The Role of Algorithm in General Secondary Education Revisited

Daniel Lessner

Department of Software and Computer Science Education, Faculty of Mathematics and Physics,
Charles University in Prague, Czech Republic
lessner@ksvi.mff.cuni.cz

Abstract. The traditional purpose of algorithm in education is to prepare students for programming. In our effort to introduce the practically missing computing science into Czech general secondary education, we have revisited this purpose. We propose an approach, which is in better accordance with the goals of general secondary education in Czechia. The importance of programming is diminishing, while recognition of algorithmic procedures and precise (yet concise) communication of algorithms is gaining importance. This includes expressing algorithms in natural language, which is more useful for most of the students than programming. We propose criteria to evaluate such descriptions. Finally, an idea about the limitations is required (inefficient algorithms, unsolvable problems, Turing's test). We describe these adjusted educational goals and an outline of the resulting course. Our experience with carrying out the proposed intentions is satisfactory, although we did not accomplish all the defined goals.

Keywords: computing science education, general secondary education, mandatory computer science foundations, concept of algorithm

1 Introduction

The average student of a Czech grammar school is not even aware of the existence of computing science (CS). We would like to change this. To provide some background for the desired discussion, we have developed an experimental introductory course to computing science, which will take up two hours weekly for ten months. The traditional role of algorithm in such courses, i.e. a preparation for programming, is revisited here, so that it would blend better into our general secondary education.

In this paper we first describe our reasons for adjusting the use of algorithms to grammar schools and the resulting goals. Then we follow the notion through the outline of our course. The next section introduces the criteria we used when working with algorithms described in natural language – an approach to enhance algorithmic thinking implemented in our course. Finally we share a few very brief observations from the realized experimental course.

To begin with we should like to clarify the terms we use.

1.1 Czech grammar schools

The Grammar school in Czech Republic is a branch of secondary education (the students are 15-19 years old). It provides general education as a preparation for further university studies of almost any kind. Around 20% of the relevant age group attend grammar schools. The curriculum is defined in the Framework Education Programme for Secondary General Education (FEP) [1]. FEP defines six so-called key competences, complex structures of knowledge, skills, attitudes and values to be developed. These competences are: to be able to solve problems, to communicate, to learn, civic, entrepreneurial competence and personal and social competence. The educational content itself is first of all a means to develop these competences. Still, one needs languages to communicate, sciences to solve problems etc.

Computer related education is represented in FEP by a subject called *Information Science and Information and Communication Technologies*. The main focus is on utilizing digital technologies. Anything deeper is mentioned only rarely and briefly. FEP formally requires every grammar-school student to "apply an algorithmic approach to problem solving". It is the last item of nine in that section, and it even includes "introduction to programming". However, the usual way is to include these issues in an *optional* seminar for motivated students. We admittedly shift and change (and sometimes indeed lower) the usual goals and contents of such seminars, but this is due our effort to include *all* grammar-school students.

1.2 Computing science

In this paper, computing science means the discipline focused on efficient information processing [2]. The requirement of efficiency naturally, but not necessarily, leads to computer utilization. From the grammar school point of view, CS is the peer to natural sciences, such as physics or chemistry [6, 11]. They have a strong theory in the background, mathematical models, terminology, methods, extremely useful everyday technical applications and they provide inspiring problems and both cognitive and technical means to solve them.

Computing science in this paper has almost nothing to do with computer user-skills and very little to do with pure programming as a tool to develop software. The role of programming in our course has to be made clear here: it is not among the educational goals. This is not a unique idea for an introduction to CS [3]. Main goals of programming at grammar schools are clear communication, using a formal system and systematic thinking and workflow. We assume that these goals can be achieved by exposing students to the use of more formal systems (e.g. regular expressions, flowcharts etc.). A variety of possibilities is shown in the Bebras contest [5].

Omitting writing programs as a goal gives us extra time to focus on our goals more directly[1]. We work with sample programs, just as with other means,

[1] After all, the ability to write programs is difficult to use without the ability to explain clearly what the program does.

to decribe an algorithm. We believe that exposing students to more paradigms is more beneficial for gaining insight into computing itself than one specific approach.

1.3 Algorithm

The idea of algorithm seems to be clear, i.e. a reliable procedure feasible automatically, with no human interaction needed. However, specific descriptions differ and a precise formal definition is beyond the reach of the usual grammar-school student. To pronounce a process an algorithm the following basic conditions are required:

- *Explicit* (and finite) description of the procedure. Otherwise there is nothing to talk about.
- The procedure produces some output (and perhaps, but not necessarily, processes some input).
- The procedure is *finite*, regardless of the inputs.
- All the instructions are *elementary*, i.e. they are all known to everyone (everything) involved.
- The procedure is *deterministic*, i.e. the next step to do is always unambiguous. The consequence is that the output depends exclusively on the input.

We should mention *universality*, i.e. many different inputs of a kind can be processed by the same algorithm. It is also an often listed requirement for algorithm in Czech textbooks. It is however not an exact criterion and it may lead to some discussion (consider the algorithm to find π, or a trajectory to a specific comet, or *two* comets). It is a desired, but not a necessary feature.

Another desired, but not defining property is correctness. It is however not the property of a procedure itself. Correctness is bound to a procedure together with its task. Only for such a couple can we consider correctness. It does not affect the algorithmic nature of the procedure anyhow. The last property to mention is *efficiency*, i.e. resources consumption of the algorithm. Again, it is extremely important and desired, but not required for a procedure to be an algorithm.

The hereby described conception of algorithm allows many applications beyond the usual context of programming or symbol manipulation, yet it is not overly general and it provides a solid basis also for these traditional cases.

2 Why Do We Teach Algorithms?

Algorithm stands in the core of computer science (also in the sense of [4, 18]), some may even argue that it *is* the core. However, this fact alone is not a sufficient and understandable reason for students to embrace the term, and it should not be sufficient for the teacher either. The usual role of algorithms in computer-related education is that of a precursor for programming (e.g. [20, 8]),

but not always, see [7]. Most of our grammar-school students will not become programmers. However, the notion of algorithm can still remain very useful for their everyday life. In this section we review the reasons for teaching algorithms and the consequential goals, both in the context of Czech grammar schools.

2.1 Reasons

Reasoning about educational content at our grammar schools must start with FEP in hand. Examining the detailed description of individual key competences (also in [21], what is a related material to FEP), we find overlaps of problem solving and communication competences with cognitive skills employed during algorithm development and utilization [22]. These include abstraction, disciplined systematic and formal thinking, and, on the other hand, creativity and appreciation for elegance. The very natural opportunity to develop key competences is the strongest reason for teaching algorithms at Czech grammar schools.

We would also like to introduce a few more thoughts about algorithms contributing to a solid general education. Formulating an algorithm, i.e. a procedure, which is reliable and does not need personal attention anymore, frees that person to do something else, perhaps something more useful, innovative, or enjoyable. The procedure can be carried out by anyone (anything) else. As any algorithm is essentially an information, it can be multiplied very cheaply, leaving the number of simultaneous processes only to physical limitations. This phenomenon is apparently of high importance in any profession supposed to provide reliable services. The capability to create and communicate reliable instructions and procedures may be the crucial competitive advantage.

Apart to the ability to utilize this phenomenon, another reason to teach algorithms is its recognition in other fields. A good example is the legal system. The administration of justice must be based solely on the law, independently of any personal opinions. To achieve this goal, the law has to be written in a certain way – meeting criteria almost identical to those we seek in algorithms. Another example is medical diagnostics.

Algorithms and their influence can be and should be recognized in other fields as well. The story of "machines taking over people's jobs" has not yet come to an end. Financial markets can serve as a surprising example. A vast amount of transactions is controlled by computers. But they are not just executing instruments, they make the actual decision on what to buy or sell and when. Understanding algorithmics may allow students to better understand such changes and hopefully to foresee them soon enough to avoid the wrong career choice. They need to take an informed stand on which boundaries not to cross and which skills to develop in order to not allow the computers to become superior to them and to make further development safe, yet leave the actual work to computers.

Needless to mention, understanding algorithms is a necessary condition for understanding programming or other deeper aspects of computing.

2.2 Goals

The reasons described above lead us to formulate the following goals regarding algorithms[2]. Regarding basic notions and skills, students are to:

- understand the basic algorithmic properties well enough to recognize them (or their absence) on specific sample procedures, including appropriate reasoning and proofs (e.g. about finiteness);
- explain the consequences of each algorithmic property in practice (e.g. reliability, independence of the executor etc.);
- read and carry out an algorithm described in various forms (e.g. natural language, pseudocode with mathematical notation, flowchart, programming language in simple cases);
- find out what a given algorithm is probably good for;
- "debug" an algorithm: test it using appropriate inputs, follow the relation between input and output, isolate the flaw;
- modify, enhance or finish an existing algorithm;
- describe a known algorithm precisely in natural language, or choose another approach when appropriate;
- utilize basic concepts and control structures such as variables, decisions, loops, functions;
- utilize known basic techniques to find unknown algorithms (e.g. decomposition, systematic examination of all options etc.);
- know the basic criteria for algorithm comparison and compare the given algorithms;
- choose an appropriate basic instruction and estimate the algorithm's complexity class.

Regarding limitations of algorithms, students are to:

- understand that algorithms with exponential complexity are often practically useless;
- be aware of the existence of workarounds for practically unsolvable problems (heuristics, approximate solutions, probabilistic algorithms etc.);
- know that well-defined principally unsolvable problems exist, i.e. we do not have an algorithm for every task;
- recognize typical problems unlikely to have an algorithmic solution (e.g. problems with no solution, questions about the future without sufficient data, dealing with subjectivity or inconsistent definitions, self-reference);
- understand that computability does not depend on a specific machine or technology, computers are equally strong;
- understand the concept of Turing's test;
- recognize real world variations of Turing's test (e.g. CAPTCHA, teacher checking for student's true understanding);

[2] These goals do not constitute the whole course – we have chosen only what is related to algorithms.

– be aware of the advantages and disadvantages of both the human mind and a computer (including that a computer can only do what is algorithmic).

Regarding attitudes and opinions related to algorithms, students are to:

– appreciate abstraction utilization and the concept of black-box;
– prefer creative work and unpredictable environment;
– prefer to avoid stereotype and routine tasks and try to have a machine solve them instead;
– judge reasonably when to utilize an algorithmic approach a to a problem;
– judge reasonably when to prefer man over a machine for a task and vice versa.

3 The Notion of Algorithm in the Experimental Course

First we briefly describe the experimental course itself, so that we could discuss how algortihms are dealt with. The purpose of the course is to find out whether and how CS can be taught to grammar-school students and to provide some evidence for the related discussion. The course will introduce students to CS and provide them with a basic overview. They should be able to utilize the basic skills in real life situations. The course will also contribute to the development of the key competences mentioned above. All this is in accordance with other natural sciences at Czech grammar schools.

The course is structured into modules, which explicitly focus on different fundamental topics. Other topics are in the background, as can be seen in the example of algorithm. A module will last one month (four 90 minutes' lessons). The modules sometimes take longer if students lack necessary mathematical knowledge and skills[3]. More details on the course design can be found in [12]. We can proceed now to the description of each module and its contribution to algorithm study.

3.1 Preliminary Work

Our previous experience showed that the notion of algorithm is too abstract and meaningless to begin with. So we decided to leave the notion for later, when we already have some more experience with algorithmic processes and their advantages. Here we describe modules which precede the algorithm module.

In the *information* module students shall understand what "information" means, how we measure it and how we encode it efficiently. We use a number guessing game as a model, similar to [7]. Information amount measuring is based on Hartley's approach [9], i.e. decrease of possibilities[4]. The module also introduces the binary numeral system and the concept of decision (or encoding) trees

[3] We need to apply mainly logic, combinatorics, probability, also logarithms are helpful. Sadly, these are rather not popular among students.

[4] Shannon's classical entropy approach is out of reach for an average grammar school student.

as a visualisation of the process. Further details are published in [13]. From this paper's point of view, trees are early algorithm examples. Last to be mentioned is the concern about efficiency: we want to be done with our tasks fast, i.e. we want our trees to be shallow and the frequent nodes to be close to the root. Students reaalize the existence of minimal, average and maximal steps needed. The ideas relevant to algorithms are not named explicitly, but they are certainly addressed and dealt with.

The *graph* module introduces graphs as a strong tool for relation modelling. We add some terminology to unify the many examples already known. Students learn that even such visual structures can be encoded into zeros and ones, e.g. as adjacency matrix. Most of the time is spent examining Eulerian graphs (can a watchman walk the park paths efficiently?) and finding the rule to determine whether a graph is Eulerian. Students reformulate the rule into an instruction set. We show it to them in the form of a simple Python program for comparison. They can also experiment with it and modify it to become acquainted with basic programming concepts.

Again, basic efficiency issues are discussed. This includes asymptotic estimates, but also more down to earth questions: Shall I first calculate all the degrees, and then check that they are even, or can I be done sooner? Do I even need to actually calculate the degrees for seeing whether they are odd or even?

The *problem* module deals with general problem solving strategies, mostly adapted from [17]. They form a base for future algorithm developing. The underlying principles are examined using classical problems, such as wolf, goat and cabbage. Apart from heuristics like "chop it into parts", "take another point of view", "forget the unnecessary conditions" and "why exactly are you even doing this?", students realize the importance of defining the actual initial and the desired final state as well as the possible changes to each state. This leads directly to state space and its systematic traversing.

3.2 The Algorithm Module

The next module deals with algorithms explicitly. We provide a more detailed description here. The whole module begins with a task to describe sufficiently some seemingly easy procedure, such as to tie a knot or fold a paper into a cap. The sufficient description will not allow the fellow student to do anything differently from what the author intended. It turns out, that almost no one is capable of instructing others properly – not even on the fly, with direct feedback. This discovery is sufficiently striking for most of the students to try to do better and investigate the related theory.

The common and desired properties of already known procedures are examined. Based on this, the notion of algorithm itself is introduced. Then each property gets some attention: what is it, what does it cause, and how do we recognize them in given examples. This is also an opportunity to show a fairly abstract tool, the idea of a decreasing measure for proving finiteness. We use real world problems such as cooking or guitar tuning and classic CS algorithms

such as bubble-sort. Correctness and efficiency are discussed, the latter in more detail in the next module.

The last area to cover touches the fundamental limitations of algorithmic procedures. We explain the idea of the halting problem and make it feel more real referring to virtualization. With a few more examples (mostly from logic), students will realize that the cause of our trouble is self-reference. Other types of unsolvable problems are examined: we cannot construct a triangle with two right angles in a plane, we can not not check the equality of two real numbers, we can not predict quantum mechanical phenomenons, we cannot assure everyone of a happy life (unless we consider the trivial solution[10]).

We can see that there is virtually nothing about algorithm development itself. These issues are distributed among all the modules, where algorithms are developed. This approach is used also for other traditional topics (e.g. debugging).

3.3 Following Extensions

The concept of algorithm is not left behind during the other modules. The *efficiency* module lets students compare algorithms for the same task and provide some tools for that, such as the idea of basic step for considering large inputs. We examine and compare linear and binary search, sorting algorithms (e.g. already known bubble-sort with merge-sort) and other examples. Based on this experience, some basic heuristics for optimization are shown, such as "reuse your results" or "prepare your data". In this module also the existence of practically useless algorithms and practically unsolvable problems is shown. Students learn how to roughly distinguish the uselessly slow algorithms.

The next module deals with *evolutionary algorithms* as an illustration of some advanced computing science to complete the overview. The basic framework is explained and the teacher provides students with a simple Python code to search Hamiltonian paths. They experiment with it, tweak some parameters and discover issues like convergence, optimal population size, keeping the best individuals through generations, exploration vs. exploitation etc. All this serves as a challenge to the known and safe conceptions. This will help students to find a place for computing in the rest of the world and prepare them for the last module.

The last module is about links to *humanities*. The core topic is the relation between human mind and computer. The concept of the Turing test is introduced. Students will take a stance on the topic and define the fundamental difference between humans and machines. Their points of view are supported by the hands-on experiences with computing gained during previous modules. This module is not built like the others, i.e. solving a problem of some computational nature. Students do not produce algorithms and quantitative analysis, they analyze and discuss arguments and produce essays and presentations instead.

4 Criteria for Algorithms in Natural Language

In this section we describe the criteria used to evaluate natural language descriptions of algorithms. First we add a few thoughts to the whole idea, proposed and used e.g. in [14, 15, 19]. In their lives, most of the students will deal exclusively with instructions written in natural language – carrying them out, assessing them, and hopefully also creating them. We naturally expect that their skills can get better thanks to mathematics, programming etc. But we find the direct and explicit approach more promising.

One of the first advantages is motivation. Students usually see more sense in enhancing the related skills than in the case of e.g. programming. Furthermore, they see very well, how wrong their descriptions are. If the procedure does not work, students can blame neither the computer nor "the obscure programming language". The advantage, which is not obvious at first, is their *growing* sympathy for formal systems of various kinds. Writing precise descriptions in natural language is difficult. This fact (supported by personal experience) makes them appreciate clear formal systems (even programming itself), which simply eliminate the possibility of many mistakes just by design. For the same reason they appreciate simple abstract models as practical under certain conditions.

We share a few more details on how we evaluate students' results. First of all, we try not to evaluate them. As in the whole course, we try to design situations, in which the result quality naturally rewards (or punishes) the student. For example, a slow algorithm literally takes its time. With algorithms in natural language, a fellow student is usually a fair judge. A misunderstanding is exactly the kind of natural "punishment" we are looking for. For providing a more formal feedback, we found inspiration in language education. Essay evaluation often breaks down to a set of criteria [16]. The evaluator assesses only the individual criteria fulfillment. The information on each criterion serves as valuable feedback to the student and allows him to advance somewhat systematically.

To employ this idea, we need a set of appropriate criteria. This work starts in the algorithm module. The criteria are continuously refined during the rest of the course. Students evaluate each other's work anyway, so we opened the process for them. This approach has clear advantages over just stating the already developed criteria. Most of all, criteria based on students' actual needs make more sense to them, they feel a sort of ownership, and they are more willing to meet the criteria. The teacher can of course make suggestions. His work, however, consists mostly of managing the process. This includes encouraging students to differentiate between necessary and satisfactory conditions and to formulate them precisely. Another aspect is generality of the criteria. We have no preferred computational model, so the criteria will work as universally as possible. Development of criteria for assessing algorithm description quality has a nice recursive touch: the criteria will be written meeting themselves.

Here is an example of a criteria set. As description flaws were similar, similar criteria were devised in both groups.

- Language: used correctly.
- Specification: clear description of allowed inputs and intended purpose.
- Semantics: clear nature of all text: instructions, asserting statements etc.
- Terminology: all used words are common knowledge or well defined.
- Instructions: do not allow alternative interpretations.
- Work flow: always clear where to continue.
- Decisions: all possible outcomes explicitly covered.
- Unexpected situations: non-existent, every possible situation is anticipated and covered[5].
- Loops and recursion: explicitly denoted body, clear ending condition, clear routine to change variables (or whatever material the loop works with).
- End: Explicitly stated, together with clear output.

For estimating the quality of a description which satisfies these necessary conditions, we have developed *recommendations*. It is usually helpful to follow them, but it is not necessary. Just as the criteria above, these recommendations are hardly surprising for anyone familiar with programming. But they are quite a discovery for the beginners.

Here is a sample list of such recommendations:

- State clearly the purpose of the algorithm.
- Put one instruction on a line, do not write in paragraphs.
- Use indentation to indicate structure.
- Number or label the lines, refer to them.
- Do not avoid repetition of words, use one word for one meaning.
- Do avoid repetition of instruction sequences. Define loops or functions instead.
- Use the concept of a variable, when appropriate.
- Comment on the algorithm, explain why it works and what the meaning behind individual instructions is.
- Each instruction must be detailed enough to prevent misunderstanding. Do not expect kind cooperation. Everything that is to be done must be said.

To illustrate the criteria we show a very wrong example. It is synthesized from the work of our students to show multiple flaws in one sample. The described procedure is supposed to add 1 to a binary number on the input. Knowing how it should work we can see that the student perhaps means well. He himself is unlikely to make a mistake while adding 1 to a binary number. However, the description itself is practically useless for anyone who does not know what to do from some other source. With the help of evaluation criteria, the student can systematically identify the flaws and correct his algorithm.

Example of a procedure described in natural language:

```
In binary numeral system we have only 1 and 0. If we run into
a 1 while adding 1, we move over to the next digit. If there
is a 1 again, we move over again, and we continue this, until
we find a 0, which we overwrite to 1.
```

[5] Strictly taken, this is impossible without a proper computational model definition or a wide range of prerequisites.

5 Evaluation

We tested the above described approach in two optional seminars (i.e. small groups of students who chose the subject among a wider offer). The concept underwent some adjustments during the school year. Also the groups were too small and proper control groups were not available. The obtained data is therefore almost useless and difficult to interpret seriously. However, we will share the main findings briefly.

The level of motivation of the students was higher than in other years (and other students) with more traditional approaches. Students have explicitly reported that they could see links to their lives, both present and future, even though not necessarily related to computers. They also found dealing with their natural language more challenging than other options.

We have not met all the defined goals. Some parts turned out to be too demanding (mostly finiteness proofs and the concept of state space). On the other hand, some turned out to be accepted surprisingly well (we had been especially apprehensive about the last two modules, but the results were good). Most of the topics worked as expected, namely asymptotic complexity is perhaps a borderline for many students.

6 Conclusion

We have shown a novel point of view to algorithms in general secondary education. It aims to match the main goals found in the Czech curriculum, mainly in developing key competences to solve problems and to communicate. On the other hand, the traditional focus on programming and computers themselves is suppressed (though not removed).

We argue that an algorithm described in natural language is a valuable tool in computing courses in general education. We have shown a criteria-based evaluation tool and explained its functions.

We observed a rise of motivation among students and improvement in their basic skills, such as precision in communication. We intend to investigate the effects of the new approach on the key competences themselves. However, that will be very challenging, for their description is far from algorithmic.

References

1. Framework Education Programme for Secondary General Education (Grammar Schools). Výzkumný ústav pedagogický v Praze, Praha (2007)
2. Aho, A., Ullman, J.: Foundations of computer science. Computer Science Press, New York (2000)
3. Bell, T., Curzon, P., Cutts, Q., Dagiene, V., Haberman, B.: Introducing students to computer science with programmes that don't emphasise programming. In: Proceedings of the 16th annual joint conference on Innovation and technology in computer science education - ITiCSE '11. ITiCSE '11, vol. 5, p. 391. ACM Press, New York, New York, USA (2011), http://doi.acm.org/10.1145/1999747.1999904

4. Bruner, J.S.: The process of education, vol. 115. Harvard University Press (1977)
5. Dagiene, V., Futschek, G.: Bebras International Contest on Informatics and Computer Literacy: A contest for all secondary school students to be more interested in Informatics and ICT concepts. Proc. 9th WCCE (2009)
6. Denning, P.J.: Computing is a natural science. Communications of the ACM 50(7), 13 (Jul 2007)
7. Fellows, M.R., Bell, T., Witten, I.: Computer Science Unplugged - offline activities and games for all ages: Original Activities Book. Computer Science Unplugged (1996)
8. Gal-ezer, J., Harel, D.: Curriculum and Course Syllabi for a High-School Program in Computer Science. Computer Science Education 9, 114–147 (1999)
9. Hartley, R.V.L.: Transmission of information. Bell System techn. Journal 7, 535–563 (1928)
10. Holan, T.: Všichni lidé šastni (2010), `http://drupal.geometry.cz/k54`
11. Hromkovič, J., Steffen, B.: Why Teaching Informatics in Schools Is as Important as Teaching Mathematics and Natural Sciences. In: Proceedings of the 5th international conference on Informatics in Schools: Situation, Evolution and Perspectives. pp. 21–30. ISSEP'11, Springer-Verlag, Berlin, Heidelberg (2011)
12. Lessner, D.: Computer science curriculum proposal for Czech grammar schools. In: Zborník príspevkov. pp. 99–104. ITAT11, PONT s. r. o., Sea, Slovakia (2011)
13. Lessner, D.: Information Theory on Czech Grammar Schools: First Findings. In: Knobelsdorf, M., Romeike, R. (eds.) Pre-proceedings of the 7th Workshop in Primary and Secondary Computing Education (WiPSCE). pp. 139–142. Hamburg (2012)
14. Miller, L.A.: Natural language programming: styles, strategies, and contrasts. IBM Syst. J. 20(2), 184–215 (1981), `http://dx.doi.org/10.1147/sj.202.0184`
15. Murphy, L., Fitzgerald, S., Lister, R., McCauley, R.: Ability to 'explain in plain english' linked to proficiency in computer-based programming. In: Proceedings of the ninth annual international conference on International computing education research. pp. 111–118. ICER '12, ACM, New York, NY, USA (2012)
16. O'Donovan, B.: Improving students' learning by developing their understanding of assessment criteria and processes. Assessment and Evaluation in Higher Education 28, 147 (2003)
17. Polya, G.: How to solve it: A new aspect of mathematical method. Princeton University Press, 2 edn. (1957)
18. Schwill, A.: Fundamental Ideas: Rethinking Computer Science Education. Learning & Leading with Technology 25(1), 28–31 (1997)
19. Simon, B., Chen, T.y., Lewandowski, G., McCartney, R., Sanders, K.: Commonsense computing: what students know before we teach (episode 1: sorting). In: Proceedings of the second international workshop on Computing education research. pp. 29–40. ACM (2006)
20. Tucker, A., Deek, F., Jones, J., McCowan, D., Stephenson, C., Verno, A.: A Model Curriculum for K-12 Computer Science: Final Report of the ACM K-12 Task Force Curriculum Committee. Computer Science Teachers Association, New York, second edn. (2003)
21. Šlejšková, E.: Klíčové kompetence na gymnáziu. Výzkumný ústav pedagogický v Praze, Praha (2008)
22. Wing, J.M.: Computational thinking. Communications of the ACM 49(3), 33 (Mar 2006)