# Teaching Programming and Algorithmic Complexity with Tangible Machines

Tobias Kohn[1] and Dennis Komm[2]

[1] University of Cambridge, UK
tk534@cam.ac.uk
[2] Department of Computer Science, ETH Zurich, and PHGR, Chur, Switzerland
dennis.komm@inf.ethz.ch

**Abstract.** Understanding the notional machine that conceptually executes a program is a crucial step towards mastery of computer programming. In order to help students build a mental model of the notional machine, visible and tangible computing agents might be of great value, as they provide the student with a conceptual model of who or what is doing the actual work.
In addition to programming, the concept of a notional machine is equally important when teaching algorithmic design, complexity theory, or computational thinking. We therefore propose to use a common computing agent as notional machine to not only introduce programming, but also discuss algorithms and their complexity.

**Keywords:** Python · turtle graphics · complexity · efficiency · notional machine

## 1 Introduction

Computer Science is in the process of becoming an accepted and integral part of general education. This is mostly in the form of an introduction to programming. However, the common goal of computer science programmes is "computational thinking" and a fostering of problem solving skills, not necessarily the mastery of a programming language. Hence, introductory programming is only a first step on a path that is supposed to lead to computational thinking.

Unfortunately, programming has turned out to be difficult to master, and few students advance to a level where they can start expressing and implementing algorithmic thinking and design. As a consequence, many courses on K-12 levels focus on just the programming. On the other hand, another approach is taken by the "Computer Science Unplugged" programme of Bell et al. [4,5,3,2] or Gallenbacher [9], which does away with the programming altogether, and introduces algorithms without the need for either programming or a computer at all. Eventually integrating both approaches into a curriculum is, of course, a good and fruitful idea. Yet, if programming courses cannot advance to a level where algorithmic principles and computational thinking can be taught, what is the actual benefit of programming education?

Despite popular belief, we are convinced that the mastery of a programming language is not a necessary prerequisite to discuss and implement various algorithms. Rather, learning to program offers a great opportunity to connect to the concepts of algorithms, complexity theory, and eventually lead to the goal of computional thinking even without the need of having introduced the full range of programming structures. The key aspect of our approach is to shift attention from the syntax of the language to the concept of the computing agent or notional machine. In short, the crucial thing is not whether to use a "for"- or a "while"-loop, but rather how the machine can do repetition and iteration to solve a task or problem (cf. [11]).

The approach presented in this article has been used in outreach programmes, and several workshops with the explicit goal of introducing the audience to the basics of complexity theory within a very limited amount of time (cf. [14]). So far, we have not had the opportunity to employ the concepts in large scale teaching, and we therefore do not include any statistics on its actual usefulness but leave this to future research.

## 2 The Notional Machine

One of the greatest challenges a programming novice faces is to understand the *notional machine*. Originally, du Boulay named the notional machine as one of five difficulties in learning to program, alongside syntax, semantics and other issues [7].

The notional machine is a model of the machine that *conceptually executes the program code*. In principle, it answers the question: what does a machine need in order to understand, follow and execute the program code? As such it differs from the actual implementation on any level, and is therefore not bound to the physical computer, or a virtual machine. The notional machine is primarily determined through the programming language, libraries and frameworks used. An excellent treatment of the notional machine can be found in Sorva [26].

Getting a good understanding of the notional machine is crucial for programming. When we regard programming as communication between man and the machine, with the programmer being the sender of the message (the program code), then the notional machine is the receiver of the message. It determines the effect of a given program code, and answers accordingly to the programmer. Of course, successful communication can only take place with a working model and understanding of the second party involved, i. e. the notional machine.

### 2.1 The Notional Machine for Algorithms

The concept of the notional machine is not limited to programming, but is equally important for algorithms in a more general setting. In the discussion of algorithms, the notional machine is frequently not made explicit, but only assumed implicitly. When discussing searching in graph theory, for instance, we usually implicitly assume a notional machine that can visit one vertex or node of the graph at any given time, and travel along edges between the vertices. Hence, in our discussion of search algorithms, we imagine to be "the mouse trapped in the maze" rather than having a bird's-eye view of the entire maze as a whole. This also means that we cannot "look down an aisle" and determine that it is an impasse to be skipped in our search.

Occasionally, we do make the notional machine explicit. For instance, when we classify *quicksort* as a "comparison sort". Or when we pose a specific problem in the form "You only have a balance scale to compare items. . . ". Note that, in an educational setting, we might be required to go to great lengths to force students to work with a specific notional machine, and keep them from "cheating" (they might, for instance, determine the relative weight of objects without requiring the scale at all).

Taking the view of complexity theory, we might argue that all algorithms are executed by a Turing machine. However, when effectively designing and working with algorithms, there is seldom the need to take such a drastically abstract approach. Instead, the notional machine executing the algorithm might already "know" how to add or subtract two numbers. Yet, based on our physically available machines, the numbers for addition could be limited in size, requiring special algorithms for large numbers. Thus, depending on the model of the machine, we might wish to factor in the size of the numbers when computing the runtime of such operations, say.

It is imperative that we make the notional machine explicit when teaching specific algorithms. For instance, most programming languages and computing systems offer a function to compute the square root of a number. It can then seem useless, even a waste of time for students to discuss Heron's algorithm for computing the square root – obviously, this is already there and available.

In the context of problem solving skills, the notional machine can be seen as the toolbox that is available to solve a given problem. In education, we then intentionally limit the toolbox, of course, to prompt and foster creativity, and to effectively help the students train their problem solving skills "in the small".

### 2.2 The Difficulty of the Notional Machine

The computer in its entirety is an incredibly inappropriate notional machine to start with. On the one hand, almost no part of a modern computer can be directly inspected. Unlike a steam engine, there are no moving parts that can be observed and understood. The computer is a black box. Various studies have found that, indeed, invisible parts in programming are one of the greatest source of difficulties and misconceptions (cf., e. g., Sorva [26]).

There are three possibilities to address the situation.

*First*, we can use metaphors like the famous box-metaphor used for variables, to help the student form a picture of the notional machine. Unfortunately, such static metaphors tend to tacitly introduce misconceptions as students derive inappropriate analogies from the metaphors. The box metaphor for variables, for instance, suggests to some students that a variable can store an arbitrary amount of values.

*Second*, we can use visualisations to provide a window into the black box. Debuggers allow to trace a program step by step, and display the current value of variables during program execution. However, while such visualisations show the machine's state between two steps, the actual working remains hidden. It is like taking pictures of a steam engine, and trying to understand its working without actually seeing the parts move.

Besides displaying the execution of a student's program code, visualisations are also popular when explaining algorithms. For instance, the sorting of an array of integers might be visualised by bars, where each bar represents a value inside the array through its height. Even though the bars might be moving around during the visualisation, it is important to understand that such a visualisation does not display the algorithm itself, but rather its effect on the data.

Hence, visualisations do not automatically foster a better understanding. Instead, students must be trained in using and understanding the visualisations themselves (cf., e. g., [15,18,27]).

*Third*, we can use a simpler machine to work with. Instead of programming the abstract "computer" with variables, memory, etc., we have the students program a simple, tangible, and understandable machine. This machine can either be physical or simulated. The crucial part is that no parts of it are hidden, and that the students can build a conceptual model of what it can do and how it works.

In reality, a simple machine might not be enough to truly act as the notional machine for the entire program. However, the simple machine could act as the *computing agent*. With "computing agent" we refer to a relatable and tangible entity that is the receiver of the commands in the program code, and acts upon them. In fact, the idea of a computing agent has been proposed in various forms for decades, and a plethora of different systems exist (cf, e. g. Kelleher and Pausch [12]). However, note that there is a danger in making the computing agent too "human", or too complex.

For our teaching, we use a *turtle*, as found in on-screen turtle graphics, as the computing agent. The turtle executes commands such as `forward(20)`. However, the program code occasionally contains elements that escape the strict turtle-as-agent metaphor. Examples for this might be `forward(100/5)` with a computation in the program code (the turtle metaphor does not explain how such a computation is performed), or control structures such as loops.

Even though the computing agent is not a full blown notional machine, it seems to be sufficient for educational purposes, as it helps the students to form a mental model of *how the machine works* and *what it can do*. It is, in fact, elucidating to realise that practically all successful education environments provide a computing agent (or several) as a core feature (cf., e. g., Logo [21], Greenfoot [13], Kara [10], Scratch [22], Alice [23]). We might even hypothesise that large parts of the success can be attributed to the computing agent rather than, say, what programming language is used (for instance, Lewis [16] found little difference between Logo and Scratch).

## 3 Turtle-Graphics

The turtle, seen as a machine, has a small set of clearly defined and understandable basic commands. Most importantly, the semantics of these commands is concrete: there is no need for variables, or hidden state to explain this machine (even though the implementation typically makes heavy use of such abstractions). Instead of abstract variables to capture state, we can use, for instance, the turtle's colour, or the colour of its pen.

### 3.1 The Turtle as Computing Machine

We regard the turtle as a machine with two data registers (its proper colour and the pen colour, respectively), and one address register (its location on the canvas). The canvas itself acts as storage or memory. The turtle's orientation has the role of a "direction bit" to control the direction of commands such as `forward()`.

According to our model, we provide basic commands to read a value (which is always a colour) from the storage (canvas) to a register, store a value in the storage, copy a value between the registers, or swap the values of the two registers or a register with the storage. If we denote the turtle's colour value by $r_T$ ($r$ for "register"), the colour value of its pen by $r_P$, and its position (address) by $r_A$, we can express the turtle commands in a more abstract manner as follows:

| | |
|---|---|
| set_pos($v$) | $r_A \leftarrow v$ |
| set_color($v$) | $r_T \leftarrow v$ |
| set_pen_color($v$) | $r_P \leftarrow v$ |
| dot() | $r_P \rightarrow [r_A]$ |
| swap_color( ) | $r_T \leftrightarrow [r_A]$ |
| swap_turtle_colors() | $r_T \leftrightarrow r_P$ |
| assume_pixel_color() | $r_T \leftarrow [r_A]$ |

In principle, the canvas is a two-dimensional array of pixels. With modern devices, however, the pixels are far too small, so that, for most purposes, we use *dots* with a diameter similar to the size of the turtle.

*Comparing Colours.* There are various colour models and spaces, such as *RGB*, *CMYK*, *HSB*, etc. The prevalent colour model in programming is probably the *RGB*-model, and colours are usually accessed as 24-bit integer values.

Since our algorithms are based on comparing colour values, we need to impose an ordering on a usually three- or even four-dimensional space. We could, of course, just use the 24-bit *RGB*-values, along with the usual ordering of integers. However, we found the results to be aesthetically more pleasing when switching to an *HSB*-model, where the primary key for comparison and ordering is the *hue* of a colour value.

*Further Extensions.* In addition to the basic operations on the turtle's state, we have added two more concepts to our implementation: first, the turtle can determine not only the value of the pixel underneath the turtle, but also the value of the pixel directly in front of the turtle. Second, some dots are subject to "gravity" and will fall down to the bottom of the screen, unless they hit another dot (exposed through the command `drop_dot()`).

## 4 How to Measure Efficiency

For a theoretical computer scientist, the term "efficient algorithm" commonly refers to an algorithm with polynomial time complexity. In a broader sense, it depends on the context what is meant; "efficiency" may at times also correspond to memory usage, energy consumptions, code maintenance, or scalability. However, our current efforts address algorithms' time complexities, i. e., running time as depending on the length of a given input.

Without a formal introduction to computer science, what usually first comes to mind is to use some absolute measurements such as milliseconds. For the purpose of discussing algorithms, however, this is unsatisfactory, because the absolute time taken depends on many things that are independent of the algorithm used, such as the machine it is run on, or the programming language. What makes more sense is to count the number of elementary operations carried out by the machine. This, of course, depends on the notional machine, and its set of "elementary" operations. In fact, counting elementary operations is a prime example of requiring a notional machine.

*Different Classes of Complexity.* Measuring the runtime of an algorithm is in itself not an adequate measure for the quality of an algorithm. In the overall picture, a particularly good algorithm distinguishes itself not by handling a specific instance of the problem well, but by scaling well with the size of the problem instance.

In the context of single-number-based algorithms, we take the number of bits (or digits) of the input number as a measure for the "problem size". For a list of values, we take the number of values in the list to be the appropriate measure. We would then like to assess how much longer the algorithm needs if we add another bit to the number, or value to the list, respectively.

To give a hands-on example, consider the problem of assessing the divisibility of an integer $N$, represented by $n$ decimal digits. We can then give three examples of different complexity:

– We can check if $N$ is divisible by 2 by checking the last digit of $N$, irrespective of the size of $N$. Hence, assessing if a number is odd or even can be achieved in constant time.
– In order to check if $N$ is divisible by 3, we can build the sum of all digits, and then check if the resulting sum is divisible by 3. This is commonly known by all students. Building the sum of all digits depends on the number of digits $n$. If we add one more digit, we have to add one more number. Hence, time complexity for this problem is linear in $n$.
– To check if an integer $N$ has any non-trivial divisor at all, much more work is needed. As a first approximation, we can argue that we need to check more or less each integer between 2 and $N - 1$, to assess if it is a factor of $N$. Adding another digit usually multiplies the required work by a factor of 10. Time complexity is therefore exponential and can be expressed as $10^n$.

*Efficiency in Programming.* The actual implementation of an algorithm may seem to differ significantly from a theoretical presentation of the same algorithm. Switching to an implementer's perspective bears the danger of focusing too much on the efficiency of implementation details, as opposed to the efficiency of the underlying algorithm as discussed above. This issue can be understood in part originating from a change of the underlying notional machine: the notional machine of any given programming language hardly ever coincides with the one used to discuss the algorithm in a theoretical framework.

To smoothen the transitions between a theoretical discussion of an algorithm, and its implementation, it is desirable to use a unified model. Turtle graphics as presented here offers an opportunity for such an unified model. The efficiency can then be described in turtle operations, which in turn, have direct correspondence in code, and on the screen during program execution.

## 5 Primality Testing

In this section, we describe two different approaches of how to sharpen the student's view on complexity theory using the example of primality testing.

### 5.1 Geometric Approach

Testing for primality can be seen as a geometric problem. Given a number of dots $N \geq 2$, we try to arrange the dots in the shape of a rectangle. If this succeeds only when all dots are in one column or one row, respectively, $N$ is a prime number; for instance, 15 dots can form a rectangle with dimensions $3 \times 5$, whereas there is no rectangle with dimensions $k \times m$ for 17 dots with $1 < k, m < 17$.

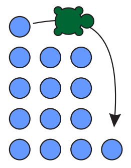

**Fig. 1.** Given a number of dots, the turtle takes dots from the top, and drops them into the column on the right, trying to balance all columns and thereby create a rectangle.

Starting a single column of $N$ dots on the very left of the screen, the turtle tries to rearrange the dots so that they form a rectangle. To that end, the turtle picks up the top-most dot, and drops it into the column to the right. During this process, the turtle works its way from left to right, and jumps down to the next row if no dot remains on the top-layer. If the column to the right has been filled to the top, such that all columns have equal height, the number $N$ is (literally) obviously divisible. Otherwise, the turtle starts to drop the dots into the next column, trying to balance all dots with one more column. By adding a new column to the right whenever needed, the turtle continues trying to balance the dots into the existing columns until it finds a solution (which always exists in the form of a row of height one).

Program 1 in the appendix shows the implementation of a geometric primality testing algorithm, using the dot-based turtle graphics (see Figure 1).

Given this geometric approach, it is indeed evident that each possible arrangement is actually tried twice. Divisibility by two, for instance, is once tried using two columns, and once using two rows (if the number is a prime). It can directly be reasoned that the algorithm can stop as soon as the number of columns exceeds the number of rows. In a more abstract way, the turtle has to test only factors up to $\lfloor\sqrt{N}\rfloor$, because once it is found out that there is no rectangle with dimensions $k \times l$ with $k \leq \lfloor\sqrt{N}\rfloor$, all possible $k, l$-combinations to follow have already been tested with $k$ and $l$ being switched.

### 5.2 Semi-Geometric Approach

The geometric approach presented above has various limitations, particularly with respect to possible optimisations. We therefore describe a semi-geometric variant, which allows us to introduce various optimisations.

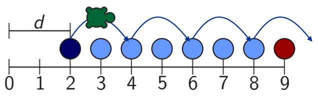

**Fig. 2.** For each possible factor $\boldsymbol{d}$ with $\mathbf{2} < \boldsymbol{d} < \boldsymbol{N}$, the turtle jumps $\boldsymbol{d}$ steps, trying to reach the $\boldsymbol{N}^{\text{th}}$ dot (coloured red in our implementation). The dot at position $\boldsymbol{d}$ is removed, and the turtle starts again at position $\boldsymbol{d} + \mathbf{1}$.

The basic algorithm is shown in Program 2 in the appendix. Based on the idea of the sieve of Eratosthenes [20], we visualize a number $N$ as a row of $N-2$ blue, and one red dot (we have to omit the dot representing "1" since we do not want to test whether $N$ is divisible by 1). The turtle is then set on a dot $d$ and jumps forward in steps of $d$ dots. If the turtle hits the red dot, $d$ is a factor of $N$. Otherwise, the dot representing $d$ is removed and the turtle checks the next possible factor.

In contrast to the true sieve of Eratosthenes, Program 2 does not cross out any multiples of a given number $d$. Hence, the turtle checks each integer up to $N$.

A first optimisation can be introduced by crossing out multiples of any tested factor $d$, implementing Eratosthenes' sieve properly. In our semi-geometric variant, this means that the turtle has to remove any dot it encounters. In our actual implementation, this could be done by colouring the dot in a darker blue (against a black background).

## 6 Sorting

As a second example to demonstrate the power of turtle graphics to illustrate the efficiency of certain algorithms, we use the standard problem of sorting. Due to space restrictions, we can only include a version of *Minsort* here.

As setup for our algorithms, the turtle starts by drawing a column of coloured dots in the middle of the screen. The colours are chosen at random, based on the HSB-model, where only hue varies. Before a sorting algorithm starts, the turtle is placed at the bottom of the column of coloured dots, heading up, and having no colour of its own.

The turtle picks up the first (bottom-most) dot it encounters, assuming its colour and removing the dot from the screen. Subsequently, the turtle moves through all remaining dots, swapping its colour with any dot, whose colour is to be found "less" than the current colour. Once the turtle reaches the top, it moves slightly to the right and drops the coloured dot into a new column, before returning to the initial position (called "origin" in Program 3) at the bottom of the original column.

As before, the algorithm is simple enough so that no variables are needed, beyond the turtle's own colour. In other words, the entire algorithm is directly visible on screen with no hidden state.

In addition to Minsort, we have also implemented Bubblesort, Quicksort, and Mergesort in our turtle framework. However, the limited capabilities of the turtle adds additional issues, and complexities. Mergesort, for instance, lends itself very well to a framework with a total of three turtles cooperating. This, in turn, requires more programming background from the students.

For a manageable implementation of Quicksort (see appendix B), we had to make use of a variable, thereby going beyond the initial framework of no hidden state. Instead of discussing the Python code, we have discussed the visualisation with the students, who were often quick to point out that the presented algorithm would not be optimal if the initial array was "already sorted in reverse".

## 7 Related Work

In order to make algorithms and complexity theory accessible to students in a K-12 context, CS unplugged programs, such as those by Bell et al. [3] or Gallenbacher [9], have been developed, and found widespread success. Moreover, Bell et al. [2] argue that understanding complexity theory, even on a rather intuitive level, gives high-school students a very good picture on what computer science is about, and makes them less likely to choose studying it for the wrong reasons.

Turtle graphics is often associated with the Logo programming language; see, e.g., Thornburg [30]. Today, however, most educational programming languages provide libraries for turtle graphics, including Python. Caspersen and Christensen [6], for instance, present an access to object-oriented programming based on turtle graphics. One of the most essential aspects of turtle graphics are its immediate visual feedback, and the support for a teaching style that goes from the concrete towards the abstract [6].

One of the most extensive treatises on turtle graphics is offered by Abelson and DiSessa [1]. In the early tradition of turtle graphics, their discussion, however, has a strong focus on geometry, mathematics, as well as simulations.

According to du Boulay, building a conceptual model of the notional machine is one of the five difficulties a novice programmer faces. In the light of his and subsequent research, as presented below, we would argue that a comprehension of the notional machine is, indeed, one of the crucial prerequisites to succeed in learning to program.

The notional machine is often implicit or hidden from the novice programmer. This invisibility of the machine's operation has been mentioned as a hurdle and source of misconceptions several times. Sorva [26], for instance, writes that "Many misconceptions, if not most of them, have to do with aspects that are not readily visible, but hidden within the execution-time world of the notional machine".

Sorva [26] reports that conceptual models, which might just be "a simple metaphor or analogy, or a more complex explanation of the system", are not only useful tools, but help improve the understanding and performance of the students (cf. also Schumacher and Czerwinski [24]). He advocates the use of visualisations and metaphors, in particular to "concretize the dynamic aspects of programs." In addition, Sorva points out that an active engagement by the students is, in fact, more important than the particular visualisation technique used.

We see this in accordance with our approach to use a concrete, and visually oriented notional machine to teach programming. The crucial part, however, is that we do not use a visualisation of an underlying machine, but rather a visually tangible computing agent, as this means that students actively engage with the "visualisation" instead of merely consuming it.

Finally, it is interesting to compare these findings with the neo-Piagetian theories as reported by Lister [17], or Lister and Teague [29]. Lister argues that novice programmers initially work on a preoperational stage (or even sensimotor stage), where "they can trace code accurately, but they struggle to reason about code." [29] In particular, his findings indicate that students use *concrete values* to trace and understand code, and that this is a necessary step in the development of higher abstraction and reasoning skills.

Hence, using concrete and directly visible values during program execution might address students better on the level of their respective stage in reasoning, and thereby help them progress in their learning. Further studies in this direction are certainly warranted.

## 8 Conclusion

Even though programming is an important part of Computer Science Education, it should not be limited to programming alone. Computational, or algorithmic thinking needs to be treated as an equally important part. Computer Science Unplugged programmes offer a great way to introduce computational thinking early on, without extensive programming prerequirements.

In this article, we have presented an approach with focus on the notional machine, and computing agent. Education in both programming, as well as algorithmics could benefit strongly from explicit incorporation of assumptions about the underlying computing agent. Instead of visualising just states of the machine, we can then visualise the actual work of the agent itself, and provide a firmer framework for learning metaphors. For beginning classes, we have used an enhanced turtle as the computational agent, with so far very positive experience.

Future research, as well as classroom experience will be needed to properly evaluate the applicability, and validity of our approach.

## References

1. H. Abelson and A. DiSessa. *Turtle Geometry.* MIT Press, 1981.
2. T. Bell, P. Andreae, and L. Lambert. Computer science in New Zealand high schools. In *Proc. of ACE 2010*, 2010.
3. T. Bell, I. Witten, and M. Fellows. CS unplugged – Computer science without a computer, http://csunplugged.org. Last visited on September 17th, 2017.
4. Tim Bell, Frances Rosamond, and Nancy Casey. *Computer Science Unplugged and Related Projects in Math and Computer Science Popularization*, pages 398–456. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
5. Timothy Bell, Jason Alexander, Isaac Freeman, and Mick Grimley. Computer science unplugged: school students doing real computing without computers. 13, 01 2009.
6. M. E. Caspersen and H. B. Christensen. Here, there and everywhere – on the recurring use of turtle graphics in CS1. *Proc. of ACSE 2000*, 2000.
7. Benedict Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.
8. Benedict du Boulay, Tim O'Shea, and John Monk. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, 14(3):237 – 249, 1981.
9. J. Gallenbacher. *Abenteuer Informatik.* Springer, 4th edition, 2017.
10. W. Hartmann, J. Nievergelt, and R. Reichert. Kara, finite state machines, and the case for programming as part of general education. In *Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, HCC '01, pages 135–, Washington, DC, USA, 2001. IEEE Computer Society.
11. J. Hromkovič, T. Kohn, D. Komm, and G. Serafini. Combining the power of Python with the simplicity of Logo for a sustainable computer science education. In *Proc. of ISSEP 2016*, LNCS 9973, pp. 155–166, 2016.
12. Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137, June 2005.
13. Michael Kölling. The greenfoot programming environment. *Trans. Comput. Educ.*, 10(4):14:1–14:21, November 2010.
14. D. Komm and T. Kohn: An introduction to running time analysis for an SOI workshop. *Olympiads in Informatics* 11:77–86, 2017.
15. A. W. Lawrence, A. M. Badre, and J. T. Stasko. Empirically evaluating the use of animations to teach algorithms. In *Proceedings of 1994 IEEE Symposium on Visual Languages*, pages 48–54, Oct 1994.

16. Colleen M. Lewis. How programming environment shapes perception, learning and goals: Logo vs. scratch. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 346–350, New York, NY, USA, 2010. ACM.
17. Raymond Lister. Concrete and other neo-piagetian forms of reasoning in the novice programmer. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, ACE '11, pages 9–18, Darlinghurst, Australia, Australia, 2011. Australian Computer Society, Inc.
18. L. Ma, J. Ferguson, M. Roper, and M. Wood. Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education*, 21(1):57–80, 2011.
19. C. Moore and S. Mertens The Nature of Computation. Oxford University Press, 2011.
20. M. E. O'Neill. The genuine sieve of Eratosthenes. *The Journal of Functional Programming* 19(1):95–106, 2009.
21. Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas.* Harvester Press, 1980.
22. Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, November 2009.
23. Susan H. Rodger, Maggie Bashford, Lana Dyck, Jenna Hayes, Liz Liang, Deborah Nelson, and Henry Qin. Enhancing K-12 education with Alice programming adventures. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pages 234–238, New York, NY, USA, 2010. ACM.
24. Robert M. Schumacher and Mary P. Czerwinski. *Mental Models and the Acquisition of Expert Knowledge*, pages 61–79. Springer New York, New York, NY, 1992.
25. R. Solovay and V. Strassen. A fast Monte Carlo test for primality. *SIAM Journal of Computing* 6(1):84–85, 1977.
26. Juha Sorva. Notional machines and introductory programming education. *Trans. Comput. Educ.*, 13(2):8:1–8:31, July 2013.
27. Juha Sorva, Jan Lönnberg, and Lauri Malmi. Students' ways of experiencing visual program simulation. *Computer Science Education*, 23(3):207–238, 2013.
28. Juha Sorva and Teemu Sirkiä. Uuhistle: A software tool for visual program simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 49–54, New York, NY, USA, 2010. ACM.
29. Donna Teague and Raymond Lister. Manifestations of preoperational reasoning on similar programming tasks. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*, ACE '14, pages 65–74, Darlinghurst, Australia, Australia, 2014. Australian Computer Society, Inc.
30. D. D. Thornburg Friends of the turtle: On Logo and turtles. *Compute!*, 1983.

## A Python Programs

**Program 1** Starting with a single column of dots, the turtle tries to rearrange the dots to a true rectangle. If this succeeds, the number of dots is not prime.

```
def remove_dot():
    pickup_dot()      # remove a dot from the top
    head_right()      # and drop it to the right
    forward()         # of the current dots
    while not is_pixel_empty():
        forward()
    drop_dot()              # let dot fall down
    # check if the column to the right is full
    if not is_pixel_empty():
        head_left()
        if is_pixel_ahead_empty():
            pickup_dot()    # drop the dot into
            head_right()    # a new column to
            forward()       # the right
            drop_dot()

def find_next_dot():
    head_left()
    # check if the rectangle is complete, i. e.
    # the number of dots is not prime
    if is_pixel_ahead_empty():
        turn_left()
        forward()
        turn_right()
        if not is_pixel_empty():
            if current_y() <= 1:
                print "prime!"
            else:
                print "not␣prime!"
            exit()
    while not is_pixel_ahead_empty():
        forward()

while True:
    remove_dot()
    find_next_dot()
```

**Program 2** An graphical implementation of a primality testing algorithm. In contrast to the sieve of Eratosthenes, multiples of prime factors are not "crossed out." Hence, the algorithm tests each number up to $N$.

```
def test_factor():
    set_pos("origin")
    while is_pixel_empty():
        forward()
    if is_pixel_of_color("red"):
        print "prime!"
        exit()
    drop_dot()
    d = current_x()
    while is_pixel_of_color("blue"):
        forward(d)
    if is_pixel_of_color("red"):
        print "not␣a␣prime!"
        exit()

while True:
    test_factor()
```

**Program 3** This implementation of Minsort is straightforward: the turtle picks up the first colour in the list, and then swaps it whenever it finds a colour that is "smaller." The thus found minimal colour is dropped into the column on the right.

```
while True:
    # Find the first (lowest) dot
    set_pos("origin")
    head_up()
    while is_pixel_empty():
        forward()
    pickup_dot()
    # Search the dot with "smallest" colour
    while not is_pixel_ahead_empty():
        forward()
        if get_pixel_hue() < get_turtle_hue():
            swap_color()
    head_right()
    forward()
    drop_dot()
    # Is the second column completely filled?
    if not is_pixel_empty():
        break
```

## B Quicksort

For Quicksort, the turtle assumes the colour of the first dot in encounters, and then compares the colours of all subsequent dots to its own colour (the pivot colour). Dots with a colour hue that is smaller (or equal) are placed to left, those with larger colour hue are placed to the right (see Program 4, and Figure 3). As long as more than one dot remains in any stack, the procedure is repeated recursively.

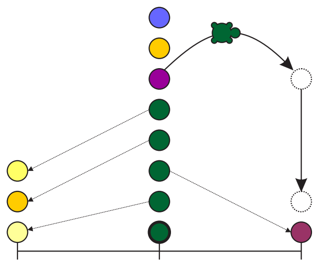

**Fig. 3.** In our implementation of Quicksort, the turtle takes the color of the first dot at the bottom as the pivot. It then moves all dots either to the left or to the right, creating two new columns to be sorted afterwards.

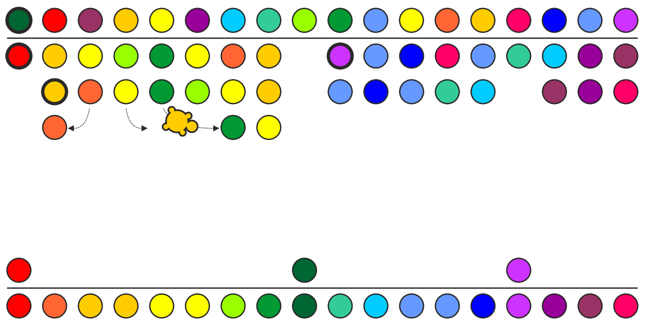

**Fig. 4.** An alternative implementation of Quicksort, where the dots are arranged horizontally. Each pivot colour is dropped to the bottom after the sorting step: due to the gap between the left and the right hand side, its position in the result is already known. At the very bottom, the final solution is shown.

Instead of arranging the dots as vertical stacks as in Figure 3, it is also possible to arrange the coloured dots as horizontal lines as in Figure 4. Note that the line on the right side grows from right to left, leading to the order of the dots being reversed on each step. Since the pivot dot is neither added to the left, nor the right hand side, a gap forms in between the two sides. While an implementation without additional variables is, in principle, feasible, the search for the right spot to place the new dot might take a large amount of both program code, and execution time.

So far, we have never progressed in class enough to actually discuss Python code of Quicksort as shown in Program 4. The use of recursion makes a discussion on entry level difficult (it is possible to do it without recursion, but this does not necessarily lead to better understandable program code). However, we have presented the horizontal version several times as a basis for further discussion, with great results: students often realised, for instance, that the algorithm works best if the pivot colour divides the dots into two lines, or piles, of approximately equal size, and fails to be efficient for dots, "which are sorted in reverse" (it seems that the idea of sorting an already sorted line does not necessarily occur to high school students).

**Program 4** An implementation of Quicksort using turtle graphics. Here we need an abstract variable "delta" to leave room for further columns, which appear due to recursion.

```python
def sort(delta):
    set_pos("bottom")  # only change y-coordinate
    if delta < 1 or is_pixel_empty():
        return
    head_up()
    assume_pixel_color()
    while not is_pixel_ahead_empty():
        forward()
        swap_color()        # save pivot color
        if get_pixel_hue() > get_turtle_hue():
            jump_left(delta)
            drop_dot()
            jump_right(delta)
        else:
            jump_right(delta)
            drop_dot()
            jump_left(delta)
        assume_pixel_color() # assume pivot color
    head_left()           # sort columns on ...
    forward(delta)
    sort(delta // 2)      # ... the left ...
    head_right()
    forward(delta * 2)
    sort(delta // 2)      # ... the right ...
    head_left()
    forward(delta)        # back to middle pos
```