a human; secondly, the problem-solving process should be effective. The combination of these concepts and procedures serves as the theoretical foundation for the activities presented in the third section of the present paper.

Issues with high-level programming languages

The burden of syntax

Using a high-level programming language to teach computational thinking is a common practice in folk pedagogy (Lister, 2008). However, as being teachers and professors with years of programming experience it is challenging to communicate effectively with novice students. Even writing a simple program like the infamous, tedious and dull "Hello World!" program requires substantial technical and syntactical knowledge (Guzdial & Soloway, 2002). These syntax-related rules and knowledge items quickly accumulate, diverting time from problem-solving, while also putting an extra cognitive load on students (Kahneman, 2011; Kirschner et al., 2006; Lister et al., 2006; Sweller et al., 2011).

The misplaced focus on syntax also impacts assessment. Quizzes and tests relating to a specific high-level programming language require students to analyze them through different lenses. Relying on a language, makes it easy to assess language-specific knowledge instead of universal computer science concepts (see Figures 1 and 2).

Un premier programme

Voici un premier petit programme écrit en Python :

```
a = 4
b = 7
print("À vous de jouer")
x = int(input())
y = int(input())
if x == a and y == b:
  print("Coulé")
else:
  if x == a or y == b
    print("En vue")
  else:
    print("À l'eau")
```

Figure 1. Beginner Python exercise: using int(input()) is language-specific (Dowek et al., 2013)

```c
#include <stdio.h>

int main()
{
    int i;
    for (i=0; i<10; i++)
        printf("Hello!\n");

    return 0;
}
```

```c
#include <stdio.h>

int main()
{
    int i;
    for (i=0; i<10; i++);
        printf("Hello!\n");

    return 0;
}
```

How many times will "Hello!" be printed in the above program?

A) 0 B) 1 C) 9 D) 10

Figure 2. Emphasis on C syntax with the extra semicolon in the second question (right)

Compare the two questions in Figure 2: introducing an extra semicolon in the second question (right) clearly measures syntactic knowledge and has little to do with computational thinking. Even the first question (left) has its own problems that will be detailed in a later section. One might argue that using flowcharts, pseudocode or similar tools comes with its own syntax. While that is true, these rules are less complex and more forgiving than high-level programming languages. Tools like Flowgorithm also have built-in visual programming interfaces (Cook, 2024). These make it even easier to focus on problem-solving instead of syntax.

Language-specific features

Solving real-world problems with a high-level programming language requires sufficient knowledge of language-specific features and tools. These not only require extra time and effort to learn, but also often prove to be confusing for beginners. An argument can be made that other solutions have their own toolsets. For example, you must learn the meaning of the different shapes when using flowcharts to build algorithms and programs. Even then, it only provides a minimal set of tools, all stemming from language-independent computer science concepts that are proven to have longevity (Buitrago Flórez et al., 2017).

What makes it much worse in the case of high-level programming languages? A practical example is introducing loops via the infamous "for" loop construct. Unfortunately, this common practice is particularly confusing from a beginner's perspective. One of the first things students learn is how algorithms are sequential and the order of execution matters. Having to deal with how loops change the default flow of the program is challenging enough in itself. When the for loop is used for this purpose, students must untangle a complex, non-trivial flow of execution (Figure 3). Try to answer these questions for each of the three expressions:

- Is the statement part of the loop? (no, yes, yes)
- When does the statement execute? (once before the loop, before each iteration, at the end of each iteration)
- What is the execution order of the statements?

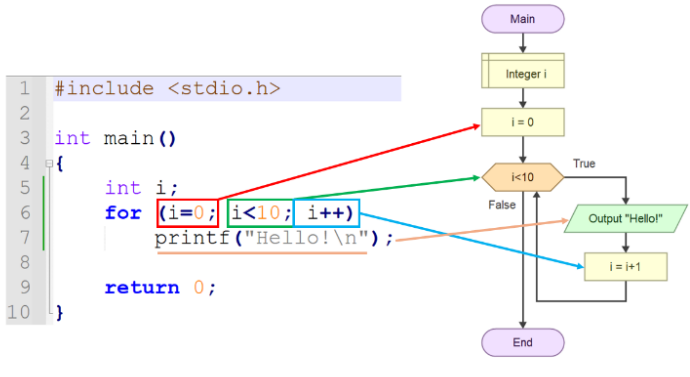

Figure 3. Untangling a C for loop can be confusing, even for more intermediate programmers

How helpful is the syntax of the for loop when you try to answer those questions? From a beginner's perspective, this is a considerable challenge. Computational thinking already requires students to think at multiple levels of abstraction (Wing, 2006). Introducing cognitively demanding constructs such as the for loop in addition to an already challenging task makes the work more difficult for them.

This does not mean that high-level programming languages cannot be used to introduce the concept of loops. With a "while" loop used instead, it is easier to write the C program equivalent to the flowchart in Figure 3. That said, this example still portrays the problem well: when teachers and curricula focus on teaching language-specific tools instead of focusing on computational thinking, it hinders the learning process (Pólya, 1957; Wolfram, 2020).

Honing an axe instead of using a Swiss Army knife

A final issue of exclusively relying on a high-level programming language to develop computational thinking comes from this exclusivity. A person can spend years honing and swinging an axe. Chopping up logs will become second nature to that person. However, opening a can with that axe – while not impossible – is still not the most effective way to do it.

In computational thinking terms, take the Hungarian national curriculum textbook as an example: students learn Python throughout their K12 education (Ministry of Justice, 2020; Abonyi-Tóth et al., 2020). Assume that the school has a small weather station that has been collecting data for over a year. As an interdisciplinary project, students need to answer questions based on the collected data. The project involves mathematical calculations, data analysis and data visualization. If their computational thinking skills were solely developed by programming in Python, the list of prerequisites for this project is considerably long:

- solid foundation of computer science concepts: data types, variables, conditionals, loops, data structures, common algorithms;
- solid foundation of the Python language: variables, conditionals, loops, lists, dictionaries, reading data from files;
- data analysis libraries (e.g., numpy, pandas);
- data visualization libraries (e.g., matplotlib).

It is evident that Python is not the best tool for this job in a K12 classroom. Doing the project with an algorithm-oriented spreadsheet approach (e.g., Sprego), drastically reduces the amount of time and effort spent on the project (Csapó et

al., 2020a, 2020b; Csernoch, 2014; Csernoch & Biró, 2015; Csernoch et al., 2015). At the same time, a spreadsheet software can introduce those previously mentioned concepts of computer science, similar to any high-level programming language (Csernoch, 2014). Picking the appropriate tool for the task is just as important as solving the problem. One can think of computational thinking as a versatile Swiss Army knife as opposed to a sharp axe: these skills are transferable and applicable in many different contexts.

Alternatives to high-level programming languages

This section details the available alternatives – non-traditional programming approaches – to develop the computational thinking and algorithmic skills of STEM students. These didactic tools and methods can be used both with K-12 and higher education students. However, we must emphasize that teachers must use authentic, real-world input and data to create meaningful tasks and scenarios which are relevant to the students (Csernoch, 2025). The same underlining methods can be used to develop CT, while the data and the specific tasks are tailored to the target group's previous background knowledge, interests, and age characteristics (Csernoch, 2025).

These tools and methods are categorized into three practical subsections based on how many computers are needed when using them. However, these should not be treated as mutually exclusive categories; they can be mixed and matched as needed.

Unplugged

Using unplugged methods has multiple advantages. First, no computers are needed in the classroom. Second, having no computers means that there are no distractions: the lack of interfaces, IDEs means that students can focus on algorithms and computational thinking at its core (Bell & Newton, 2013; Biró & Csernoch, 2017).

These tools and exercises work best with foundational computer science topics: fixed- and floating-point binary arithmetic, character encodings and basic algorithms such as finding a minimum or maximum value, conditional counting, linear search, binary search and sorting algorithms.

Additionally, the most impactful advantage of using unplugged methods is that it prevents students from skipping the first and second crucial steps of understanding the problem, analysing the data and creating a plan for the problem-solving process (Pólya, 1957; Wolfram, 2020). Students who start solving a problem in front of a computer often opt for a time-consuming trial-and-error process that is ineffective (Reynolds, 2008). Empirical research in unplugged computer science pedagogy suggests that these activities are best used to introduce topics and should always be followed by semi-unplugged or plugged-in activities (Bell & Vahrenhold, 2018). They are not alternatives to traditional instruction, but a way to complement them and reduce the cognitive load of computer-use from the beginning.

Methods from drama pedagogy can be used as a visual and tactile way to approach computational thinking and algorithmic skills. Students become active participants as they identify and assume a role within the algorithm. For example, conditional counting is an algorithm that determines how many elements (actors) satisfy a given condition. A real-life scenario would be a group of students visiting a theme park on a field trip (Sebestyén et al., 2018). Going on a ride is limited to those who satisfy a height-limit condition. The teacher needs to count the number of students who can go on the ride before buying the appropriate number of tickets. In this example, a green jersey is used to mark the students who pass the height check. The activity can be acted out with the following script:

(1) teacher marks the height limit with a line on the board;
(2) students form a queue in front of the board;
(3) for each student: compare height to limit; if height $\geq$ limit, go to step 4; else, go to step 5;
(4) student puts on a green jersey;
(5) student exits queue;
(6) teacher counts the number of students with a jersey.

Acting helps with visualizing and contextualizing computer science concepts and algorithms (Figure 4). Another benefit of using unplugged tools is that they create a point of reference before moving to a semi-unplugged or plugged-in environment (Gulácsi et al., 2019). For example, when students use this algorithm in the context of a high-level programming language, teachers can refer to this activity to help them recall the steps:

- "How did we start the process?"
- "What was the next step?"
- "How did we arrive at the solution?"

Figure 4. Unplugged conditional counting activity, Ferris wheel is drawn on the blackboard, selected students wearing a jersey

Digital text-processing can also be introduced in an unplugged environment. There are millions of digitally produced texts that are filled with errors (Csernoch et al., 2023). Printed documents containing errors can be used to gradually introduce students to different types of errors (Sebestyén et al., 2022). This provides an opportunity to build a mental model of errors that are visible in print before jumping into word processing software (Figure 5).

Using a printed version of the digital document is merely the beginning of the learning process: this tool is an excellent example of how unplugged activities can be combined with plugged-in methods. After finding and correcting the errors visible in print, students can move on to recognizing, understanding and fixing digital errors (Csernoch et al., 2024; Sebestyén et al., 2022). Empirical evidence shows that using the Error Recognition Model yields in better understanding of handling digital texts (Csernoch et al., 2024; Sebestyén et al., 2022).

This has multiple benefits for developing computational thinking: students learn the differences between syntactic and semantic errors. Furthermore, effectively fixing these errors requires students to have a thorough understanding of

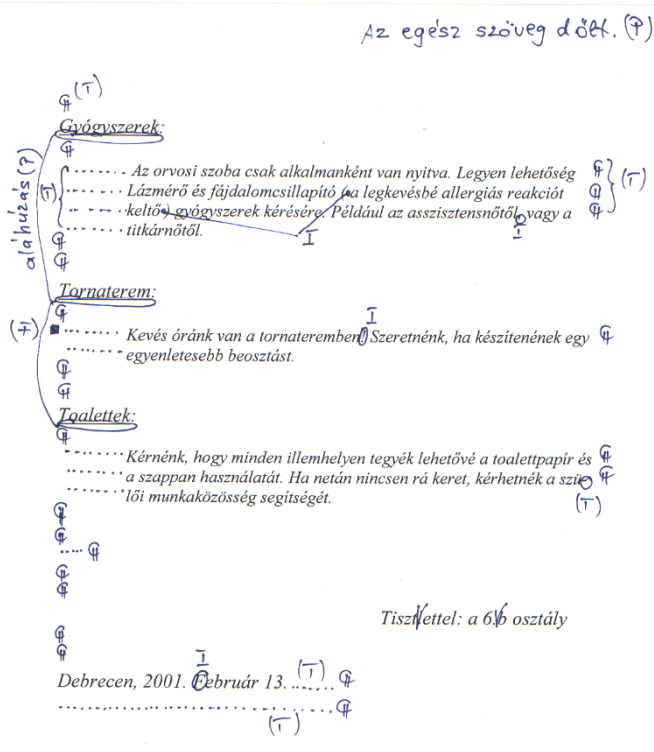

Figure 5. Marking errors in a printed document

the different scopes for formatting. Again, this is universal knowledge which can be translated into high-level programming later.

Presentation design is another area of interest for unplugged activities. Students at all levels of education are expected to create presentations. Nevertheless, presentation design is rarely discussed along the lines of computational thinking and unplugged activities.

With presentations, starting in front of a computer is quite dangerous: skipping the first step of the problem-solving process becomes too tempting (Pólya, 1957; Reynolds, 2008; Wolfram, 2020). Creating a pen-and-paper prototype forces students to think about the structure of the presentation, the number of unique slide layouts and the amount of information to display per slide (Figure 6). Applying this technique speeds up the following plugged-in process and yields better end-products (Reynolds, 2008). For another unplugged activity, the ERM

model can be used for presentations as well (detailed in the previous paragraphs) (Sebestyén et al., 2022).

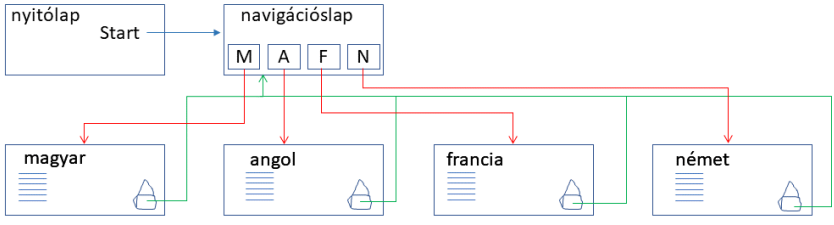

Figure 6. Pen and paper prototype for presentations

For another set of unplugged activities, physical components are useful for building and testing algorithms. For example, sorting algorithms can be practiced with a deck of playing cards and a few tokens or dice. Students can deal themselves a random hand and use a specific sorting algorithm step-by-step to see and feel how the process works (Figure 7). This activity can be used in two different ways:

- easier variation: the teacher can outline the whole algorithm on the board, step-by-step;
- harder variation: the teacher shows a video or visualization showcasing the execution of the sorting algorithm on a set of values; students then try to reconstruct the algorithm and test it with their cards.

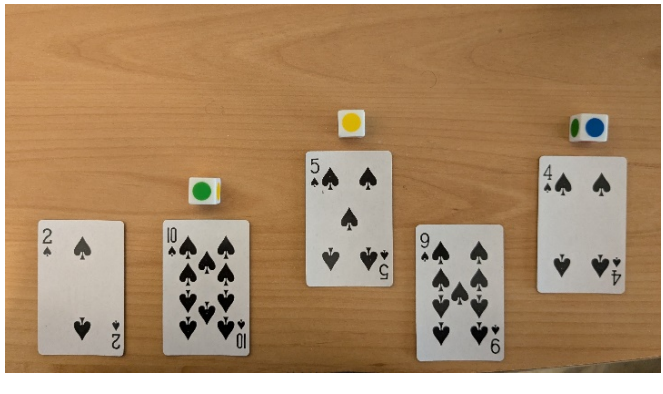

Figure 7. Comparing two elements during a selection sort

Another fundamental concept in computer science is thinking in terms of inputs and outputs and how they relate to each other. Ordering the sequence

of steps requires understanding how the output of a previous step serves as the input of the current one (Csapó et al., 2020a, 2020b; Csernoch, 2014; Csernoch & Biró, 2015; Csernoch et al., 2015). Origami boats with different colours and sizes can be used to demonstrate this visually (Figure 8). Using this method in a spreadsheet environment is more effective than traditional methods (Csapó et al., 2020a, 2020b; Csernoch et al., 2015).

Figure 8. Origami boats used as a reference point during a plugged-in lesson

For each step of an algorithm, an origami boat is crafted. Three pieces of information must be displayed on each boat:

- input data (written inside);
- output data (written outside);
- description of what that step does (written outside).

Thus, a smaller boat can be placed inside a larger one, indicating that the output of the smaller boat is used as input for the larger one. As an example, assume we have a table with bestselling book titles and the year of publication in parentheses:

- Title of the Book (2024)
- Another Bestseller Book Title (2023)
- ....

Table 1 details how this method can be used to demonstrate a simple algorithm that extracts the year as an integer from all the entries.

boat	step	input	output
1st (small)	display the five right-most characters	book records; 5 (number of characters)	XXXX)
2nd (bigger)	display the four left-most characters	boat 1; 4 (number of characters)	year as string
3rd (biggest)	convert string to integer via multiplication	boat 2; 1 (multiplier)	year as integer

Table 1. Building an algorithm using boats

Semi-unplugged

Semi-unplugged tools and methods require at least one computer (or smartphone) present in the classroom. They can be used as gateways to help students' transition from unplugged to plugged-in problem-solving. There are two use-cases for semi-unplugged tools.

First, they can be used for additional visual stimulation to further facilitate and motivate students to understand certain algorithms. Second, a semi-unplugged environment is suitable for a teacher-led session where students can test how their unplugged ideas and algorithms work in a computer environment. A key advantage of this approach is that the presence of only one computer in the classroom minimizes distractions, allowing students to concentrate on the implementation and evaluation steps (Pólya, 1957; Wolfram, 2020). Not having to deal with interfaces and syntax themselves can result in fewer distractions (Lister et al., 2006). Students have time to rely on slow thinking to determine how ideas and algorithms can be translated into actionable steps in a specific environment (Kahneman, 2011; Csernoch, 2017).

There are numerous visualization tools available to demonstrate computer science algorithms. One option is to use interactive animations like Visualgo (Halim, 2024). The main advantage of using such programs and websites is visual clarity: there is nothing to distract users from the sorting algorithm (Figure 9).

However, there are visually more appealing alternatives. In particular, the AlgoRythmics research group uses dances and performances to visualize algorithms. The videos are available on YouTube (AlgoRythmics, 2024). Utilizing these leads to higher student motivation without hindering their understanding

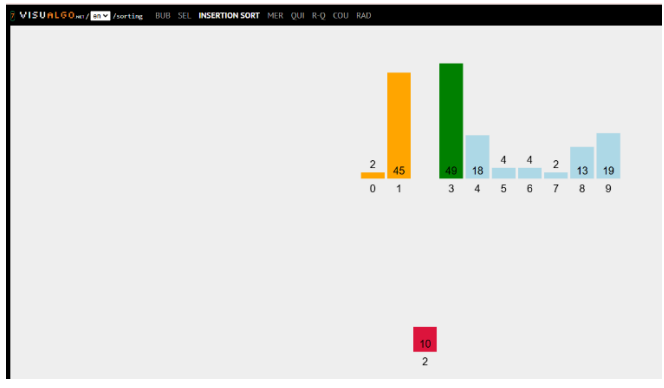

Figure 9. Insertion sorting algorithm animation playing on visualgo.net

of the algorithms (Kátai et al., 2024). Furthermore, all these visualizations can be combined with other techniques discussed in the article. For example, when students need to learn the bubble-sorting algorithm, the following set of activities can be used to develop computational thinking:

(1) Students watch the bubble-sorting algorithm AlgoRythmics video (Figure 10).

(2) Students use cards, tokens (or dice) to reconstruct the algorithm.

(3) The teacher inputs their algorithm into a program. Students then test, evaluate, and debug the algorithm.

Figure 10. Bubble-sorting with Hungarian folk dance, courtesy of AlgoRythmics on YouTube

Finally, these visualizations are not limited to sorting algorithms and data structures. There are 2D and 3D visualization tools available for linear searching and conditional counting algorithms (Csapó & Sebestyén, 2017; Dienes & Gulácsi, 2018). These tools can be combined with unplugged methods like the origami boats as they provide a visual reference point to the algorithms, much like the dances mentioned above (Figure 11).

Figure 11. Conditional counting algorithm visualizations in 2D (left) and 3D (right)

From unplugged to semi-unplugged

The ideas presented in the unplugged section are all suitable for semi-unplugged environments. For each of the examples below, the teacher operates the computer during these activities, so students can focus on computational thinking and problem-solving.

As a follow-up to the drama pedagogy unplugged activity, students can solve a similar problem in a different context. For example, the teacher presents a set of data and a conditional counting question in a spreadsheet environment. Students design an algorithm using the origami boats. Finally, they translate these steps into spreadsheet formulae. The output provided by the spreadsheet software is compared to the desired output written on the outside of each boat, which makes the evaluation step easier (Gulácsi et al., 2019).

Finding all the errors in printed digital texts in the ERM unplugged activity can be followed by opening the same document in a word processor. Students can design algorithms to correct the recognized errors. Usually, fixing those errors often reveals a new set of challenges because digital texts are full of digital errors

(Csernoch et al., 2023; Sebestyén et al., 2022). This semi-unplugged exercise is an effective gateway to plugged-in lessons, as it forces students to see the consequences of an erroneous digital text.

The main purpose of the presentation-design unplugged activity is to encourage students to use their computational thinking skills to formulate a plan before designing their own presentations (Reynolds, 2008). Error recognition is a valuable addition to this exercise. However, printing presentations is impractical, as they are almost always used digitally. Hence, the semi-unplugged environment is well-suited for presentation error recognition and presentation analysis. The teacher can present various erroneous presentations and ask students to analyse them. This exercise is not only enjoyable but also helps students develop their critical thinking and analytical skills. Before starting their own presentations, students will have already made a design and layout plan and know what mistakes and pitfalls to avoid, which leads to a better end-product (Reynolds, 2008).

Finally, using physical components to design algorithms can be directly used for testing, evaluating and debugging them in a semi-unplugged environment. For example, when students successfully emulate the steps of a sorting algorithm, the computer can be used to translate those steps into a program. Any environment or programming language is suitable for this purpose. However, it is recommended to select one with an easy debugging tool because step-by-step execution helps students with the evaluation step. For origami boats, this translation process is even simpler: all input, output and instructions are already defined for each step, it only needs to be translated to the syntax of the target environment or programming language.

Plugging it in

What happens when students sit in front of a computer? The common misconception of younger generations being more tech-savvy and so-called "digital natives" has been debunked (Kirschner & De Bruyckere, 2017). This means teaching them is equally challenging as it was before.

There are several universal definitions and concepts in computer science that are useful for STEM students: scope, variables, inheritance, loops, conditionals and so forth. These concepts have longevity, meaning they have endured since the dawn of computer science and remain foundational (Buitrago Flórez et al., 2017). However, teachers must avoid the misconception that these concepts, and computational thinking in general, should only be discussed within the realms of programming. For example, word processing programs can be used to teach the

meaning of scope; presentation master slides are an effective way to get a hands-on experience with inheritance; spreadsheets can be used to introduce variables, and other similar applications. Due to space limitations, this article does not elaborate further regarding the available plugged-in alternatives.

Limitations

While most tasks presented in the previous section have empirical evidence proving their efficiency, some do not. This research gap should be addressed, especially for semi-unplugged methods. The subjective teacher experience is positive for these methods, but empirical evidence is needed to provide a quantifiable, objective measurement regarding their efficiency.

Conclusion

In conclusion, computational thinking skills should not be developed solely through high-level programming. Focusing solely on a programming language introduces the burdens of learning syntax and language features. Additionally, focusing on one environment or language complicates the selection of the appropriate tool when faced with a new problem.

There are better alternatives available, enabling STEM students to focus on problem-solving without distractions. A combination of using unplugged, semi-unplugged and plugged-in tools offers the same benefits as learning a high-level programming language with a smaller cognitive load, fewer syntax rules and less language-specific knowledge.

Even if students need to learn a high-level programming language eventually, having solid computer science foundations and computational thinking skills provides a strong advantage. All the visual tools, real-life examples and activities build a mental model and reference point of algorithms and concepts. Both students and teachers can refer to these activities when needed during the learning process.

Acknowledgements

This article was supported by the EKÖP-KDP-2024 University Research Scholarship Program – Cooperative Doctoral Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

References

Abonyi-Tóth, A., Farkas, Cs., Jeneiné Horváth K., Reményi Z., Siegler G., Takács I., Varga P. (2020). *Digitális kultúra tankönyv 9.* [Digital Culture 9]. Oktatási Hivatal.

AlgoRythmics (2024). AlgoRythmics – YouTube. Retrieved December 14, 2024. https://www.youtube.com/@AlgoRythmics

Bell, T., & Newton, H. (2013). Unplugging computer science. In D. M. Kadijevich, C. Angeli, & C. Schulte (Eds.). *Improving computer science education* (pp. 75–90), Routledge.

Bell, T., & Vahrenhold, J. (2018). CS unplugged – how is it used, and does it work? In H. J. Böckenhauer, D. Komm, & W. Unger, *Adventures between lower bounds and higher altitudes: essays dedicated to Juraj Hromkovič on the occasion of his 60th birthday* (pp. 497–521). Springer.

Biró, P., & Csernoch, M. (2017). Unplugged tools for building algorithms with Sprego. In C. Mafalda. *Education and New Developments (END 2017)* (pp. 401–405). InScience Press.

Buitrago Flórez, F., Casallas, R., Hernández, M., Reyes, A., Restrepo, S., & Danies, G. (2017). Changing a generation's way of thinking: Teaching computational thinking through programming. *Review of Educational Research*, *87*(4), 834–860. https://doi.org/10.3102/0034654317710096

Cook, D. (2024). Flowgorithm – Flowchart Programming Language. Retrieved December 14, 2024. http://flowgorithm.org/

Csapó, G., & Sebestyén, K. (2017). Educational software for the Sprego method. *Turkish Online Journal of Educational Technology*, Special Issue for INTE 2017, *October 2017*, 986–999. http://www.tojet.net/special/2017_10_1.pdf

Csapó, G., Csernoch, M., & Abari, K. (2020a). Sprego: Case study on the effectiveness of teaching spreadsheet management with schema construction.

Education and Information Technologies, *25*(3), 1585–1605. https://doi.org/10.1007/s10639-019-10024-2

Csapó, G., Sebestyén, K., Csernoch, M., & Abari, K. (2020b). Case study: Developing long-term knowledge with Sprego. *Education and Information Technologies*, *26*(1), 965–982. https://doi.org/10.1007/s10639-020-10295-0

Csernoch, M. (2014). *Programozás táblázatkezelő függvényekkel – Sprego: Táblázatkezelés csupán egy tucat függvénnyel* [Programming with spreadsheet functions – Sprego]. Műszaki Könyvkiadó.

Csernoch M., & Biró P. (2015). Sprego programming. *Spreadsheets in Education* (eJSiE), *8*(1), 40 pp.

Csernoch, M., Biró, P., Máth, J., & Abari, K. (2015). Testing algorithmic skills in traditional and non-traditional programming environments. *Informatics in Education*, *14*(2), 175–197. https://doi.org/10.15388/infedu.2015.11

Csernoch, M. (2017). Thinking fast and slow in computer problem solving. *Journal of Software Engineering and Applications*, *10*(1), 11–40.

Csernoch, M., Nagy, K., & Nagy, T. (2023). The entropy of digital texts – the mathematical background of correctness. *Entropy*, *25*(2), Art. ID 302, 34 pp.

Csernoch, M., Hannusch, C., & Biró, P. (2024). Modification of erroneous and correct digital texts. *Entropy*, *26*(12), Art. ID 1015, 21 pp.

Csernoch, M. (2025). Lean digital education to resolve the paradox of the illusion of digital prosperity. *Journal of Innovation & Knowledge*, *10*(2), Art. ID 100676, 27 pp.

Dowek, G., Archambault, J.-P., Baccelli, E., Cimelli, C., Cohen, A., Eisenbeis, C., Viéville, T., Wack, B., & Berry, G. (2013). *Informatique et sciences du numérique: Spécialité ISN en terminale S: Avec des exercices corrigés et idées de projets*. Eyrolles.

Gulácsi Á., Dienes N., & Csernoch M. (2019). Sprego toolbox: A way to teach spreadsheeting meaningfully. *Turkish Online Journal of Educational Technology*, Special Issue for IETC-ITEC-INTE (Vol. 2), *December 2019*, 296–302.

Guzdial, M., & Soloway, E. (2002). Teaching the Nintendo generation to program. *Communications of the ACM*, *45*(4), 17–21. https://doi.org/10.1145/505248.505261

Halim, S. (2024). VisuAlgo – visualising data structures and algorithms through animation. Retrieved December 14, 2024. https://visualgo.net/en

Kahneman, D. (2011). *Thinking, fast and slow.* Penguin Books.

Kátai, Z., Osztián, P.-R., & Iclanzan, D. (2024). Enacting algorithms: Evolution of the AlgoRythmics storytelling. *Education and Information Technologies*, *29*, 19197–19228.

Kirschner, P. A., & De Bruyckere, P. (2017). The myths of the digital native and the multitasker. *Teaching and Teacher Education*, *67*, 135–142. https://doi.org/10.1016/j.tate.2017.06.001

Kirschner, P. A., Sweller, J., & Clark, R. E. (2006). Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist*, *41*(2), 75–86. https://doi.org/10.1207/s15326985ep4102_1

Lister, R. (2008). After the Gold Rush: Toward sustainable scholarship in computing. In S. Hamilton, & M. Hamilton (Eds.), *Proceedings of the Tenth Australasian Computing Education Conference (ACE'08)* (pp. 3–17). Australian Computer Society.

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin*, *38*(3), 118–122. https://doi.org/10.1145/1140123.1140157

Ministry of Justice (Ed.). (2020). *Magyar Közlöny*, *2020*(17). Retrieved December 14, 2024. https://magyarkozlony.hu/dokumentumok/3288b6548a740b9c8daf918a399a0bed1985db0f/letoltes

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas.* Basic Books.

Pólya, G. (1957). *How to solve it.* Doubleday.

Reynolds, G. (2008). *Presentation zen: Simple ideas on presentation design and delivery.* New Riders Pub.

Sebestyén, K., Csapó, G., & Csernoch, M. (2018). Visualising Sprego inequality problems with 2D representations. *Turkish Online Journal of Educational Technology*, Special Issue for INTE-ITICAM-IDEC (Vol. 2), *November 2018*, 888–898. http://www.tojet.net/special/2018_12_3.pdf

Sebestény, K., Csapó, G., Csernoch, M., & Aradi, B. (2022). Error recognition model: High-mathability end-user text management. *Acta Polytechnica Hungarica*, *19*(1), 151–170.

Soloway, E. (1993). Should we teach students to program? *Communications of the ACM*, *36*(10), 21–24. https://doi.org/10.1145/163430.164061

Sweller, J., Ayres, P., & Kalyuga, S. (2011). *Cognitive load theory*. Springer.

Takács, R., T. Kárász, J., Takács, Sz., Horváth, Z., & Oláh, A. (2022). Successful steps in higher education to stop computer science students from attrition. *Interchange*, *53*, 637–652. https://doi.org/10.1007/s10780-022-09476-2

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

Wing, J. M. (2017). Computational thinking's influence on research and education for all. *Italian Journal of Educational Technology*, *25*(2), 7–14. https://doi.org/10.17471/2499-4324/922

Wolfram, C. (2020). The math(s) fix: An education blueprint for the AI age. Wolfram Media.

ÁDÁM GULÁCSI
UNIVERSITY OF DEBRECEN, HUNGARY

E-mail: gulacsi.adam@inf.unideb.hu

MÁRIA CSERNOCH
UNIVERSITY OF DEBRECEN, HUNGARY

E-mail: csernoch.maria@inf.unideb.hu

(Received June, 2025)