17/2 (2019), 213–241
DOI: 10.5485/TMCS.2019.0466
tmcs@science.unideb.hu
http://tmcs.math.unideb.hu

Teaching
Mathematics and
Computer Science

Programming Theorems and Their Applications

István Fekete, Tibor Gregorics, Kinga Kovácsné Pusztai and Anna Veszprémi

Abstract. One of the effective methodological approaches in programming that supports the design and development of reliable software is analogy-based programming. Within this framework, the method of problem reduction plays a key role. Reducing a given problem to another one whose solving algorithm is already known can be made more efficient by the application of programming theorems. These represent proven, abstract solutions – in a general form – to some of the most common problems in programming. In this article, we present six fundamental programming theorems as well as pose five sample problems. In solving these problems, all six programming theorems will be applied. In the process of reduction, we will employ a concise specification language. Programming theorems and solutions to the problems will be given using the structogram form. However, we will use pseudocodes as descriptions of algorithms resembling their actual implementation in Python. A functional style solution to one of the problems will also be presented, which is to illustrate that for the implementation in Python, it is sufficient to give the specification of the problem for the design of the solution. The content of the article essentially corresponds to that of the introductory lectures of a course we offered to students enrolled in the Applied Mathematics specialization.

Key words and phrases: programming by analogy, problem reduction, programming theorems, program structure, abstract data structure, Python programming, functional style.

ZDM Subject Classification: D40.

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

1. Introduction

This article has been motivated by a university course entitled "Algorithms in Python". This is a relatively new, elective course in the curriculum of the Applied Mathematics specialization. It has been offered three times as yet, always with a great deal of interest. In the introductory lectures of the course, we provide a comprehensive overview of the theoretical programming methodology that is studied by Computer Science majors more extensively, in numerous courses.

The core of this introductory material centers on programming theorems and their applications. The overview of the theoretical background of the programming methodology is detailed in this Section of the article. Presentation of some the most important programming theorems (which are, in brief, general-purpose, abstract algorithmic patterns) is contained in Section 2. Their methodological overview, however, can be found in this current Section.

To illustrate the applicability of programming theorems, we have compiled a problem set of pedagogically motivated problems that build on one another. These problems, together with their solutions, will be presented in Sections 3, 4 and 5. Moreover, solutions will be implemented in Python. More details on the implementation can be found in Section 6.

One of the keys to professional software development might be to build our programs from reliable components and with a systematic approach. Among these components are the so-called programming theorems, which can be effectively used within the framework of structured programming, applying the principle of analogy-based problem reductions.

In this article, we focus on programming theorems and their applications using numerous examples. In our discussion, we draw on our teaching experience and the course materials developed at Eötvös Loránd University (ELTE). In addition, we refer to several research findings, which are believed to be beneficial when it comes to training professional software developers.

One of the most influential sources of research-based programming methodology was the seminal book (Dijkstra, 1976). Programming education at ELTE is heavily influenced by the comprehensive university textbook (Fóthi, 2005), which is based on the Author's early lecture notes (Fóthi, 1983), and his and his fellow instructors' teaching experience (Fóthi & workgroup, 1995).

The current philosophy and detailed content of programming courses offered to Computer Science students at ELTE can be accessed in the two-volume textbook (Gregorics, 2013). The Author presents a unified approach to programming methodologies in accordance with current research results.

The core content of teacher education in Informatics, including best practices in talent management, is contained in the series titled Mikrológia (Szlávi & Zsakó, 2004). An introduction to programming theorems forms a central part of this series. On the "Mester" portal (Horváth & Zsakó, 2013-), the Authors provide a great number of further problems, along with their solutions in which programming theorems are applied.

In proceeding to a high-level overview of the methodology, we assume that the Reader is familiar with the following terms and theoretical concepts: state space, programming problem, (abstract) program, program function, solution, semantics of program constructions as well as constructions of data types. Similarly, proofs of program correctness fall beyond the scope of this article.

In the process of software design, our starting point is the program specification. Then, through a gradual series of steps, we arrive at an abstract program, which can be easily implemented in a programming language of our choice. During this process, we introduce several proven correct sub-components, which retain the validity of the corresponding parts in the specification.

In more detail, in the design phase, we follow the principle of programming by analogy. The essence of this principle is that we try to provide a solution to the given specification with the help of a program that solves another, similar problem. Finding analogies can be made more efficient if we have a collection of general yet readily applicable programs at our disposal.

A carefully selected collection of general abstract program patterns can be called programming theorems, and the process of their application to a given problem is called problem reduction. In other words, programming theorems are general-purpose algorithmic templates formulated at an abstract level. We note that, according to experienced developers, programming theorems are important elementary building blocks of problem solving and software development.

One could characterize these algorithmic patterns as correct and abstract solutions to general problems. The problems and solutions are abstract and general, because in their description we use the following concepts and notations: f is a function and β is a property whose domain is a closed interval of integers denoted by $[m..n]$.

The problem is often expressed in terms of the array $A[1..n]$. Then, the range of indices, denoted by $[1..n]$, corresponds to the interval as found in the programming theorem. Also, the function values $f(i)$ are accessible through the elements of the array, which we denote by $A[i]$. We note that the variants of programming theorems with arrays will not form separate theorems, since these can be easily derived from the original ones with little modification.

It is important to highlight that the form of the programming theorems has not changed significantly to this day. Compared to their earlier versions, the only modifications that happened to the programming theorems are as follows. First, the parameters of the problems are given in a different way, which has resulted in a change in the problem specification. Second, the way loops are constructed has also changed. Previously, loop variables started out from the outside of the interval, while currently they start from the left endpoint. This has caused a change in the terminating conditions of loops, which will be observable in the two programming theorems concerning sequential search methods.

Next, still during the design phase, we use so-called structograms to graphically present abstract solutions in this article. Due to their abstract nature, however, it is possible to express solutions in different forms, such as pseudocode. Furthermore, solutions contain no prescriptions for the coding phase, which makes writing the program code a separate (partially, at least) creative process.

In this article, we describe abstract solutions using the structogram form as part of the design phase. The intricacies of structograms can be found in (Fóthi, 2005) and (Gregorics, 2013). We will also provide an example of the application of the pseudocode form in Section 6. Using either of these two descriptive forms, we regard solutions obtained by applying programming theorems only as abstract algorithms that contain no prescription for coding. Creating the program code is always a separate creative phase in its own right, which is partly independent of the abstract solution. Pseudocodes are naturally closer to program codes, therefore their direct application or "bridging" role can be recommended.

When the program design is implemented in Python, then the description of the problem in a specification language results exactly in the Python code in many cases. In the next section, we introduce such a concise description of the specification. In this case, the problem description in the specification language corresponds to the design of the solution; thus, detailing and algorithmizing it any further can be omitted.

The programming theorems are usually incorporated into the abstract programs by that they are used, most often in the case of smaller sized problems.

In other cases, they may form separate building blocks which can be considered “abstract submodules”. From other points of the programs they can be executed by “abstract call” statements, that is, by references to the headlines appended to the individual structograms.

2. Programming theorems

First of all, the number of programming theorems, as we shall see, is not fixed. The following six general templates represent such a minimal subset of all possible programming theorems that it provides a toolbox for tackling a considerable amount of problems. Secondly, in each of the theorems, the goal is to process the values of some functions and conditions whose domain is a subset of integers. This subset is generally a closed interval. For this reason, the theorems are often called programming theorems on intervals.

After expressing them in words, problems will also be specified formally. The following information will be contained in the specifications:

- The state space; that is, the sets of data type values, together with the variable names corresponding to the respective pieces of data.
- The precondition, which encodes the arbitrary (and still valid) initial values of input variables. (Variable names with the prime symbol will denote the initial values.)
- The postcondition, which provides the set of goal states corresponding to the respective initial states in the form of logic statements.

Among the variables occurring in the state space of a problem, it is the input variables to which the precondition of the problem assigns an initial value. Among the variables occurring in the postcondition, the output variables are those for which it is not prescribed that their value must agree with their initial value specified in the precondition. Notice that the same variable can equally be an input variable and an output variable at the same time.

We also note that the logical language used in the specification is not exactly that same as the language of the first order logic. Instead, for the sake of adequate expressive power, we use its extended “dialect”, which need not be defined precisely as it is hoped that its meaning is unambiguous for specialists.

We propose a specification language in which keywords refer to specific programming theorems; for example, SUM to summation. Besides these keywords, limits of ranges and parameters also occur in this language. Applying the specification language clarifies to which programming theorem(s) the solution is reduced. Another advantage of this system of notations is that it makes the implementation using the functional paradigm easier, which is also well supported by the various language elements and the infrastructure of Python. (Specification languages are increasingly used by systems that support automatic program generation, see Summary.)

As solutions to the specified problems, algorithms will be presented in the form of abstract programs. As mentioned above, we will use structograms to represent algorithms. They are composed of the three fundamental control structures (i.e., sequences, conditional statements, loops), which can be embedded inside one another.

The elementary programs in the boxes of structograms are the assignment, the empty statement (i.e., SKIP) and the reference to another structogram (which can be regarded as an "abstract call").

The following notations are used for the type of variables: $\mathbb{N}$ and $\mathbb{Z}$ denote the set of natural numbers and integers, respectively. The set $\mathbb{L}$ contains the two logical (Boolean) values: $\mathbb{L} = \{\text{true}, \text{false}\}$.

2.1. Counting

Problem:
Let an interval of integers $[m..n]$ and a condition $\beta : [m..n] \to \mathbb{L}$ be given. Determine the number of times β attains the "true" value on $[m..n]$. In the case of the empty interval (i.e., if $m > n$), the value returned must be 0.

Specification:

$$A = (m\colon \mathbb{Z}, n\colon \mathbb{Z}, c\colon \mathbb{N})$$
$$Pre = (m = m' \wedge n = n')$$
$$Post = (Pre \wedge c = \sum_{\substack{i=m \\ \beta(i)}}^{n} 1)$$

Initially, the counter is set to 0. At each point of $[m..n]$ that satisfies β, the counter is increased by 1. Two versions of the algorithm are provided, as seen in Figure 1 and 2. The second version presents the common technique of introducing a logical variable.

If the programming theorem of counting is used in the solution to a given problem, then the following formula can be applied in the appropriate part of specification with the actual parameters: $c = COUNT\ _{i=m}^{n}\ \beta(i)$.

Algorithm:

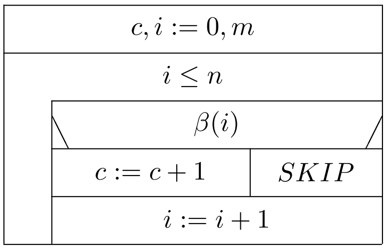


Figure 1. Counting. The base version.

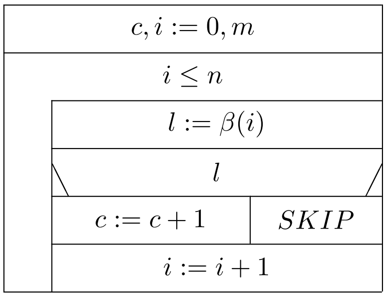


Figure 2. Counting with a logical variable

2.2. Summations

Problem:
There are two common wordings of the problem of summation. Let an interval of integers $[m..n]$ and a function $f : [m..n] \to H$ be given. Let us suppose that the addition operation is defined on the elements of H.

(1) Determine the sum of the values that f takes on the interval $[m..n]$. (In the case of the empty interval, the value of the sum is 0 by definition.)

(2) Let us now introduce the following condition: $\beta : [m..n] \to \mathbb{L}$. Determine the sum of those values of f that are attained in such points of $[m..n]$ that satisfy the property β.

Specification:

$A = (m\colon \mathbb{Z}, n\colon \mathbb{Z}, s\colon H)$

$$Pre = (m = m' \land n = n')$$
$$Post = (Pre \land s = \sum_{i=m}^{n} f(i))$$

In the case of summation restricted by a condition:

$$Post = (Pre \land s = \sum_{\substack{i=m \\ \beta(i)}}^{n} f(i))$$

Algorithms:

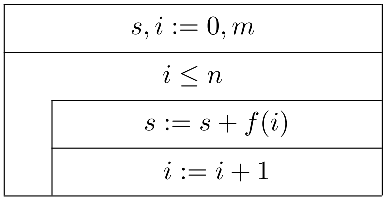


Figure 3. Summation on the whole interval

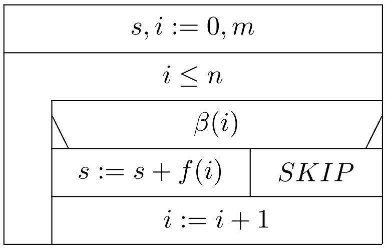


Figure 4. Summation in the points satisfying β

The structograms of the solutions to both versions of the summation problem can be seen in Figure 3 and 4.

The algorithm of the second version can be modified by introducing a logical variable that contains the value of $\beta(i)$, as we have done it in Figure 2.

Summation in general, or reference to the theorem of summing elements with a given property can be concisely given in the specification of the problem by the following formulas: $s = SUM\ _{i=m}^{n} f(i)$ and $s = \underset{\beta(i)}{SUM}\ _{i=m}^{n} f(i)$.

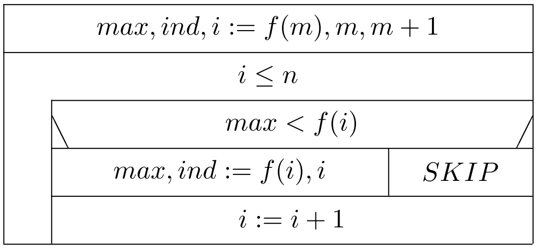
2.3. Maximum selection

Problem:
Let a non-empty interval of integers $[m..n]$ and a function $f : [m..n] \to H$ be given. Let us suppose that a total order is defined on the elements of H. Determine the point at which f attains its maximum value over the interval $[m..n]$, and calculate this value too. (If there is more than one maximum point, any of them can be selected.)

Specification:

$A = (m\colon \mathbb{Z}, n\colon \mathbb{Z}, ind\colon \mathbb{Z}, max\colon H)$
$Pre = (m = m' \wedge n = n' \wedge n \geq m)$
$Post = (Pre \wedge ind \in [m..n] \wedge max = f(ind) \wedge \forall i \in [m..n]\colon max \geq f(i))$

Algorithm:



```
$max, ind, i := f(m), m, m+1$
$i \leq n$
    $max < f(i)$
        $max, ind := f(i), i$  |  $SKIP$
    $i := i + 1$
```

Figure 5. Maximum selection

As seen in Figure 5, the two values we are interested in finding are initialized at the left endpoint of $[m..n]$, and they get modified if and only if we arrive at a larger value of f at other points of the interval (going from left to right). In the case of more than one maximum point, the leftmost will be selected.

The theorem of maximum selection can be referred to in the specification language as follows: $(max, ind) = MAX\ {}_{i=m}^{n}\ f(i)$. If we are only interested in the maximum value or its corresponding point, then the MAX function becomes single-valued.

The six programming theorems are viewed as parts of programming methodology. However, the other significant chapter of computer science is the theory of algorithms and data structures. That discipline thinks of one of the programming theorems, the maximum selection as its own part, because of being a building block of several algorithms.

2.4. Conditional maximum search

Problem:
Let an interval of integers $[m..n]$, a function $f : [m..n] \to H$ and a condition $\beta : [m..n] \to \mathbb{L}$ be given. Let us suppose that a total order is defined on the elements of H. Determine the point at which f attains its maximum value over those points of $[m..n]$ that satisfy β. As above, calculate this value too. (Note that there might not be any points satisfying β in $[m..n]$, which is inherently the case if $m > n$.)

Specification:
$A = (m\colon \mathbb{Z}, n\colon \mathbb{Z}, l\colon \mathbb{L}, ind\colon \mathbb{Z}, max\colon H)$
$Pre = (m = m' \wedge n = n')$
$Post = (Pre \wedge (l = \exists i \in [m..n]\colon \beta(i)) \wedge (l \to ind \in [m..n] \wedge max = f(ind) \wedge \beta(ind) \wedge (\forall i \in [m..n]\colon \beta(i) \to max \geq f(i))))$

Algorithm:

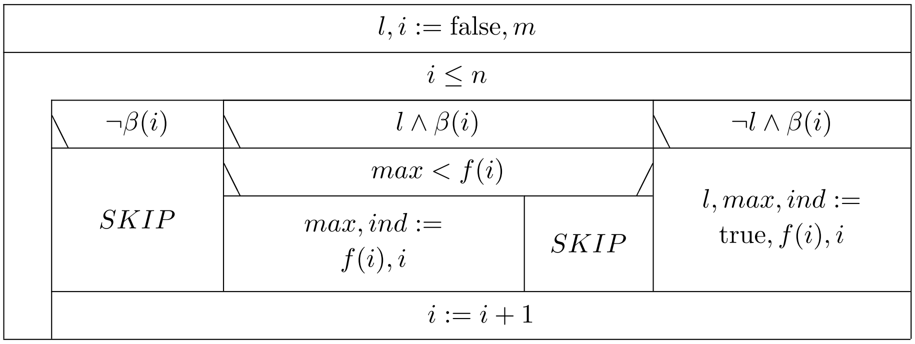


Figure 6. Conditional maximum search

The algorithm in Figure 6 uses the logical variable l to identify the first point satisfying β. At that point, the output variables get initialized. Next, at other points satisfying β, the algorithm works in a similar way as the algorithm of maximum selection. Points where β is false are skipped.

The specification of conditional maximum search can be given by the following concise formula, where the MAX function can be single- or double-valued, but the logical variable l has to be provided: $(l, max, ind) = \underset{\beta(i)}{MAX}\,{}_{i=m}^{n}\, f(i)$.

2.5. Sequential selection

Problem:
Let an integer m and a condition $\beta : \mathbb{Z} \to \mathbb{L}$ be given. Determine the leftmost point starting with m that satisfies β, if we know that such point definitely exists.

Specification:
$A = (m\colon \mathbb{Z}, ind\colon \mathbb{Z})$
$Pre = (m = m' \land \exists i \geq m\colon \beta(i))$
$Post = (Pre \land ind \geq m \land \beta(ind) \land \forall i \in [m..ind-1]\colon \neg\beta(i))$

Algorithm:

```
$ind := m$
$\neg\beta(i)$
    $ind := ind + 1$
```

Figure 7. Sequential selection

Figure 7 shows an algorithm whose loop variable guarantees termination due to the existence of a point that satisfies β. (We note that a logical variable can be introduced here as well.)

The concise form of the specification of sequential selection is the following: $ind = SELECT_{\ ind \geq m}\ \beta(ind)$.

2.6. Sequential search

Problem:
Let an interval of integers $[m..n]$ and a condition $\beta : [m..n] \to \mathbb{L}$ be given. First, decide whether $[m..n]$ contains a point that satisfies β. If so, then determine the first such point in $[m..n]$ starting from the left endpoint. (The empty interval is an equivalent of β evaluating to False on the whole interval.)

Specification:
$A = (m\colon \mathbb{Z}, n\colon \mathbb{Z}, l\colon \mathbb{L}, ind\colon \mathbb{Z})$
$Pre = (m = m' \land n = n')$
$Post = (Pre \land (l = \exists i \in [m..n]\colon \beta(i)) \land (l \to ind \in [m..n] \land \beta(ind) \land \forall i \in [m..ind-1]\colon \neg\beta(i)))$

Algorithm:

```
$l, i := \text{false}, m$
$\neg l \wedge i \leq n$
    $l, ind := \beta(i), i$
    $i := i + 1$
```

Figure 8. Sequential search

The answer to the yes/no question specified in the problem description is contained in the logical variable l (see Figure 8). If the value of l is true, then the first number satisfying β will get stored in the variable ind.

The specification of sequential search can be concisely described with a single- or double-valued function, where the logical variable l cannot be omitted: $(l, ind) = SEARCH\ _{i=m}^{n}\ \beta(i)$.

3. Three problems related to prime numbers

In this section, we present three problems that are connected to the notion of prime numbers. The algorithms that solve these problems can be derived by reduction to programming theorems, thus providing examples of the applicability of the theorems. Moreover, beyond sharing a common theme, the problems build on each other. Specifically, the algorithm to the first example will be used to solve the other two problems.

3.1. Primality test

Our introductory example, which is about deciding whether a positive integer is prime, will be solved with the technique of problem reduction. However, a separate step will also be necessary in the solution process.

Problem 1. Decide whether a natural number $n \geq 1$ is prime.

According to the most commonly used definition of prime numbers, $n \geq 2$ is prime exactly when it has no non-trivial divisors. In other words, there is no k dividing n such that $2 \leq k \leq n - 1$.

As is known, it suffices to check whether n is a multiple of any integer up to $\lfloor\sqrt{n}\rfloor$. Notice that this step does not restrict the validity of the definition – in the cases of $n = 2$ and $n = 3$, the interval $[2..\lfloor\sqrt{n}\rfloor]$ is empty, thus it does not contain any divisors, which is in line with the fact that 2 and 3 are primes.

The above definition of prime numbers cannot be applied, however, in the $n = 1$ case, but by definition, 1 is not a prime. What is important to note is that we need to separate this case (i.e., if $n = 1$) in the algorithm, and a negative answer must be given in this branch.

Let us formalize the above definition for integers $n \geq 2$ using negation:

$$\text{the integer } n \geq 2 \text{ is not a prime} \iff \exists k \; (2 \leq k \leq \lfloor\sqrt{n}\rfloor) \colon k|n$$

The description of the searching problem using the specification language introduced in the previous section is as follows:

$$l = SEARCH_{k=2}^{\lfloor\sqrt{n}\rfloor} \; k|n$$

In order to decide the validity of the statement, we use the programming theorem of sequential search. The interval as specified in the theorem will be $[2..\lfloor\sqrt{n}\rfloor]$ in this case, and the condition β will correspond to the relation $k \mid n$. We note that the *ind* variable will not be needed from the theorem, since if n is not a prime, then there is no need to know which number is its smallest non-trivial divisor.

The result of the primality test can be obtained by negating the value of the logical variable in the sequential search algorithm. Notice that in that algorithm, the logical variable l can already be used in a negated form. In this case, its meaning is not that we have not found a divisor yet, but that there is no proof to reject the assumption about the primality of n as yet. The algorithm of primality test is given in Figure 9 as a structogram.

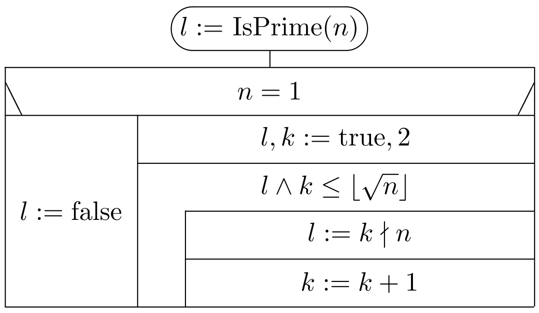


Figure 9. Primality test

3.2. First prime after a given number

Suppose we are interested in the first prime number that is greater than, say, 1000. Instead of providing an answer to this specific question, we can design an algorithm that tackles this problem in its general form.

Problem 2. Determine the first prime number that is greater than a given integer $m \in \mathbb{Z}$.

Notice that for all integers m, there is a prime that is greater than m, because there are infinitely many prime numbers. Given this fact, the condition of the sequential selection algorithm is satisfied if we start searching for the first prime number among the integers starting with $m+1$. Using the specification language, we arrive at the following postcondition:

$$ind = SELECT\ _{ind \geq m+1} IsPrime(ind)$$

The solution, as seen in Figure 10, can be derived using the programming theorem in Section 2.5. To test primality, we make use of our previous algorithm, which is referred to when we assign a value to the logical variable l.

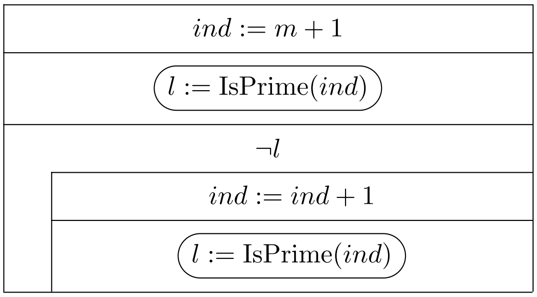


Figure 10. First prime after a given number

3.3. The number of primes in a given interval

Similarly, instead determining how many prime numbers are between 1 and 1000, it might be preferable to solve a more general problem.

Problem 3. Determine the number of prime numbers in the interval of integers $[m..n]$.

The specification of the problem can be concisely formulated as follows:

$$c = COUNT\ _{i=m}^{n} IsPrime(i)$$

We reduce this problem to the programming theorem of counting, which we detailed in Section 2.1. Again, for the primality test, we use the algorithm developed in Section 3.1. The final solution to the problem is shown in Figure 11.

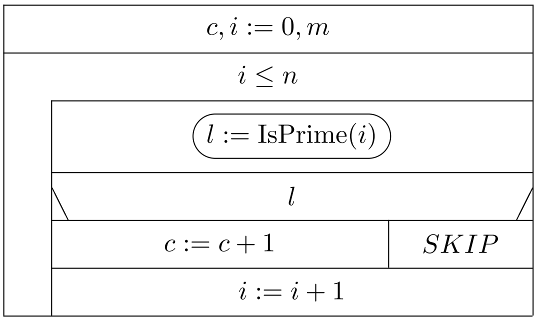


Figure 11. Number of primes in an interval

4. Embedding programming theorems

In this section, we pose a somewhat more complex problem than the previous ones.

Problem 4. Let a maixtrix of integers with m rows and n columns ($A[1..m, 1..n]$) be given, where $m, n \geq 1$. Determine the row in which the sum of prime numbers is the largest.

The wording of the problem is clearly indicative of the programming theorems of maximum selection and conditional summation as well as the notion of prime numbers. In solving the problem, we will apply these two theorems and use the algorithm of primality test. This observation can be expressed with the following specification:

$$(max, ind) = \underset{IsPrime(A[i,j])}{MAX\ _{i=1}^{m}} \ (SUM\ _{j=1}^{n} A[i, j])$$

The matrix data structure is a well-known generalization of an array whose elements are arrays themselves; in other words, it can be thought of as the column vector of its rows. We note that the problems and solutions described in terms of arrays are closely connected to the programming theorems on intervals, so much so that they can be regarded as the immediate applications of these theorems. In the following, we denote the i-th row of the matrix with $A[i, 1..n]$.

From the three concepts in the problem description, the algorithm of maximum selection is the most important constituent of the program structure. Conditional summation appears as the algorithm that generates the values to be compared. Also, this is the procedure where the primality test takes place.

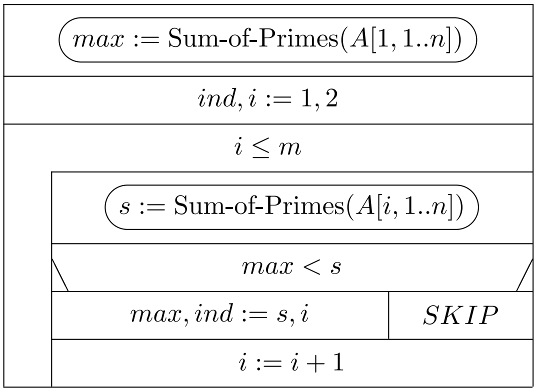


Figure 12. The row of the matrix with maximal sum of primes

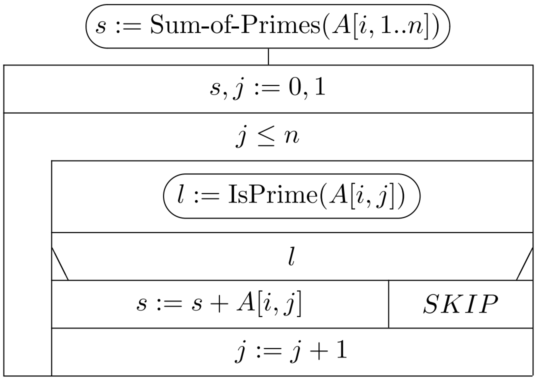


Figure 13. Sum of primes in a row of the matrix

The top layer of the solution (see Figure 12) can be derived from the programming theorem of maximum selection. Embedded in this theorem appears the algorithm that is created using the theorem of conditional summation. This algorithm is applied to the rows of the matrix to test the primality of their elements. This could be regarded as the second layer in the program structure (see Figure 13).

In our experience, there is another design approach to solving this problem. We could introduce an auxiliary array, denoted with $B[1..m]$, and apply the two programming theorems sequentially, one after the other. This solution would first collect the sum of primes from the rows of the matrix and store them in B. Then, it would perform a maximum selection on B. No matter how logical and straightforward this solution may seem, our algorithm-based viewpoint cannot possibly accept it, because of the redundancy of the linear-space auxiliary array. Furthermore, this solution would be less time efficient as the original one.

For the sake of completeness, we note that using data-driven program design, we could devise another abstract and well-structured program; however, this is beyond the scope of this article. Nevertheless, without being skilled at programming methodologies, solving problems of similar complexity can easily lead to design-related difficulties. For example, in this problem, we were required to use some guiding principles to coordinate the three concepts in the problem description so that the final solution could be clearly structured and easy to understand.

5. Dijkstra's shortest path algorithm, array version

Our last problem is a well-known one – the single-source shortest paths problem for graphs with non-negative edge weights. The procedure that solves this problem is Dijkstra's algorithm (Cormen et al., 2003), (Fekete & Hunyadvári, 2016). In this section, we attempt to provide a new angle for the design phase by reducing the iterative block of the algorithm (i.e., the selection of the minimal cost open node) to the programming theorem of conditional maximum (here, minimum) search.

Problem 5. Let $G = (V, E)$ be an either directed or undirected graph with non-negative edge weights. Let a node $s \in V$ be fixed; this will be the source (or initial) node. Determine the shortest (i.e., minimal cost) paths to all nodes of the graph starting from s.

In Figure 14, there is an undirected graph with positive edge weights, in which the source node is $s = 1$. The shortest paths form a (directed, in fact) spanning tree in the graph, as can be observed in Figure 15 along with the costs of reaching the respective nodes.

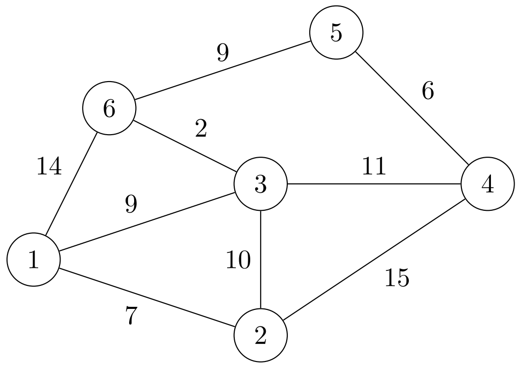


Figure 14. Dijkstra's algorithm: the initial state of the graph

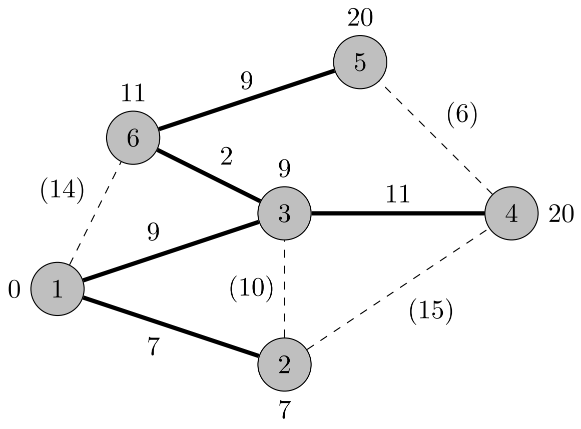
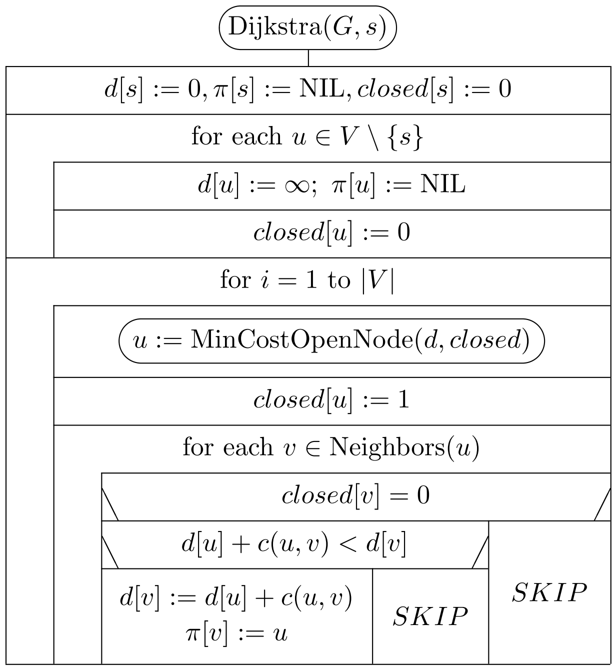


Figure 15. Dijkstra's algorithm: the spanning tree of the minimal cost paths

The graph in Figure 14 and 15, in drawn form, corresponds to the notion of abstract data structure. Dijkstra's algorithm will be given at this level of abstraction. Then, as a next step, the array or the adjacency list version of the algorithm could be implemented with some coding skills.



```
Dijkstra($G, s$)
$d[s] := 0, \pi[s] := \text{NIL}, closed[s] := 0$
for each $u \in V \setminus \{s\}$
    $d[u] := \infty;\ \pi[u] := \text{NIL}$
    $closed[u] := 0$
for $i = 1$ to $|V|$
    $u :=$ MinCostOpenNode($d, closed$)
    $closed[u] := 1$
    for each $v \in$ Neighbors($u$)
        $closed[v] = 0$
            $d[u] + c(u, v) < d[v]$
                $d[v] := d[u] + c(u, v)$
                $\pi[v] := u$
            else
                $SKIP$
        else
            $SKIP$
```

Figure 16. Dijkstra's algorithm (high level)

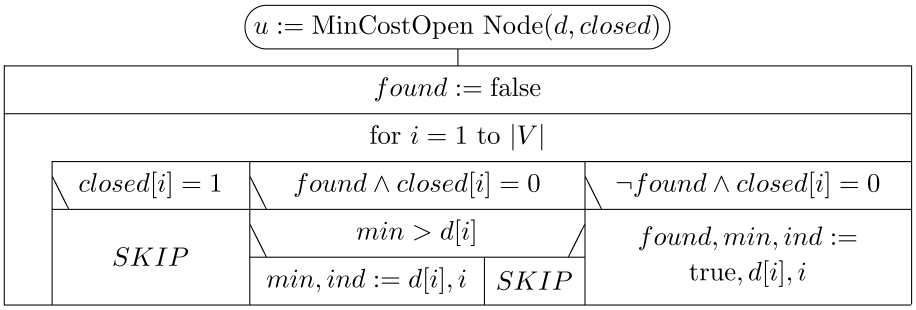
It can be proven that the selected node can be closed after this step, because no other path with a smaller cost can be found to this node. Since the algorithm is greedy, every node is selected exactly once.

We emphasize that, in each iteration step, Dijsktra's algorithm selects a node u which is not closed yet (i.e., $closed[u] = 0$). Considering the still open nodes, there is no cost that is smaller than the cost corresponding to node u (i.e., $d[u]$). (Of course, if there is more than one node with a minimal cost, we can select any of them.)

In this procedure, we can recognize the programming theorem of conditional maximum search, as discussed in Section 2.4. In this case, it is in fact a minimum selection, which necessitates flipping the inequality sign in only one instance. The search interval in the theorem now becomes $[1..|V|]$, the condition $closed[i] = 1$ corresponds to $\beta(i)$, and the cost values, denoted with $d[i]$, replace the function values $f(i)$. Our consideration leads to the following specification:

$$(found, min, ind) = \underset{closed[i]=0}{MIN}\ {}_{i=1}^{|V|} d[i]$$

The result of this problem reduction is the structogram in Figure 17.



```
$u := \text{MinCostOpen Node}(d, closed)$
$found := \text{false}$
for $i = 1$ to $|V|$
    $closed[i] = 1$
        $SKIP$
    $found \wedge closed[i] = 0$
        $min > d[i]$
            $min, ind := d[i], i$
        else
            $SKIP$
    $\neg found \wedge closed[i] = 0$
        $found, min, ind := \text{true}, d[i], i$
```

Figure 17. Search for an open node with minimal cost

We call the attention to the fact that the algorithm which selects the minimal cost open node is quite complex per se. By being able to find an analogy with a programming theorem, not only have we simplified the process of program development quite considerably, but we have also improved several safety aspects in program design. Last but not least, the final form of the algorithm can be comprehended more easily this way.

6. About the Python programming

As was mentioned in the Introduction, this article has been motivated by a university course titled "Algorithms in Python". A general overview of the course will be given below, which will then be followed by a short description of the first topic of the lecture series. A more in-depth description of the entire course could be the topic of a future article.

6.1. Introducing the Python course

A comprehensive description of the course can be accessed through the course materials on the Lecturer's website (Fekete, 2019). These materials include a general description of the course, a brief summary of each topic, the program codes in Python as presented in the lectures and a list of home assignments.

First of all, it should be noted that this is not a course focusing on teaching the programming language in the classical sense. The emphasis is rather placed on solving problems related to the world of algorithms and data structures with the help of Python programs. The Python language is used on a basic level. That is to say, programming the graphical interface is beyond the scope of the course, but the functional properties of the language are used later in the course.

The course has four basic pillars. Firstly, the problems generally come from Mathematics, secondly, more specifically, from the field of algorithms and data structures, thirdly, Python is utilized in their solution, and fourthly, general results from programming methodology are applied during the designs of the programs.

Next, the general guiding principle of the lectures is the following. During the first weeks, we use the language intuitively. A comprehensive overview of the language-related elements always follow the applications in subsequent weeks as well. This also means that only three lectures are allocated to the presentation of a summary of the most important Pythonic elements, while the rest of the lectures revolve around solving sample problems by writing programs interactively during the lectures.

Some individual work is expected of students in order for them to acquire the most basic elements of Python. To guide them in this process, we recommend them several relevant resources in book form as well as from the Internet.

Among the books, the one that is the most relevant to our course in its objectives and tone is the book (Goodrich et al., 2013). By being perhaps the first of its kind, it presents a unique way of teaching algorithms using the basic data structures of Python. In Python, arrays are replaced with indexable lists; there are no pointers that are visible to and can be accessed by the programmer; and, furthermore, the ordered list of n elements (i.e., the n-tuple) and the dictionary (which is basically a hash-table) are among the simplest Pythonic data structures.

Each topic of the course is covered in one or two lectures. Among these topics are programming theorems and their applications, the most typical algorithms with different runtime complexities, problems that can be solved with backtracking, graph algorithms and designing intelligent input-output methods from the console.

Finally, the course can be completed by submitting home assignments. It is suggested that students work on these problems in pairs. Students also have to present their solutions and programs to members of the course staff.

We also mention that the syllabus and some course materials of an extracurricular course tailored to 12- to 18-year-old students (Princz, 2017) were presented (and a paper submitted) in the international conference in commemoration of the 100th anniversary of the birth of Tamás Varga, held at ELTE in Nov 2019.

6.2. Presenting the introductory topic of the course

The first topic of the course "Algorithms in Python" is about programming theorems and their applications. Together with a brief summary of the basic elements of the language, programming theorems are covered in three lectures. Their presentation corresponds to the structure we described in Section 2. Afterwards, the five problems described above are posed, and their abstract, theoretical solutions are provided subsequently, all in line with Sections 3, 4 and 5.

In the Introduction, the pedagogical merits of the problem set were taken into account. From a Pythonic aspect, it should be added that the problems provide ample opportunity for applying almost all fundamental elements of the language. It should especially be highlighted that the solution to the 4th problem could be made significantly shorter and more efficient by using two programming theorems embedded in one another, which would be an example of functional programming (cf. this Section). An interesting aspect of the 5th problem is that edge-weighted graphs are represented with a dictionary whose keys are the nodes of the graph, and the values are lists of ordered pairs (which represent the neighbors of the nodes with the weights of the corresponding edges).

In the following, we provide an in-depth solution to one of the problems. Later, when students become more skilled, solutions can be less detailed.

When implementing the solver algorithms, we break with the structogram form that has been used so far. There is no one-to-one correspondence between the clear and streamlined form of structograms and the coded versions of the algorithms, because it would result in program codes that are pragmatically not accepted by the programming profession. For this reason, we have redrafted the abstract descriptions of the algorithms.

Pseudocodes are used for this purpose, which were originally introduced, precisely defined and applied for describing algorithms in one of the most authoritative books in the world of algorithms and data structures, that is (Cormen et al., 2003).

The structogram in Figure 7, which depicts the algorithm solving the first problem (that is, the primality test), can be rewritten using pseudocode as in Figure 18. There is one new element in the algorithm, namely the fact that as soon as the first divisor of the tested number is found, the execution breaks out of the search loop and returns the value False.

```
IsPrime($n$)
1 if $n = 1$ then
2     return false
3 else
4     $k := 2$
5     while $k \leq \lfloor\sqrt{n}\rfloor$ do
6         if $k \mid n$ then
7             return false
8         $k := k + 1$
9     return true
```

Figure 18. Primality test, version 1, pseudocode

Additional modifications of the pseudocode can be seen in Figure 19. First, we dropped the else branch of the conditional expression, and second, the while-loop has been replaced with a for-loop. The corresponding Python code is provided in Figure 20.

```
IsPrime($n$)
1 if $n = 1$ then
2     return false
3 for $k = 2$ to $\lfloor\sqrt{n}\rfloor$ do
4     if $k \mid n$ then
5         return false
6 return true
```

Figure 19. Primality test, version 2, pseudocode

```python
def is_prime(n):
    if n <= 1:
        return False
    for k in range(2, int(n**0.5)+1):
        if n % k == 0:
            return False
    return True
```

Figure 20. Primality test, version 2, Python code

Python as a multi-paradigm programming language supports the use of imperative, object-oriented as well as functional programming. From the latter, list comprehension and lambda expressions are used explicitly in the course. Using this approach, everything that is composed of a head element and the rest of the list can be regarded as a list. Furthermore, lists represent a universal data structure that can be traversed with aggregator functions (e.g., max()), enumerators and anonymous functions (i.e., lambda expressions), without using explicit while- and for-loops.

```
def row_prime_sum(row):
    return sum(elem for elem in row if is_prime(elem))

max_ind, max_row = max(enumerate(A),
    key=lambda ind_row: row_prime_sum(ind_row[1]))

max_ = row_prime_sum(max_row)
```

Figure 21. The row of the matrix with maximal sum of primes, Python code

The Python program code snippet in Figure 21 demonstrates the change from the imperative loops toward the modern functional programming. This program code is the functional solution to the fourth problem, whose classical solution at an abstract level was given in Section 4. The resulting program code is shorter and more efficient than the one written using the imperative paradigm; however, it can be less intuitive for programmers working with non-functional paradigms.

The Python programs presented above originate from working examples, omitting the test data and the import of the is_prime function on Figure 21.

7. Summary

In the following, we summarize the key points of the article and provide some further considerations.

Programming theorems organically emerge in a professional environment, which means that their number is not fixed from a theoretical perspective. Besides the six fundamental theorems that we discussed in this article, there are more complex programming theorems in use. Notable examples of more advanced theorems include the binary search algorithm and the theorems of merging, assortment and solving recurrence relations (Fóthi, 2005), (Gregorics, 2013), (Szlávi & Zsakó, 2004). The Authors provide ten programming theorems on the "Mester" portal mentioned in the Introduction. There are an average of 30 problems belonging to each of the theorems (Horváth & Zsakó, 2013-).

The presentation of programming theorems has been extended by introducing elements of a specification language. Afterwards, five problems follow, building on one another, which also provide an opportunity for applying each of the programming theorems.

The first three problems were related to the notion of prime numbers. Testing primality presented some interesting peculiarities. The case of $n = 1$, which is not a prime, requires a unique treatment; since the application of the programming theorem did not cover this case. In the case of $n \geq 2$ (where n is an integer), the programming theorem of sequential search could be applied.

The second and third problems could be solved by applying the programming theorems of sequential selection and counting elements with given features. The algorithm of testing primality was used in solving both problems, which was indicated by a reference to the headline of the structogram (i.e., abstract function call).

The next, more complex problem was described for arrays, and it turned out to be fairly instructive for the following reason. Its wording contained three key concepts whose handling required sufficient amount of methodological knowledge. In line with the analogy-based approach we followed in the article, the problem was solved by a conditional summation which used the algorithm of testing primality and was embedded in a maximum selection. It is worthwhile to consider that the top layer of the algorithm operates on the matrix as a whole while lower levels procedures work with the rows and the elements of the rows of the matrix.

An attractive application of the programming theorem of conditional maximum search could be found in designing the array version of Dijkstra's algorithm. The algorithm finds the minimal cost paths to all nodes starting from a given source node by selecting a minimal cost open node in each step. Then, it tries to improve the costs to reach all of its open neighbors. Afterwards, the selected node becomes closed.

By recognizing the programming theorem of conditional maximum (here, minimum) search in this procedure, we have simplified the design phase and improved the reliability of the solution at the same time. Another important aspect to consider is the enhanced readability of the resulting algorithm.

We have also provided a glimpse into how solutions to the problems can be implemented in Python. The structograms showing the solver algorithms were first converted to pseudocodes, which are closer to the implementation phase. This step was illustrated by the algorithm of the primality test. A solution using the functional paradigm was also given to the more complex, matrix-related problem.

We present some additional topics that are outside of the scope of this article, but are doubtless linked to the broader topic area.

Although we have not dealt with proving the correctness of programming theorems in detail, it might be worthwhile to provide a high-level overview of how such proofs can be constructed. The pre- and postconditions in the specification as well as the invariant statements in the abstract program mark the starting point of the proof method. The key concept of the proof is the weakest precondition. Given a program block and the postcondition to be satisfied, there exists a detailed calculus to calculate the weakest precondition. The proof procedure is aligned with the structure of the program. In each step, we prove that the known possible initial states of given program blocks are in the largest set of possible initial states that are calculated from their postcondition (in fact, this is expressed by the weakest precondition) (Dijkstra, 1976), (Fóthi, 2005), (Gregorics, 2013).

In the examples of this article, programming theorems were applied to integer intervals and arrays (namely, vectors and matrices). It is important to note, however, that the theorems can be extended to such structures on which an enumerator can be defined; that is, structures on which the functions First(), Next(), Current() and End() can be constructed (Gregorics, 2013), (Gregorics, 2010), (Gregorics, 2012a).

The theory of programming and the theory of algorithms and data structures have many points of contact. For example, the programming theorems of maximum selection and binary search can be considered to be part of the theory of algorithms as well. Furthermore, a variant of coalescing appears in merge sort, and solving recurrence relations is also a common algorithmic problem. These examples show that the world of algorithms (e.g., the chapter of search and selection) can offer a range of potential solutions, especially when a given step in the design phase of a program requires an algorithmic background; this is called the function-oriented design (Gregorics, 2013).

In other cases, the introduction and implementation of composite data types could be necessary in the type-oriented design (Gregorics, 2013). The theory of algorithms and data structures appears as a related field here, cf. the last example. Fundamental data structures, such as arrays, stacks, queues, lists, trees and heaps are part of both fields; consequently, they can be essential when it comes to solving programming problems (Cormen et al., 2003), (Fekete et al., 2001), (Fekete & Hunyadvári, 2016).

Constructing the most appropriate program structure is within the scope of modular programming (Gregorics, 2013). In the case of smaller problems, this is not a burning issue; however, it may be beneficial to consider the possibility of structuring our programs already during the process of laying the theoretical foundations. We mention that by modifying some of the fundamental notions discussed in the article and by introducing the concepts of subspace and dynamic programs, we can pave the theoretical way to modular programming (Gregorics, 2012c).

One of the research directions in the field of software technology attempts to put the principle of executable specification into practice. (A good example of this could be the queries in the well-known SQL language in database programs.) Our research group has also reached some preliminary results. An automatic program-generating system has been devised that produces programs based on their specification (Csepregi et al., 2007). With a similar aim in mind, a library of class templates has also been implemented from which, through inheritance and instantiation, object-oriented solutions can be given to problems related to programming theorems (Gregorics, 2012b).

These additional topics show that there are many other interesting questions to be analyzed and discussed.

Acknowledgements

The authors gratefully thank Péter Princz for useful discussions, reviewing the text and solving several hard typographical problems.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2003). *Új algoritmusok.* Scolar Kiadó.

Csepregi, S., Dezső, A., Gregorics, T., & Sike, S. (2007). *Automatic implementation of service required by components.* PROVEX'2007 Workshpop, ETH Technical Report 567.

Dijkstra, E. W. (1976). *A discipline of programming.* Englewood Cliffs, NJ, Prentice-Hall.

Fekete, I. (2019). *Python kurzus.* `http://ifekete.web.elte.hu`. ELTE.

Fekete, I., Gregorics, T., & Hunyadvári, L. (2001). Abstraction levels of data type. In E. Kovacs (Ed.), *Proceedings of the 5th international conference of applied informatics* (p. 55-64). Eger, Hungary.

Fekete, I., & Hunyadvári, L. (2016). *Algoritmusok és adatszerkezetek.* Digitális Tankönnyvtár.

Fóthi, Á. (1983). *Bevezetés a programozáshoz.* Tankönyvkiadó.

Fóthi, Á. (2005). *Bevezetés a programozáshoz.* ELTE Eötvös Kiadó.

Fóthi, Á., & workgroup. (1995). Some concepts of a relational model of programming. In L. Varga (Ed.), *Proceedings of the fourth symposium on programming languages and software tools* (p. 434-446). Visegrád, Hungary.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data structures and algorithms in Python.* John Wiley & Sons.

Gregorics, T. (2010). Programming theorems on enumerator. *Teaching Mathematics and Computer Science, Debrecen*, *8*(1), 89-108.

Gregorics, T. (2012a). Abstract levels of programming theorems. *Acta Universitatis Sapientiae, Informatica*, *4*(2), 247-259.

Gregorics, T. (2012b). Analogous programming with template class library. *Teaching Mathematics and Computer Science, Debrecen*, *10*(1), 135-152.

Gregorics, T. (2012c). Concept of abstract program. *Acta Universitatis Sapientiae, Informatica*, *4*(1), 7-16.

Gregorics, T. (2013). *Programozás 1-2.* ELTE Eötvös Kiadó.

Horváth, G., & Zsakó, L. (Eds.). (2013-). *Mester online feladatbank.* `http://mester.inf.elte.hu`. ELTE IK, NJSzT.

Princz, P. (2017). *A computer study group for 12-18 yrs.* `https://gitlab.com/users/princzp/projects`.

Szlávi, P., & Zsakó, L. (2004). *Módszeres programozás 19. programozási tételek.* ELTE Mikrológia sorozat.

István Fekete
ELTE Faculty of Informatics
3in Research Group, Martonvásár, Hungary

E-mail: fekete.istvan@inf.elte.hu

Tibor Gregorics
ELTE Faculty of Informatics

E-mail: gt@inf.elte.hu

Kinga Kovácsné Pusztai
ELTE Faculty of Informatics

E-mail: kinga@inf.elte.hu

Anna Veszprémi
ELTE Faculty of Informatics

E-mail: veanna@inf.elte.hu

(Received September, 2019)