Computer Science and Information Systems 21(3):781–806 https://doi.org/10.2298/CSIS230730015P

A Novel Approach to Source Code Assembling in the Field of Algorithmic Complexity

Đorđe Pešić[1], Milena Vujošević Janičić[2], Marko Mišić[1], and Jelica Protić[1]

[1] University of Belgrade - School of Electrical Engineering
Bulevar kralja Aleksandra 73, RS-11120 Belgrade, Serbia
pd235049p@student.etf.bg.ac.rs
{marko.misic, jelica.protic}@etf.bg.ac.rs
[2] University of Belgrade - Faculty of Mathematics
Studentski trg 16, RS-11000 Belgrade, Serbia
milena.vujosevic.janicic@matf.bg.ac.rs

Abstract. Computational complexity analysis plays an essential part in the education of computer and software engineers. For that reason, it is carefully studied in programming courses, as well as in the algorithms and data structures courses. The number of students who learn programming is rapidly growing, but the number of teachers cannot keep up with that trend. Therefore, it is necessary to develop tools that can ease and accelerate the daily tasks of teachers, especially for learning purposes and in the context of automating the processes of exam preparation. We propose a novel template- and rule-based approach and a corresponding software system for assembling synthetic source code segments of defined time complexity. Based on the developed grammar, the system can produce source code segments with a broad scope of different time complexities while guaranteeing the complexity of the generated segment. The system can be used for generating questions for exams as it can assemble a large number of different code segments that can be given as questions that have similar difficulty levels. The system was evaluated both by human experts and ChatGPT tool.

Keywords: automated source code assembling, computational complexity, time complexity, rule-based assembling.

1. Introduction

Software is becoming an essential part of everyday life. The software industry is growing constantly, with a high and increasing demand for well-trained programmers. Many countries invest a lot in the development of programming education, leveraging different innovative approaches from primary education [13] to higher education [60] levels. Computer science and engineering schools have an essential role in such education. For example, at the Department of Electrical Engineering and Department of Software Engineering of the School of Electrical Engineering at the University of Belgrade, introductory programming courses are taught in the first year of undergraduate studies. These courses attend more than 1000 students [33]. The number of students is expected to grow even more in the following years amid the increasing interest in those topics [48].

The number of teaching personnel does not follow this increased number of students. In addition, the number of exam periods is six times per course per year. This puts pressure on the teaching staff and makes automated test assembling and evaluation crucial in

maintaining the quality of the teaching process and helping the professors and teaching assistants in their everyday responsibilities.

Computational complexity and analysis of algorithms are fundamental disciplines in computer science. Computational complexity classifies computational problems according to their difficulty and assigns complexity classes to them [26], [14], [12]. The analysis of an algorithm describes how good it is from the quantitative point of view [26] and includes time and space complexity analysis. Both disciplines help programmers understand and estimate the resources necessary to execute programs. Time complexity approximates the time needed for the program execution depending on the problem size, while space complexity models the computer memory usage [50]. The problem size is usually input data size, noted as a dimension of the problem n.

Many researchers have studied these areas from the early days of computer science. Donald Knuth summarized concepts of computational complexity theory and the big O notation [25], first introduced a notation for lower-bounded functions [27], and created the term *analysis of algorithms* [26]. Papadimitriou represented complexity as a number-theoretic concept and examined P and NP problems [39]. Arora and Barak collected a substantial quantity of complexity classes [5]. They also analyzed space complexity and represented the theoretical concept of randomized and quantum computation [5].

Understanding the basic concepts of the computational complexity of algorithms, especially time complexity, is a substantial part of introductory courses on programming, algorithms, and data structures. There are well-known classes of computer programs with certain time complexities used in real-life applications, such as binary search, different sorting methods, tree and graph traversals, finding shortest paths in graphs, and similar. Most universities recognize the necessity of teaching these concepts. For example, Table 1 shows the first five universities on the ARWU (Shanghai) list [51] for the year 2022 and the courses that cover the topics on computational complexity as a part of their curriculum. The selected list sorts these universities by Computer Science and Engineering subject.

However, besides real-life examples, there is frequently a need for synthetic source code examples with defined time complexity. Based on our previous work and efforts [40,44], we propose a novel, template- and rule-based approach to source code assembling of predefined time complexity. The approach is implemented through a software system that generates code segments based on the defined code templates and interaction rules. One of the most important goals of the system is to improve both the teaching process and the quality of student examination. Assembled segments can be used in classes for teaching or given to students at exams. Teachers can choose one of the segments in manual mode or let the software system choose it in automated mode. Usage of the proposed software significantly decreases exam assembly time, as instructors configure it once and then use the assembled segments.

The software can help to prevent cheating with the possibility of generating many segments according to the given criteria. Therefore, each student can get a different question. In addition, the system does not contain a database of questions in its final form, which prevents possible database theft. To secure result correctness, the software calculates and verifies the time complexity of the generated segments.

Table 1. World Universities and their courses which study algorithm time complexity

University	Courses
Massachusetts Institute of Technology (MIT)	Introduction to Computer Science Programming in Python (6.001), Introduction to Algorithms (6.006), Automata, Computability and Complexity (6.045), Design and Analysis of the Algorithms (6.046)
Stanford University	Mathematical Foundations of Computing (CS103), Programming abstractions (CS106B), Introduction to automata and complexity theory (CS154), Design and analysis of algorithms (CS161), Computational complexity (CS254)
University of California, Berkeley	The Structure and Interpretation of Computer Programs (CS61A), Efficient Algorithms and Intractable Problems (CS170), Computability and Complexity (CS172)
Carnegie Mellon University	Great Ideas in Theoretical Computer Science (15251), Algorithms & Advanced Data Structures (15351), Algorithms (15451), Undergraduate complexity theory (15455)
Tsinghua University	Logic and Computation I, Combinatorics and Algorithms Design, Advanced Theoretical Computer Science

Keeping in mind the educational perspective of a large number of students, the necessity of automatizing the exam assembling process, and cheating prevention, the main contributions of the paper are:

- We developed a novel, formal approach for assembling code segments with verified target time complexity.
- We implemented a software system based on our formal approach and made it publicly available and open source [42].
- We evaluated the implemented system and showed that assembled segments resemble hand-crafted and that the tool can significantly decrease exam preparation time.

The paper is organized as follows. We introduce computational complexity analysis emphasizing time complexity analysis in Section 2. We define template- and rule-based proposed system principles, the assembling approach, and the time complexity calculation in Section 3. We present the system architecture in Section 4. We evaluate the system in Section 5. We discuss related work in Section 6. We briefly conclude and summarise directions for future work in Section 7.

2. Theoretical background

We introduce all the necessary definitions for understanding the proposed system's design principles. We also note the undecidability of the symbolic complexity calculation.

2.1. About time complexity calculation

We assume that a problem size depends on only one input parameter n. This assumption is not a limiting factor because the time complexity calculations are usually done in one

dimension [14], [12]. The function $T(n)$, which approximates the program execution time as a function of the problem size n, is defined with several elementary steps. Each step has a constant execution time [14]. Unfortunately, even for small programs, this function becomes very complex. Therefore, it is usually reduced to the function with the same asymptotic behavior for sufficiently large n. For example, the function $T(n) = n^2 + log(n) + n + 2^n$, is reduced to the function 2^n. This is formalized by big O notation, where $O(2^n)$ denotes a class of functions that have 2^n as their upper limit. The big O notation is defined by Definition 1 [12].

Definition 1. *For a given function* $T(n)$*,*

$$T(n) = O(f(n))$$

if there exist $c \in \mathcal{R}$ *and* $n_0 \in \mathcal{N}$ *such that* $c > 0$ *and* $n_0 > 0$ *and for all* $n \in \mathcal{N}$ *such that* $n > n_0$ *it holds*

$$0 \leq T(n) \leq c \cdot f(n)$$

.

Expression $T(n) = O(f(n))$ denotes that function $T(n)$ belongs to the class of functions $O(f(n))$. When this notation is used, the analysis of the complexity of an algorithm is reduced to the analysis of the complexity of the general structure of the algorithm.

Different characteristics can be proved directly from the definition [12]. Two of them, used in our approach for complexity calculation, are based on calculating complexity when two independent code segments are put into the sequence and when one code segment is nested into a loop. In the case of sequence, the resulting complexity is a sum of the two complexity classes. In the case of nesting, there are some cases where the resulting complexity is a product of two complexity classes.

It can be proved that the sum of the two complexity classes is equal to the larger complexity class [12], as shown in Equation 1

$$O(f(n)) + O(g(n)) = O(max(f(n), g(n))) \,. \tag{1}$$

The product of two complexity classes is the complexity class of the product function [12], as shown in Equation 2.

$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n)) \tag{2}$$

Although time complexity function can have arbitrary form, there is a subset of function classes that are commonly used in real-world programming and therefore have exceptional importance in education: constant $O(1)$, logarithmic $O(log(n)$, linear $O(n)$, polynomial $O(n^k), k > 1$, exponential $O(k^n), k > 1$ and factorial $O(n!)$. Equation 3 compares these function classes.

$$O(1) < O(log(n)) < O(n) < O(n^k) < O(k^n) < O(n!) \tag{3}$$

2.2. Halting problem and the complexity calculation

In the general case, the time complexity calculation of an arbitrary source code is an undecidable problem. This is a direct consequence of the undecidability of the halting problem (proved by Alan. M. Turing [56]). The halting problem is the problem of determining for an arbitrary program and an arbitrary input whether the program will finish running or continue to run forever. Since it is not possible to construct an algorithm that can determine if each program stops its execution, then it is also not possible to make an algorithm that would count the number of executed statements, i.e., which would calculate the time complexity of an arbitrary program. The Listing 2.1 shows the example of the Collatz conjecture problem [11]: a program that looks very simple, but its time complexity cannot be calculated.

The Collatz conjecture program generates a sequence of numbers. For the arbitrary number n, there is no mathematical proof that the sequence will eventually reach number 1 and that the loop will terminate. Conway proved that Collatz-type problems are undecidable [11].

```
while(n != 1){
  if(n % 2 == 0){
    n = n / 2;
  } else {
    n = 3 * n + 1;
  }
  print(n);
}
```

Listing 2.1. Pseudo code of the Collatz conjecture problem

2.3. Time complexity calculation approaches in software

A time complexity calculation of a source code is a complex task, undecidable for an arbitrary source code. However, extensive research is done to estimate and predict time complexity or generate a code segment of the given complexity.

Early work in this context was done by McCabe, who proposed the concept of cyclomatic complexity [30]. Cyclomatic complexity is a software metric that measures the number of linearly independent paths in the control flow graph of the given source code. It is then used to estimate the complexity of the program. However, it does not quantify the number of passes through the paths in the execution context of the program. Hence, it cannot be used directly to estimate time complexity.

The Master theorem for divide-and-conquer problems was given in [7]. It provides an asymptotic analysis with big O notation for recurrence relations presented in typical divide-and-conquer algorithms. In general, the time complexity of the algorithm that takes the problem of the size n and partitions it in partitions each sized n/b can be calculated with Equation 4:

$$T(n) = a \cdot T(n/b) + f(n)\,. \tag{4}$$

where a is the number of partitions and $f(n)$ is the time to create partitions and combine their results. The time complexity of well-known algorithms such as binary search of a sorted array, binary tree traversal, and merge sort can be calculated with this approach. However, the approach is limited only to divide-and-conquer problems.

In the recent period, there were attempts to use machine learning methods to estimate the time complexity of the code. Those methods operate on hand-crafted features or appropriate code embeddings, which are used to learn the internal structure of the code. Code representations such as *code2vec* [4] are used in this context in [53] to predict the time complexity class of a given program written in Java. For training purposes, they collected annotated dataset CoRCoD: Code Runtime Complexity Dataset [32][53], extracted from online judges, with 932 source code examples with five different classes of complexities: $O(1)$, $O(log(n)$, $O(n)$, $O(n \cdot logn)$, $O(n^2)$. The results showed that code embeddings have comparable performances to hand-crafted features for classification using Support Vector Machines, which showed the best accuracy, precision, and recall on the test set. However, the accuracy was at most around 70% and varied across classes significantly.

New research on time and space complexity prediction can be found in [52],[34]. Siddiq et al. used a GPT3-based code generation tool *GitHub Copilot* to predict the runtime complexity of a given source code using zero-shot prompting [52]. The obtained accuracy ranges from 45.44% to 56.38%, depending on the suggestions given to the system. A transformer-based approach to space and time complexity is considered in [34]. The authors state that existing time complexity calculation libraries and tools only apply to limited use cases because of the lack of a well-defined rule-based system. They proposed using code-based language models, such as BERT, CodeBERT, GraphCodeBERT, CodeT5, and Longformer. They achieved prediction accuracy ranging from 72.55% for C++ codes to 92.08% for Java codes, depending on the model used.

Another machine learning approach was conducted in [36] using a dataset based on exam questions from the Programming 1 course at the University of Belgrade, School of Electrical Engineering. The dataset contained 61 exam questions and solutions, each representing a code segment written in Pascal and its time complexity. Features were crafted from the source code using three models: count vectorizer, term frequency-inverse document frequency, and embedding layers in neural networks. The results showed that classical machine learning models, such as random forests, perform better than more advanced models based on neural networks. However, the model's accuracy was not high due to the relatively small annotated data set used in training.

The authors of this paper adopted another approach to time complexity calculation. Through several papers [44], [45], [40], [41], we developed a set of methods for source code assembly of a given time complexity. In the first approach, a set of basic, handcrafted, and verified complexity segments was used to produce new segments with code parametrization and obfuscation based on simple rules given in Equation 1 and Equation 2 [44]. However, only basic complexity classes and combining techniques were supported. Further approach [45] introduced a more abstract assembly strategy modeled with XML and abstract syntax trees. The paper [40] was oriented toward educational and technical aspects of the system. These aspects are further elaborated in the extended version of the conference paper [41], where we introduce a formal approach to the segment assembling process with a prototype implementation described with the subset of C grammar. The system is based on templates and rules together with the time complexity calculation approach done by symbolic calculation using the Yacas tool. This significantly broadens the scope of assembled code segments and their complexity. Although promising, symbolic calculation has limitations due to its high time complexity and non-applicability for some

complexity classes. Preliminary analysis and assessment of the system for teaching and evaluation were given in [43]. The results suggest that the system produces code segments with similar semantics but different textual representations.

3. Segment assembling process and time complexity calculation

This section presents an algorithm for source code assembling and all the necessary concepts. The algorithm merges two code segments (templates) with known complexities using a chosen rule, which lets us calculate and guarantee the exact complexity of the resulting segment. The resulting segment can be used in further segment assembling. This enables rule cascading and complex segment assembling.

3.1. Basic building blocks, templates, and rules

A template is an arbitrary source code segment, assembled of building blocks, recognized from imperative programming. Building blocks are assignment statements and simple *while* loops. *If* statements are generated in the assembling process, as will be described. *For* and *do-while* loops are generated by transforming *while* loops at the end of the assembly process (not further discussed).

Figure 1 shows different template examples, denoted with T1 to T10. Templates T1, T3, and T5 are examples of assignment statements, and they are predefined. Templates T2 and T9 are examples of simple loops. Templates T2, T4, and T6 to T10 are products of the assembling process.

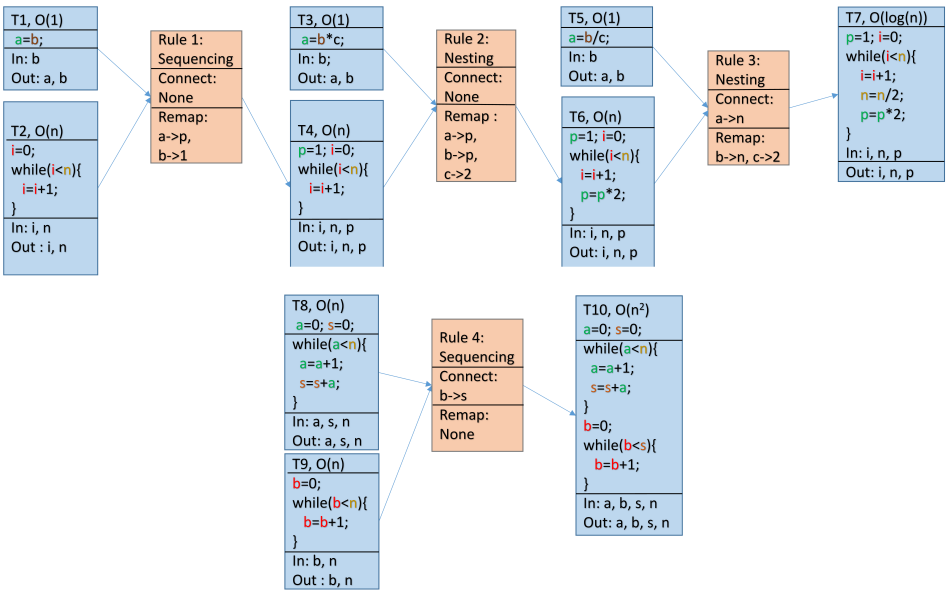

Fig. 1. Assembling process with different templates and rules

We support assignment statements that include expressions with addition, subtraction, multiplication, and division. We also support different kinds of simple loops. The structure of a simple loop is described with grammar. In the grammar, we use the symbols presented in Table 2 and abbreviations given in Listing 3.1. We also introduce the operator $*$ (star) to shorten the number of grammar rules (without information loss). The operator $*$ takes one permutation from the set of all permutations of the sequence of the terminal or non-terminal grammar symbols. Symbols presented in Table 2 are non-terminal symbols. They are mapped to variables and constants used in particular source code segments. For example, in Figure 1 variable i in template T1 is a specialization of symbol a_0, and constant 1 is a specialization of symbol z_1.

Table 2. Symbols used in the grammar

Symbols	Description
a_0, a_1	Variables holding lower loop bounds
b_0, b_1	Variables holding upper loop bounds
s_0	Variable holding a sum
v_0, v_1, ..., v_6	Auxiliary variables
c_0, c_1, c_2	Numerical constants greater than zero
z_0, z_1	Numerical constants greater than zero ordered such that $z_0 < z_1$
o_0, o_1	Numerical constants greater than one ordered such that $o_0 < o_1$

```
init(lower,upper) := (lower=z0; upper=n;)*
init(lower,upper,aux0) := (lower=z0; upper=o0; aux0=n;)*
body(lower,upper,aux0,aux1,aux2) := body1(lower,upper) |
                                    body2(lower,upper,aux0,aux1) |
                                    body3(lower,upper) |
                                    body4(lower,aux0,aux2)
body1(lower,upper) := (lower=lower+z1; [upper=upper+z0;])* |
                      (upper=upper-z1; [lower=lower-z0;])* |
                      (lower=lower+c0 | upper=upper-c1)
body2(lower,upper,aux0,aux1) := (lower=aux0*aux0;
                                 upper=aux1*aux1;
                                 aux0=aux0+z0;
                                 aux1=aux1-z0;)*
body3(lower,upper) := (lower=lower*o1;
                [upper=upper-z0; | upper=lower+z0; | upper=upper*o0;])*
                | (upper=upper/o1;
                [lower=lower-z0; | lower=lower+z0; | lower=lower/o0])*
body4(lower,aux0,aux1)  :=  (lower=aux0*aux0; aux0=aux0+z0;)* |
                            (lower=lower+aux1; aux1=aux1+z0;)*
```

Listing 3.1. The abbreviations for initializations of variables and definitions of bodies that are used in the grammar

A description of a supported structure of a simple loop segment is given in Listing 3.2. The simple loop segment contains initialization statements, one *while* loop, and a loop body. We initialize only those variables that are used in the loop body. The body is a sequence of statements. We define four different body types, denoted with $body_1$, $body_2$, $body_3$, and $body_4$ (Listing 3.1). Each type contains its corresponding statements listed in

the table. Simple loops with $body_1$ and $body_2$ have complexity $O(n)$, loops with $body_3$ have complexity $O(log(n))$ and loops with $body_4$ have complexity $O(sqrt(n))$.

```
simple_loop :=  init(a0,b0)
                init(v0,v1,v2)
                while(a0<b0){
                    body(a0,b0,v0,v1,v2)
                }
```

Listing 3.2. Grammar rule that describes a simple loop segment

Two source code segments are data dependent if the first segment produces a result used in the second segment and if that result impacts the time complexity of the combination of these two segments. Otherwise, these two source code segments are mutually independent. Figure 1 also shows an example of the data dependency where two segments are put into a sequence. Template T8, with $O(n)$ complexity, produces a value for variable s. After the loop, the value stored in s is a sum of the first n numbers. The variable s is forwarded as an upper loop bound of the T9. Therefore, the time complexity of the sequence is $O(n^2)$.

Each template has ports defined by the system. Ports describe data entry points (input ports) and exit points (output ports). A port is a variable that satisfies one of the following conditions: the variable value is gathered from the other template (input port), or the variable value is forwarded to the other template (output port). Template T1 from Figure 1 has $O(1)$ complexity, input port b, and output ports a and b. The output port of the template can be connected to the input port on another template. Port connections represent data flow between templates and exist only if that data flow impacts the resulting complexity calculation. This happens when templates have data dependencies. Port connections are used to mark particular data dependencies.

During the assembling, the variables inside one template can be renamed or specialized to constants. In this case, renaming increases the possibility of template reuse inside different rules. Renaming is also used in the creation of port connections between two templates. In this case, all appearances of the output port names are replaced with the input port names.

An operation is a template merging method. It can be *sequencing*, *nesting*, and *selecting*. Sequencing operation puts one template after another. The nesting operation puts one template inside another template. Selecting operation puts one template into the *then* branch of an *if* statement or two templates into the *then* and *else* branches of the *if* statement. *If* statement additionally has a selection condition. We use conditions that do not trivially evaluate to *true* or *false*. Therefore, the code in existing branches must be considered when calculating time complexity.

The rule collects all the information needed to assemble a new segment. This information contains used templates, the data dependencies between them (denoted as port connections), and the operation used for template merging. Also, the rule defines variable renaming. Figure 1 contains three rule examples. Rule 1 puts templates T1 and T2 into the sequence, renames variable a to p, specializes variable b to constant 1, and produces template T4. This combining does not have data dependencies (and therefore does not use port connections). Rule 3 nests template T5 in T6. In this case, data dependency between T5 and T6 is created (and therefore port a is connected to port n), variable b is renamed to n, and variable c is specialized to constant 2. The assembled segment is a temporary

result. To be accepted as a result, its time complexity must be calculated. The following section describes the time complexity calculation.

3.2. Time complexity calculation

Time complexity calculation differs in two cases: when assembling is done using two templates with and without data dependencies between them. The following sections describe the time complexity calculation in both cases.

Assembling without data dependencies If data dependencies are not present, the time complexity calculation is straightforward. For the sequencing and selecting operations, the resulting complexity is a sum of the two complexity classes. In the case of nesting, the resulting complexity is a product of two complexity classes. Therefore, starting with simple loops and assignment statements, we can assemble segments with the complexity classes described with Equation 5.

$$O(n^i \cdot log^j(n) \cdot sqrt(n)^k), \text{where } i, j, k \geq 0\,. \tag{5}$$

Additionally, complexity classes calculated with data dependencies can be mutually multiplied or multiplied with complexity classes described with Equation 5. All these possibilities provide significant coverage of important complexity classes (denoted in Section 2.1) and other non-trivial complexity classes.

If the assignment statements or the if statements (with constant complexity) are nested inside the simple loop and data dependencies are not present, the result is an enhanced loop segment. Templates T6 and T8 (Figure 1) are examples of enhanced loops.

Assembling with data dependencies If a segment is assembled from two templates with data dependencies between them, time complexity calculation is challenging, and we introduce some constraints to make the calculation possible. When data dependencies are present, the algorithm for time complexity calculation requires a defined segment structure such that the time complexity can be calculated analytically. This is not a limiting factor because support for more data dependencies can be easily added when necessary. We support the following three cases of template assembling with data dependencies:

1. One or more assignment statements are nested inside one empty *while* loop. This case is used for assembling the simple loop segments when the system is started. Empty *while* loop and assignment statements are predefined.
2. One or more assignment statements are nested inside one simple loop.
3. Two enhanced loops are sequenced or nested.

In the first two cases, the newly created loop body must comply with one of the four supported simple loop bodies (Listing 3.1). This implies that the assembled loop is a simple loop segment, and its time complexity can be determined, as described in Section 3.1. Figure 1 illustrates this situation. Template T5, having the O(1) complexity, is nested into template T6, width $O(n)$ complexity. Template T5 changes the loop bound of the T6. Consequently, the resulting segment has $O(log(n))$ complexity. The third case is described in the following paragraphs.

Sequencing with data dependencies Listing 3.3 describes the supported structure of the sequence of the two enhanced loop segments.

```
loop_sequencing   : =
    (init(a0,b0) init(v0,v1,v2) (s0=0;|s0=1;) v3=c0;)*
    while(a0<b0){
      (body(a0,b0,v0,v1,v2)
      (s0=s0+o0*v3;|s0=s0*o0*v3;|s0=s0+z0;|s0=s0*o0;))*
    }
    (init(a1,b1) init(v4,v5,v6) b1=s0;)*
    while(a1<b1){
      body(a1,b1,v4,v5,v6)
    }
```

Listing 3.3. Grammar rule that describes two enhanced loop segments put into a sequence

The left-side variable from that statement (marked as s_0) is used as an upper loop bound in the second segment, which creates a data dependency. Let the $O(f(n))$ be the complexity of the first segment in sequence and $O(g(n))$ be the complexity of the second segment in sequence. The resulting complexity calculation is given in Table 3. There are four ways to calculate the overall complexity, depending on the chosen statement.

Table 3. Asymptotic value of the variable s_0 after the first segment is executed and the resulting complexity of the sequence of two enhanced loops

Statement	Variable s_0	Resulting complexity
$s_0 = s_0 + o_0 * v_3$	$\approx f(n^2)$	$max(O(f(n), O(g(f(n^2)))))$
$s_0 = s_0 * o_0 * v_3$	$\approx f(n!)$	$max(O(f(n), O(g(f(n!)))))$
$s_0 = s_0 + z_0$	$\approx f(n)$	$max(O(f(n), O(g(f(n)))))$
$s_0 = s_0 * o_0$	$\approx f(o^n)$	$max(O(f(n), O(g(f(o^n)))))$

Nesting with data dependencies We describe three supported cases when two enhanced loop segments are nested, and data dependency is present between them. These three cases lead to three different ways of resulting complexity calculation. The data dependencies are created with the assignment statements placed inside the outer loop and before the inner loop. These statements modify the bounds of the inner loop in a controlled manner. Grammar in Listing 3.4 describes these three cases.

Let the $O(f(n))$ be the outer loop complexity and $O(g(n))$ be the inner loop complexity. In the first case, inner loop bounds are constants, which implies that the inner loop does not impact the complexity calculation anymore, and the resulting complexity is $O(f(n))$.

In the second case, we use multiplication and division to modify the inner loop bounds, and the resulting complexity is described with Equation 6.

$$\sum_{i=1}^{f(n)} g(o_0^i) \tag{6}$$

In the third case, we use addition and subtraction for the same purpose. The resulting complexity calculation for this case is described with Equation 7.

$$\sum_{i=1}^{f(n)} g(i \cdot z_0) \tag{7}$$

These formulas cannot be analytically calculated for the arbitrary functions $f(n)$ and $g(n)$. Therefore, we use the following functions for $f(n)$ and $g(n)$: n, $log(n)$ and $sqrt(n)$. These functions are supported complexity classes of used enhanced simple loop segments (as described in Section 3.1). The resulting complexity is given in Table 4 and Table 5.

```
loop_nesting  :=  loop_nesting1 | loop_nesting2 | loop_nesting3
loop_nesting1 :=  init(a0,b0) init(a1,b1) init(v0,v1,v2) v6=c0
        while(a0<b0){
          body(a0,b0,v0,v1,v2)
          init(v3,v4,v5)
          ((a1=z1; b1=z0;) | (a1=v6; b1=v6;))
          while(a1<b1){
            body(a1,b1,v3,v4,v5)
          }
        }
loop_nesting2 :=  init(a0,b0) init(a1,b1) init(v0,v1,v2)
        while(a0<b0){
          body(a0,b0,v0,v1,v2) init(v3,v4,v5)
          ((a1=a1*o0; b1=n;) | (b1=b1/o0; a1=c1;))*
          while(a1<b1){
            body(a1,b1,v3,v4,v5)
          }
        }
loop_nesting3 :=  init(a0,b0) init(a1,b1) init(v0,v1,v2)
        while(a0<b0){
          body(a0,b0,v0,v1,v2)
          init(v3,v4,v5)
          ((a1=a1+z0; b1=n;) | (b1=b1-z0; a1=c2;))*
          while(a1<b1){
            body(a1,b1,v3,v4,v5)
          }
        }
```

Listing 3.4. Grammar rules that describe two nested enhanced loops

Table 4. Lookup table for the complexity calculation by Equation 6

f(n) / *g(n)*	n	$log(n)$	$sqrt(n)$
n	o^n	n	$o^{sqrt(n)}$
$log(n)$	n^2	$log(n)^2$	n
$sqrt(n)$	$sqrt(o^n)$	$sqrt(n)$	$sqrt(o^{sqrt(n)})$

Table 5. Lookup table for the complexity calculation by Equation 7

f(n) / *g(n)*	n	$log(n)$	$sqrt(n)$
n	n^2	$log(n)^2$	n
$log(n)$	n	$log(n) \cdot log(log(n))$	$sqrt(n) \cdot log(sqrt(n))$

4. Implementation details

We present the architecture of the proposed system for source code assembling based on the defined templates and rules. The system can produce many resulting code segments from a limited set containing initial templates and rules. This is important for the simplicity of usage and for controlling the correctness of each output.

4.1. System overview

The system generates various source code segments using the set of available segments and combining operations. It has a modular structure. Figure 2 presents system modules. These modules are the graphical user interface (GUI), template assembling module, rule assembling module, template database, rule database, rule execution, complexity calculation module, result segment processor, and segment-assembling manager.

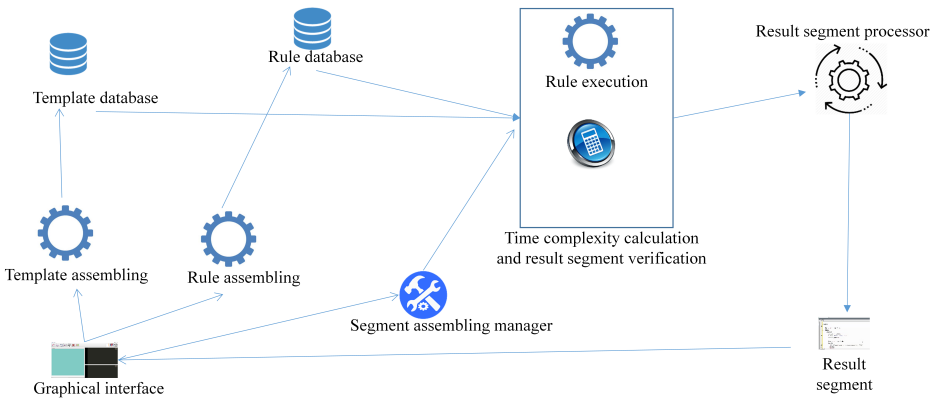

Fig. 2. System overview

Rule and template assembling modules are directly connected to the databases and the GUI. The template assembling module collects information from the GUI and then builds the template intermediate representation, which is further downloaded to the database. Similarly, the rule assembling module gets information from the GUI, creates the rule representation, and saves it within the corresponding database.

4.2. Graphical user interface

Graphical user interface is introduced in [40], and has been constantly improved. It serves for creating custom user templates. These templates can be used to enrich the template database or to test the system. Figure 3 displays a template assembling GUI after starting the tool.

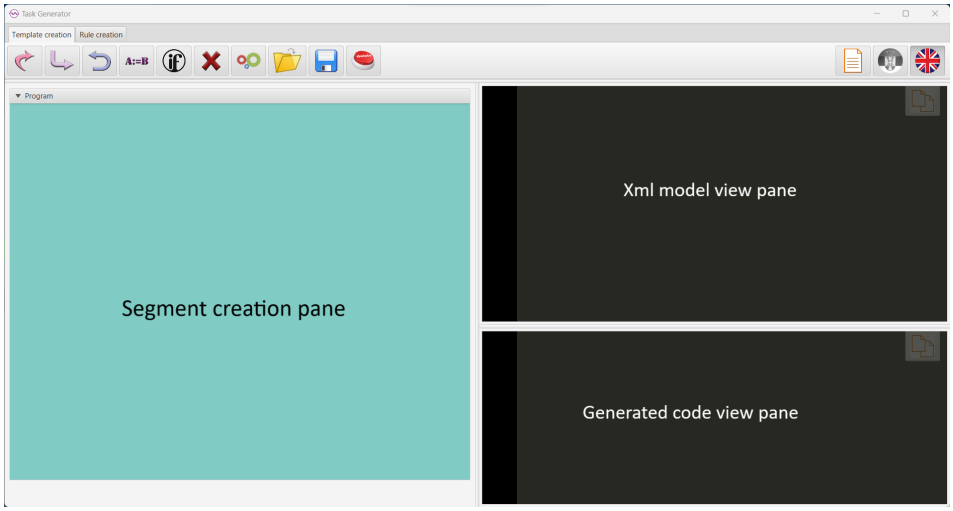

Fig. 3. GUI for template assembling

The template assembling GUI has the following parts: the toolbar, segment creating pane, XML model view pane, and generated code view pane. The toolbar contains commands for adding segment building blocks (loops, expressions, and selections), deleting segments, generating code from graphical representation, saving to a file and copying the generated code, and selecting an interface language (English and Serbian are currently supported). The click on the creation pane adds or removes a segment, and depending on the position of the click, we can achieve nesting or sequencing. The XML model view pane displays the intermediate implementation of the generated segment, while the generated code view pane displays the final source code. Rule assembling GUI is implemented similarly. It has a similar role as the template assembling GUI and enables creating custom rules and testing the system.

4.3. Rule execution

The rule execution module is the core of the system. Figure 4 presents the steps of the rule execution process.

Two operands used during the rule execution process can be templates, rules, or their combination. If an operand is a template, it is forwarded to the next step. If the operand is a rule, it is executed, and the output is delivered to the next step. Operation execution

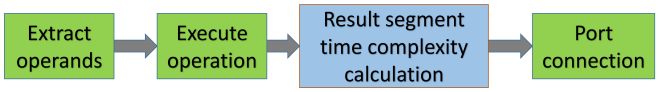

Fig. 4. Rule execution diagram

is straightforward. The *nesting* operation nests the second operand into the first. The *sequencing* operation creates a sequence where the second operand follows the first. The *selecting* operation puts the first operand into the *then* branch and the second operand into the *else* branch. *Else* branch may not exist.

The most complex step is the time complexity calculation of the resulting segment, verified according to the theory and grammar described in Section 3. The last step is establishing data dependencies using port connection, when needed. The rule definition contains mappings between the first segment's output ports and the second segment's input ports. These mappings are executed via variable renaming.

The output of rule execution is an intermediate program segment representation, which can be stored in the template database and used as an input to another rule. The result segment processor takes the intermediate segment and converts it to the source code of the target language. Currently, we support C and C++ programming languages, but other programming languages can be easily added to the system. During the processing, the resulting segment is obfuscated [44]. Obfuscation changes the resulting source code, without changing its time complexity, by modifying variable names not included in data dependencies. As a result, the number of generated segments is additionally increased.

The segment assembling manager module is responsible for implementing higher-level assembling strategies, which use the current template and rule mechanism to generate many segments. There are two envisioned scenarios for higher-level assembling strategies. The first is to assemble a set of segments of some desired (target) time complexity. The other one is different: starting from the seed set of basic templates and rules, the system should assemble a set of segments and let the system calculate their time complexity. A large number of segments can be further used in online examinations and CAT (computer-aided testing).

5. Evaluation and discussion

We evaluated our proposed solution in several contexts. First, similarly to some recent studies from the open literature, we used a machine learning approach to verify code segment complexity. Second, we presented our experiences of using assembled segments in exams and discussed them with the help of field experts in the form of an interview on the topic.

5.1. Evaluated code segments

To evaluate briefly and concisely, we focused on the four most significant complexity classes: $O(log(n))$, linear $O(n)$, linear logarithmic $O(n \cdot log(n))$, and quadratic $O(n^2)$. We chose two representative code segments for each class: one hand-crafted (Listings 5.1,

5.3, 5.5, and 5.7), typically given as an exam question or presented in lectures, and one assembled with our system (Listings 5.2, 5.4, 5.6, and 5.8).

It can be seen that some of the hand-crafted code segments represent well-known algorithms, such as binary search and insertion sort, which are often taught as examples in programming and algorithms and data structure courses. Those examples are shown in Listing 5.1 and Listing 5.7. The other two hand-crafted code segments do not represent any particular algorithm but were assembled with target complexity in mind. In Listing 5.3, it can be noted that a "dead" loop has been inserted into the code with no iterations to confuse the less knowledgeable students.

```
d = 0, g = n-1;
while (d <= g) {
  s = (d + g) / 2;
  if (arr[s] == k)
    break;
  else if (k < arr[s])
    g = s - 1;
  else
    l = s + 1;
}
```

Listing 5.1. Hand-crafted code segment with time complexity $O(log(n))$ which represent binary search algorithm

```
f = 0;
a = 1;
b = n;
while (a<b)
{
  a = a*2;
  f = f+1;
}
d = f;
c = 1;
while (c<d)
{
  d = d-1;
  c = c*2;
}
```

Listing 5.2. Assembled code segment with time complexity $O(log(n))$

Assembled segments with our tool follow the rules presented in Section 3.2. The segments are slightly longer than those hand-crafted, mainly because the processor uses more intermediate variables to assure correctness. Compilers-based techniques (such as constant folding, constant propagation, strength reduction, copy propagation, common subexpression elimination, and arithmetic simplification [1]) could further simplify the code.

```
i = s = 0;
while (i < n) {
  i += 1;
  j = m = 0;
  for (; j <= m; ++j) {
    m *= i;
      s++;
  }
  s--;
}
```

Listing 5.3. Hand-crafted code segment with time complexity $O(n)$. Note the dead inner loop

```
a = 1;
b = n;
do
{
  a = a+4;
}
while (a<b);
a = 1;
b = n;
while (a<b)
{
  b = b-24;
}
```

Listing 5.4. Assembled code segment with time complexity $O(n)$

```
s = 0;
for (i = 0; i < n; i++) {
  j = 1;
  while (j < i) {
    s += i * j;
    j *= 2;
  }
}
```

Listing 5.5. Hand-crafted code segment with time complexity $O(n \cdot log(n))$

```
a = 1;
b = n;
do
{
  b = b/2;
  c = 1;
  d = n;
  do
  {
    d = d-1;
  }
  while (c<d);
}
while (a<b);
```

Listing 5.6. Assembled code segment with time complexity $O(n \cdot log(n))$

```
for (int i = 1; i < n; ++i ) {
  int k = a[i], j = i - 1;
  while ( ( j > 0 ) && a[j] > k ) {
    a[j + 1] = a[j];
    j = j - 1;
  }
  a[j + 1] = k;
}
```

Listing 5.7. Hand-crafted code segment with time complexity $O(n^2)$ which represents insertion sort algorithm

```
a = 1;
b = n;
while (a<b)
{
  d = n;
  c = 1;
  while (c<d)
  {
    c = c+13;
    d = d-64;
  }
  b = b-24;
}
```

Listing 5.8. Assembled code segment with time complexity $O(n^2)$

5.2. ChatGPT-based evaluation

We used ChatGPT [38], a language model-based chatbot, to assess several hand-crafted and assembled code segments. Although the validity of such an evaluation in different areas is widely discussed [47], [2], several recent studies [55], [9], [29] suggest the ability of AI tools to provide explanations, examples, and guidance on complex programming tasks. In addition, machine learning techniques can be used for time and space complexity prediction [53], [34], [52].

We asked ChatGPT to answer the following two questions for each of the code segments given in Section 5.1:

- What is the time complexity of the following piece of code?
- Is this piece of code written by humans or generated by an algorithm?

The snapshot of the whole conversation with ChatGPT can be found at the following link, but also on the GitHub project [42]. Conclusions drawn from the ChatGPT experiment are given in the following paragraphs.

Regarding the first question, ChatGPT successfully calculated the time complexity for all given segments, except for the hand-crafted code segment from Listing 5.3 and the computer-assembled code segment from Listing 5.8, where it firstly gave wrong answers. However, with additional inputs and hints, ChatGPT eventually gave correct answers. As mentioned, the code from Listing 5.3 has a "trick" in the form of a dead loop, which confused the chatbot. The piece of code given in Listing 5.8 indirectly uses the dimension of the problem n through intermediate variables b and d. It confuses ChatGPT, as it could also deceive an inattentive human reader.

Based on the answers to the second question, we conclude that ChatGPT does not imply that any code segment was machine-assembled. A typical answer resembles: *"This code appears to be something a human might write. It follows typical programming patterns using constructs like loops and variable manipulation. There are no obvious signs of unusual patterns or styles that would suggest an algorithm generated it"*. It shows that from this perspective, we managed to imitate the humans in the way they assemble similar code segments. ChatGPT also recognized well-known binary search and insertion sort algorithms as human-written.

5.3. Real-life experiences

Instead of the hand-crafted counterparts, code segments generated by our system have been occasionally used as exam questions in the Programming 2 course at the University of Belgrade, School of Electrical Engineering, for the past three years. Typically, they were used two or three times out of five exam periods in which the exam can be taken. There was no observable change in students' success on these questions. Therefore, students find code segments generated by our system equally difficult to solve.

To assess our solution from an instructor's perspective, we organized a focus group of four field experts engaged in courses that deal with code complexity calculation. We described our approach and presented the participants with the same code segments given in Section 5.1. We asked them for comments on the following general questions:

- How much time is needed to assemble a code segment of a target complexity as an exam question?
- How useful is the presented solution in the context of preparation of exam questions?
- For the presented pieces of code, to distinguish whether they were written by humans or generated by an algorithm?

The participants agreed that the effort needed to assemble a code segment as an exam question greatly depends on its time complexity and the desired structure of the code. On average, it varies from 15 to 30 minutes, together with final verification. Taking a code segment from some predefined repository or a book significantly reduces that time, as only final verification and some code obfuscation are needed. In that context, a presented software solution can be beneficial to produce a large code base and avoid question repeating. They also pointed out that the offered approach guarantees target complexity, which reduces the time needed for verification and guarantees correctness.

The participants were more successful in determining whether the code segment was machine-assembled or hand-crafted. They were correct for all hand-crafted cases and three out of four machine-assembled cases. However, they mostly noted that this is due to the unusual naming conventions for the variables used in assembled segments and the generally excessive use of variables, which can be corrected in the further developments of the system.

6. Related work

As the most fundamental concepts of programming, analysis of algorithms and computational complexity have been studied from the early days of computer science. On the

other hand, recently, automated test assembling and evaluation have attracted significant research interest because introductory programming courses have become massive. We provide information on the related research and compare it with our approach.

Algorithm time complexity is a topic of various studies. There are two different approaches: empirical (profiling) and analytical. In the first approach, a program is instrumented and executed, possibly several times and with various inputs. Instrumentation is achieved by inserting additional instructions into the source code or binary file, and these instructions observe program behavior during execution. *Valgrind* is a tool that does extensive dynamic binary instrumentation [35]. *Gprof* is a tool that does a combination of compile-time instrumentation and runtime sampling [20]. Based on the collected empirical data, time complexity can be estimated. Further, profiling can help with program optimization. McGeoch, Precup, and Cohen developed a heuristic algorithm for finding upper bounds among datasets [31]. A dataset is represented by the two vectors X and Y, such that $Y = f(X)$, where f is an unknown function. The problem is to analyze X and Y and estimate complexity bound $O(f(X))$.

S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson [19] developed a profiling software called *trend-prof*. It constructs empirical computational complexity models. The authors showed that for real-world programs, linear and exponential complexity models are enough for roughly modeling execution time. We decided to use a formal approach because empirical complexity calculation involves compiling, linking, and executing the generated segment numerous times with extensive input data. In addition, our software should generate many segments, which would take too long if empirical complexity calculation is used.

In the analytical approach, the time complexity of the program is analyzed statically, i.e., without executing the program. This approach might give more precise results than profiling but is generally undecidable as it requires calculating loop summaries symbolically. J. Gustafson, A. Ermedahl, C. Sandberg, and B. Lisper [22] presented a method for deriving a static worst-case execution time using symbolic execution. To estimate the time, they estimate the upper bound on the number of loop iterations. The estimated time is a number, while our work uses symbolical values for complexity functions. Research on symbolic execution [54] is based on the computation of loop summaries for loops along acyclic paths leading to the target location. It demonstrates that the usage of the symbolic execution can lead to a very complex calculation that requires SMT [6] solvers usage. S. Gulwani, K. K. Mehra, and T. Chilimbi [21] developed a technique for computing symbolic bounds on the number of statements as a function of scalar input and user-defined functions that describe input data structures. They add counters to the source code, and the modified code is symbolically executed and generates symbolic bounds. However, their algorithm may not calculate the result in some cases. The mentioned approaches use extensive symbolic computation, where entire programs are subject to execution. We use a symbolic execution-based technique only when we combine two program segments with data dependencies. Instead of using SMT solvers, we constrained our software to the cases when symbolic calculation can be done analytically, and then precalculated results are incorporated into the software as lookup tables.

There are various approaches to automated test assembling and evaluation. One approach is to choose the questions from the database considering various parameters, such as the examination area, question difficulty, the appearance of the previous exams, etc.

That kind of software system often uses artificial intelligence, logic programming, and genetic algorithms [10], [23], [59]. Another approach is the question parameterization [16], [17], [18]. If the questions are used for programming exams, parameterization can be done with the change of the input data, which results in a change in the output. Parameterized parts can be marked or chosen randomly. Our software also uses question parameterization during the assembling process.

The idea for the source code segments assembling process is based on A. W. Bierman's work [8]. He presented program segments as a sequence of abstract expressions derived from the rules applied to the specific source code. Our approach is the reverse: starting from the set of defined templates and rules, templates are combined with the rules, and as a result, the segments are assembled.

The evaluation can be automated when the examination is done using pencil-and-paper tests or online. In the case of pencil-and-paper tests, answers are written on particular paper forms, which are then scanned and further processed by specialized software [33]. Online examination is usually performed by popular learning management systems such as Blackboard, Canvas, and Moodle [15]. Different techniques are used for automated evaluation [3], [24]. New approaches combine several techniques [57]. For example, M. V. Janicic, M. Nikolic, D. Tosic, and V. Kuncak developed the software tool for automated grading of students' assignments, using testing, software verification, and control flow graph similarity measurement [58]. Our system automatically generates correct answers. Therefore, it can evaluate generated questions.

The difficulty levels of generated questions are similar, although the questions look different to the students. In our earlier study [43], we have shown that there is a noticeable difference between textual similarity of the generated segments that was additionally verified using the JPlag tool [46] for source code similarity detection. On the other hand, the semantic similarity of the generated segments was high.

The cheating problem can be present no matter how exams are organized. With the help of software tools, teachers can prevent cheating in different ways. One way is to check if the solutions offered by students are a product of cheating. This check can be done with various tools for software comparison [49], [46], which detect similar solutions. Another way is trying to prevent cheating before the examination even starts. This prevention can be achieved by assembling the exam questions such that each student is given a different set of questions with similar characteristics. This is especially adequate for computer-aided tests but can also be used for paper tests. Our approach supports this solution.

Finally, the approach to source code assembly with defined time complexity presented in this study has several limitations. Although a broad scope of segments of different time complexities can be produced, they are still limited to a closed set based on the defined grammar. However, all practically important examples of time complexities found in the most important scholarly books from the field [12], [14], [39], [25] can be produced. The only notable exception is the support for code segments that contain recursion, which is not supported by our current approach and will be the topic for future research.

7. Conclusion

This paper presents a novel approach to source code assembly for defined time complexity. A formal, grammar-based method models a broad class of code segments with different time complexities. Code templates and rules guide the assembly process. In contrast with machine learning approaches, our approach guarantees the time complexity correctness of the generated code segment.

The implemented system can be used both as an educational tool during the teaching process and as a helper tool for the automated assembly of exam questions. The assembling algorithm makes it possible to follow the assembling process so students can easily understand the presented time complexity.

There are several directions for future work. The core of the system can be further expanded to include additional, rarely used classes of time complexities that are not present in the system. Produced segments can be assigned weights based on defined metrics, which would facilitate the final choice of segment by the instructor.

From our experience, students find time complexity calculation difficult. In that sense, any effort that can help students learn it step by step can be beneficial. One of the apparent directions for future work is to make our tool suitable for educational purposes, which follows the recommendations for new approaches in learning code complexity analysis [28] and a general call for innovative technologies in education [37].

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA (2006)
2. Aiyappa, R., An, J., Kwak, H., Ahn, Y.Y.: Can we trust the evaluation on chatgpt? arXiv preprint arXiv:2303.12767 (2023)
3. Ala-Mutka, K.M.: A survey of automated assessment approaches for programming assignments. Computer science education 15(2), 83–102 (2005), https://doi.org/10.1080/08993400500150747
4. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages 3(POPL), 1–29 (2019), https://doi.org/10.1145/3290353
5. Arora, S., Barak, B.: Computational complexity: a modern approach. Cambridge University Press (2009)
6. Barrett, C., Tinelli, C.: Satisfiability modulo theories, Handbook of Model Checking. Springer, Cham (2018)
7. Bentley, J.L., Haken, D., Saxe, J.B.: A general method for solving divide-and-conquer recurrences. ACM SIGACT News 12(3), 36–44 (1980), https://doi.org/10.1145/1008861.1008865
8. Biermann, A.W.: A simple methodology for studying program time complexity. Computer Science Education 1(4), 281–292 (1990), https://doi.org/10.1080/0899340900010402
9. Biswas, S.: Role of chatgpt in computer programming. Mesopotamian Journal of Computer Science 2023, 8–16 (2023)
10. Bošnjaković, A., Protic, J., Bojić, D., Tartalja, I.: Automating the knowledge assessment workflow for large student groups: A development experience. The International journal of engineering education 31(4), 1058–1070 (2015)

11. Conway, J.: Unpredictable iterations. The Ultimate Challenge: The 3x+ 1 Problem pp. 49–52 (1972)
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2022)
13. Dagienė, V., Jevsikova, T., Stupurienė, G., Juškevičienė, A.: Teaching computational thinking in primary schools: Worldwide trends and teachers' attitudes. Computer Science and Information Systems 19(1), 1–24 (2022)
14. Dasgupta, S., Papadimitriou, C.H., Vazirani, U.: Algorithms. McGraw-Hill Education (2006)
15. Draskovic, D., Misic, M., Stanisavljevic, Z.: Transition from traditional to lms supported examining: A case study in computer engineering. Computer Applications in Engineering Education 24(5), 775–786 (2016), https://doi.org/10.1002/cae.21750
16. Geerlings, H., van der Linden, W.J., Glas, C.A.: Optimal test design with rule-based item generation. Applied Psychological Measurement 37(2), 140–161 (2013), https://doi.org/10.1177/0146621612468313
17. Glas, C.A., van der Linden, W.J.: Computerized adaptive testing with item cloning. Applied Psychological Measurement 27(4), 247–261 (2003), https://doi.org/10.1177/0146621603027004001
18. Glas, C.A., van der Linden, W.J., Geerlings, H.: Estimation of the parameters in an item-cloning model for adaptive testing. In: Elements of adaptive testing, pp. 289–314. Springer (2009)
19. Goldsmith, S.F., Aiken, A.S., Wilkerson, D.S.: Measuring empirical computational complexity. In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. pp. 395–404 (2007), https://doi.org/10.1145/1287624.1287681
20. Graham, S.L., Kessler, P.B., McKusick, M.K.: Gprof: A call graph execution profiler. ACM Sigplan Notices 17(6), 120–126 (1982), https://doi.org/10.1145/872726.806987
21. Gulwani, S., Mehra, K.K., Chilimbi, T.: Speed: precise and efficient static estimation of program computational complexity. ACM Sigplan Notices 44(1), 127–139 (2009), https://doi.org/10.1145/1594834.1480898
22. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In: 2006 27th IEEE International Real-Time Systems Symposium (RTSS'06). pp. 57–66. IEEE (2006), https://doi.org/10.1109/RTSS.2006.12
23. Hernández-Del-Olmo, F., Gaudioso, E.: Autotest: An educational software application to support teachers in creating tests. Computer Applications in Engineering Education 21(4), 636–640 (2013), https://doi.org/10.1002/cae.20508
24. Ihantola, P., Ahoniemi, T., Karavirta, V., Seppälä, O.: Review of recent systems for automatic assessment of programming assignments. In: Proceedings of the 10th Koli calling international conference on computing education research. pp. 86–93 (2010), https://doi.org/10.1145/1930464.1930480
25. Knuth, D.E.: The art of computer programming. Vol. 1: Fundamental algorithms. Addison-Wesley (1968)
26. Knuth, D.E.: The analysis of algorithms. In: Actes du Congres International des Mathématiciens (Nice, 1970). vol. 3, pp. 269–274 (1970)
27. Knuth, D.E.: Big omicron and big omega and big theta. ACM Sigact News 8(2), 18–24 (1976), https://doi.org/10.1145/1008328.1008329
28. Licht, B.: Obstacles in learning algorithm run-time complexity analysis. University of Nebraska at Omaha (2022), theses/Capstones/Creative Projects. 193. https://digitalcommons.unomaha.edu/university_honors_program/193
29. Lo, C.K.: What is the impact of chatgpt on education? a rapid review of the literature. Education Sciences 13(4), 410 (2023)
30. McCabe, T.J.: A complexity measure. IEEE Transactions on software Engineering SE-2(4), 308–320 (1976), https://doi.org/10.1109/TSE.1976.233837

31. McGeoch, C.C., Cohen, P.: How to find big-oh in your data set (and how not to). In: Sixth International Workshop on Artificial Intelligence and Statistics. pp. 347–354. PMLR (1997)
32. MIDAS Research Laboratory, I.D.: Corcod: Code runtime complexity dataset (2019), [Online]. Available: https://github.com/midas-research/corcod-dataset (current October 2023)
33. Mišić, M., Lazić, M., Protić, J.: A software tool that helps teachers in handling, processing and understanding the results of massive exams. In: Proceedings of the Fifth Balkan Conference in Informatics. pp. 259–262 (2012), https://doi.org/10.1145/2371316.2371370
34. Moudgalya, K., Ramakrishnan, A., Chemudupati, V., Lu, X.H.: Tasty: A transformer based approach to space and time complexity. arXiv preprint arXiv:2305.05379 (2023)
35. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices 42(6), 89–100 (2007), https://doi.org/10.1145/1273442.1250746
36. Nešković, R.: Predicting time complexity of algorithms on pascal programming language using machine learning techniques. University of Belgrade, School of Electrical Engineering (2021), master thesis (in Serbian)
37. Omonayajo, B., Al-Turjman, F., Cavus, N.: Interactive and innovative technologies for smart education. Computer science and information systems 19(3), 1549–1564 (2022)
38. OpenAI: ChatGPT, https://chat.openai.com/
39. Papadimitriou, C.H.: Computational complexity. In: Encyclopedia of computer science, pp. 260–265. John Wiley and Sons (2003)
40. Pešić, Đ., Mišić, M., Protić, J., Vujošević Janičić, M.: Sistem za generisanje programskih segmenata za ispitivanje u oblasti vremenske slozenosti algoritama. In: ETRAN 2017, Kladovo, Serbia (2017), (in Serbian)
41. Pešić, Đ., Mišić, M., Protić, J., Vujošević Janičić, M.: Prototype implementation of segment assembling software. SJEE 15(1), 71–83 (2018)
42. Pešić, Đ., Mišić, M., Vujošević Janičić, M., Protić, J.: Task generator project, https://github.com/djpesic/TaskGenerator
43. Pešić, Đ., Protić, J., Vujošević Janičić, M., Mišić, M.: Ispitivanje kvaliteta softverski generisanih segmenata u oblasti vremenske složenosti algoritama za automatizovano sastavljanje ispita. In: XXV Skup Trendovi razvoja: Kvalitet visokog obrazovanja, Kopaonik (2019), (in Serbian)
44. Pešić, Đ., Purić, S., Mišić, M., Protić, J.: Softversko generisanje pitanja i odgovora iz oblasti analize složenosti algoritama za testove na kursevima programianja. In: XXII Skup Trendovi razvoja: Nove tehnologije u nastavi, Zlatibor (2016), (in Serbian)
45. Pešić, Đ., Purić, S., Mišić, M., Protić, J.: Softversko generisanje programskih segmenata baziranih na strategijama modeliranim pomoću xml-a. In: ETRAN 2016, Zlatibor, Serbia (2016), (in Serbian)
46. Prechelt, L., Malpohl, G., Philippsen, M., et al.: Finding plagiarisms among a set of programs with jplag. Journal of Universal Computer Science 8(11), 1016 (2002), http://dx.doi.org/10.3217/jucs-008-11-1016
47. Rudolph, J., Tan, S., Tan, S.: Chatgpt: Bullshit spewer or the end of traditional assessments in higher education? Journal of Applied Learning and Teaching 6(1) (2023)
48. Savic, M., Ivanovic, M., Lukovic, I., Delibašic, B., Protic, J., Jankovic, D.: Students' preferences in selection of computer science and informatics studies–a comprehensive empirical case study. Computer Science and Information Systems 18(1), 251–283 (2021)
49. Schleimer, S., Wilkerson, D.S., Aiken, A.: Winnowing: local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data. pp. 76–85 (2003), https://doi.org/10.1145/872757.872770
50. Sedgewick, R., Flajolet, P.: An introduction to the analysis of algorithms. Pearson Education India (2013)

51. ShanghaiRanking: Computer science & engineering. Global Ranking of Academic Subjects (2022), [Online]. Available: http://www.shanghairanking.com/rankings/gras/2022/RS0210 (current July 2023)
52. Siddiq, M.L., Samee, A., Azgor, S.R., Haider, M.A., Sawraz, S.I., Santos, J.C.: Zero-shot prompting for code complexity prediction using github copilot. In: 2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE). pp. 56–59. IEEE Computer Society (2023)
53. Sikka, J., Satya, K., Kumar, Y., Uppal, S., Shah, R.R., Zimmermann, R.: Learning based methods for code runtime complexity prediction. In: Advances in Information Retrieval: 42nd European Conference on IR Research, ECIR 2020, Lisbon, Portugal, April 14–17, 2020, Proceedings, Part I 42. pp. 313–325. Springer (2020)
54. Strejček, J., Trtík, M.: Abstracting path conditions. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. pp. 155–165 (2012), https://doi.org/10.1145/2338965.2336772
55. Surameery, N.M.S., Shakor, M.Y.: Use chat gpt to solve programming bugs. International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290 3(01), 17–22 (2023)
56. Turing, A.M., et al.: On computable numbers, with an application to the entscheidungsproblem. J. of Math 58(345-363), 5 (1936)
57. Vujosevic-Janicic, M., Maric, F.: Regression verification for automated evaluation of students programs. Comput. Sci. Inf. Syst. 17(1), 205–227 (2020), https://doi.org/10.2298/CSIS181220019V
58. Vujošević-Janičić, M., Nikolić, M., Tošić, D., Kuncak, V.: Software verification and graph similarity for automated evaluation of students' assignments. Information and Software Technology 55(6), 1004–1016 (2013), https://doi.org/10.1016/j.infsof.2012.12.005
59. Yildirim, M.: A genetic algorithm for generating test from a question bank. Computer Applications in Engineering Education 18(2), 298–305 (2010), https://doi.org/10.1002/cae.20260
60. Zheng, G., Zhang, X., Wang, R., Zhao, L., Wang, C., Wang, C.: Construction of innovative thinking training system for computer majors under the background of new engineering subject. Computer Science and Information Systems 19(3), 1499–1516 (2022)

Đorđe Pešić is an experienced software developer, with extensive industrial experience in system software development in various domains: compilers, IoT, blockchain, and accessibility software. He received his BSc in 2014, MSc in 2015, and enrolled PhD studies. His focus now is on accessibility and educational software development, with a focus on Rust programming language. He is currently employed at Grid Dynamics.

Milena Vujošević Janičić received a PhD degree in Computer Science from the University of Belgrade in 2013. She is currently an Associate Professor at the Department of Computer Science and Head of the Computer Science Study Program at the Faculty of Mathematics, University of Belgrade. She also works as a principal researcher at Oracle Labs. Her main research interests are in compiler optimizations, automated bug finding, and the application of software verification techniques in different fields.

Marko Mišić received his BSc degree in 2007, his MSc degree in 2010, and his PhD in 2017 in the field of computer engineering and informatics from the University of Belgrade, Serbia. He is an associate professor at the School of Electrical Engineering of

the University of Belgrade. His main fields of interest are parallel programming with an emphasis on GPU computing, image processing, machine learning, social and complex network analysis, as well as educational tools and applications.

Jelica Protić received the PhD degree in electrical engineering from the University of Belgrade in 1999. She is currently a Full Professor and the Head of Department of Computer Engineering and Informatics with University of Belgrade, the School of Electrical Engineering. With Milo Tomasevic and Veljko Milutinovic, she co-authored Distributed Shared Memory: Concepts and Systems (IEEE CS Press, 1997) and presented numerous pre-conference tutorials on this subject. She has long term experience in teaching a diversity of courses in programming languages, as well as the development of various educational software tools. Her research interests include distributed systems, consistency models, complex networks, and all aspects of computer-based quantitative performance analysis and modeling.

Received: July 30, 2023; Accepted: January 23, 2024.