Informatics in Education, 2025, Vol. 24, No. 1, 99–144
© 2025 *Vilnius University*
DOI: 10.15388/infedu.2025.05

How to Teach Problem Solving and Algorithm Design in High Schools by Constructive Induction or How to Reach True Competences in Informatics Education

Juraj HROMKOVIČ*, Regula LACHER
Universitätstrasse 6, 8092 Zürich, Department of Computer Science, ETH Zürich, Switzerland
e-mail: juraj.hromkovic@inf.ethz.ch, regula.lacher@inf.ethz.ch

Received: September 2024

Abstract. The design of algorithms is one of the hardest topics of high school computer science. This is mainly due to the universality of algorithms as solution methods that guarantee the calculation of a correct solution for all potentially infinitely many instances of an algorithmic problem. The goal of this paper is to present a comprehensible and robust algorithms design strategy called "constructive induction" that enables high school students to discover solution methods for a large variety of algorithmic problems. The concept of constructive induction is based on searching for a universal method for solving any instance of an algorithmic problem when solutions of smaller problem instances are available.

In general, our approach strengthens learners in problem solving and their ability to use and develop abstract representations. Here we present a large collection of tasks that can be solved by constructive induction and show how to use this method to teach algorithm design. For some representative algorithmic tasks, we offer a detailed design of lessons in high school classes. We explain how our implementation of teaching in classrooms supports critical thinking, sustainability of acquired knowledge, problem solving, and the ability to abstract, and so contributes to reaching deep competences in algorithmic thinking.

Keywords: critical thinking, computational thinking, algorithm design, constructive induction, searching and sorting, combinatorics and counting, algorithmics, arithmetics.

1. Introduction or Why Teaching Algorithms in High Schools is Not Easy

"Why to teach algorithms?" Because it forces learners to strengthen their ability to abstract and to solve problems. Abstraction and problem solving are crucial dimensions of the human way of thinking and basic instruments for discovering and shaping the world.

* Corresponding author.

Therefore, abilities to abstract and solve problems should be a key issue in any educational system. Especially, the main focus of teaching mathematics and computer science has to be devoted to supporting the learner's ability to abstract and solve problems in their abstract representations.

"Why is teaching algorithms difficult?" To answer this question let us start with relating it to the question "Why is teaching the concept of variables in introductory programming courses the first big threshold?" Good programming courses for novices pay attention to the cognitive load of students and make sure that progress is made in very small steps. The starting point is to view programs as descriptions of activities in the programming language that are understandable for machines (computers, robots, etc.) with the goal to delegate the execution of these activities to technology. In this simplified scenario one program describes exactly one activity. If you take a good choice of your programming language and initial exercises, the first programs describe activities whose execution can be observed visually instruction by instruction (see the concept of the *notional machine* (see du Boulay (1986); Cypher (1993); Fincher *et al.* (2020); Hromkovic *et al.* (2016); Kohn (2017); Kohn and Komm (2018); Lieberman (2001)). With this approach students can develop a program and immediately investigate its properties and functionality, and in this way verify the correctness of their program and get ideas for extending the functionality of programs. This is the reason why even very small kids in primary school can master some basic programming with success and joy. This is also why the repeat-loop with the hidden control variable was introduced to Logo (see Papert (1980)).

If you introduce variables, even as passive input parameters, the game changes heavily. Why is programming with variables much harder? One program is not responsible for only one activity anymore, but depending on the values of its parameters for potentially infinitely many different activities (Arnold *et al.* (2019)). How now to check the functionality of your program? One cannot teach young pupils complete induction proving correctness of programs or any sophisticated verification method of software engineering. For sure you can try to test your program for a few values of its parameters. Consequently, you can fail to believe in the correctness of an incorrect program, but the main problem is whether these few tests are really sufficient to get an intuition of why your program should work properly for all values if its parameters. And the minimal goal of any computer science teacher should be to offer at least some intuition of the program functionality.

An algorithmic problem consists of infinitely many concrete problem instances, and an algorithm is a solution method that works correctly and successfully (and if possible efficiently) for any one of these infinitely many instances. Hence you are asked to develop a strategy that behaves well in all infinitely many possible situations. This is a very nontrivial task, especially if you take into account that high school students have almost no experience with the universal quantifier. We cannot assume any experience with using mathematical induction for proving infinitely many parameterized claims, and so we cannot strive to prove the formal correctness of the algorithms developed. Of course we could omit verification proofs. But if our lessons have an educational value, then they have to offer some reasonable intuition, why the algorithm designed works properly. To reach this goal, teaching must be designed in such a way that students have enough freedom to design algorithms to a high extent on their own. First, students have to be guided to solve concrete instances of an algorithmic problem and using the acquired experience to design more general solving strategies. Then they have to be trained to find counter examples for proposed strategy to strengthen their experience. The highest art of successful teaching is to design a guidance that enables students to discover working solution strategies by their own. This is a very nontrivial task we will try to approach in the subsequent sections.

Let us consider another dimension in answering our question "Why is teaching algorithms in high school not easy?" A teacher can aim to explain some famous algorithm and the goal could be to be able to execute the algorithm by hand. We question this goal. First of all, the value of teaching acting according to a given pattern is very low (for a more detailed discussion see Hromkovic and Lacher (2023) and Dagiene *et al.* (2021)), because it contributes very little to the development of our thinking (creativity, improvisation, fantasy, problem solving). Every procedure we can describe can be automized and so executed more reliably and quicker by technology than by humans. Secondly, most efficient algorithms are so specifically adjusted to the problems they solve, that a small change in the specification of the algorithmic problem can cause (and usually does) the algorithm to fail. We say that such algorithms are not robust and so their teaching does not guarantee any essential progress in developing and strengthening the ability of students to solve problems. We argue that we have to teach robust strategies for problem solving that can be successfully applied to a big variety of problems.

The remaining question now is which of the algorithm design methods can be taught successfully in high schools. In the following we present and recommend one of these methods, called *constructive induction*, and show how a teacher can use it in schools to train problem solving and algorithm design. Constructive induction (as probably the oldest general problem solving strategy) can be viewed as a simple version of recursion, and so it can be used as a well understandable introduction to more general algorithm design method as the famous recursive ”divide and conquer” and the bottom-up strategy ”dynamic programming”.

This paper is organized as follows. In Chapter 2 we present the concept of constructive induction. In Chapter 3 we show how to use constructive induction to solve a variety of different mathematical problems. In Chapter 4 we apply constructive induction to design algorithms for various algorithmic problems. Finally, in Chapter 5 we present detailed designs of lessons in high schools for some representative mathematical and algorithmic tasks. The goal of presenting high school implementations is showing how to strengthen the ability of students to solve problems, and how to support students to discover algorithms to high extent by their own instead of presenting some famous algorithms to students as finalized scientific products.

2. Constructive Induction

Every teacher knows *mathematical induction* (also called *complete induction*) used to prove infinitely (countably) many parameterized claims. The word induction has its origin in Latin ("inductio" – "to lead into" in English). Probably the oldest induction proof was done by al-Karaji in the tenth century for proving the binomial theorem (Pascal's triangle) about the form of coefficients (Bussey (1917); Tanton (2021)). Unfortunately, this manuscript was lost and we only have the reference in the book "The Brilliant in Algebra" by Al Samawal al-Maghribi (around 1150). In European culture induction as a proof method was used for the first time by Francesco Maurolico in 1575 (Vacca (1909)) for proving that the sum of the first n odd integers is n^2. It took several years until the method was used again (Blaise Pascal, 1654; Jacob I Bernoulli, 1686). In 1888, Richard Dedekind started to call this proof method "complete induction." Giuseppe Peano presented induction as a part of his axiomatic system in 1889. Since then, this method belongs to the fundamental instruments of mathematics. Because reading, correcting, and writing proofs is not a mandatory subject of mathematics in high school in most countries, one can question whether it is reasonable to teach mathematical induction in computer science lessons. But we want to deal with constructive induction here, which is easier available to students than mathematical induction.

Constructive induction for building sequences of objects or for solving problems is much older than complete induction as a proof method and was used already in ancient time. The simplest fundamental example is the construction of natural numbers. For each number n we can construct the next larger number $n + 1$. In this case we use the constructive induction to construct an infinite sequence of objects. Another antique example is the claim that there are infinitely many primes. Take the n smallest primes $p_1, p_2, \cdots, p_n$.

Now we develop a strategy for finding the next prime p_{n+1}. Take the number

$$m = p_1 * p_2 * \cdots * p_n + 1\,,$$

which is not divisible by any of the first n primes $p_1, \ldots, p_n$. So there must be a prime in the interval between $p_n + 1$ and m. Now one has to check these finitely many candidates from the smallest one to the largest one to find the next prime. What we see in this example? For any n, we have a strategy for how to find the $(n + 1)$-th prime if you have the first n primes, and you proved that this strategy works (i.e., that there are infinitely many primes).

In the examples above we saw how we can generate the next object of an infinite sequence of objects step by step if we know the previous ones. For sure this method works only if we have or can construct the starting object of this sequence.

We can extend the constructive induction presented above for solving problems and designing algorithms. The idea is as follows. First, we have to parameterize the problem. This means that we have to partition the set of its infinitely many instances into infinitely many classes numbered by natural numbers. The classes can be finite or infinite. For instance, a parameter of a graph can be the number of its vertices, the number of its edges, the size of its adjacency matrix, the maximum degree of its vertices, or its diameter. A parameter of an integer can be the integer itself or the length of its representation in some number system. A parameter for sorting or searching can be the number of elements or the length of the whole input representation. After parameterizing a problem, we say that a problem instance is of size k if it belongs to the k-th class.

The idea of using constructive induction to solve problems is now very similar to complete induction. First, one solves the problem for the smallest parameter, i.e., the smallest problem instances. We speak about the base of the induction. Then, for any positive integer n, one executes a general strategy, how to use solutions for instances smaller than n (with parameters smaller than n), in most cases even only with parameter exactly $n-1$, to construct a solution for any instance of size n. The key issue here is searching for this general strategy.

In Chapters 3 and 4 we show that there are plenty of problems for which such strategies are quite natural and can be discovered successfully. This is important, because after experiencing some examples the students can start searching for algorithmic solutions to similar problems on their own. In Chapter 3 we show how to use induction to solve problems, and in Chapter 4 we apply constructive induction for designing algorithms. Chapter 5 is devoted to the presentation of a detailed implementation of lessons about solving algorithmic problems by constrictive induction. We design here how teacher can proceed in such a way that students discover as much as doable by their own with minimal support (guidance instead of explanations). In this way we show how to teach high school students to design efficient algorithms.

3. Solving Problems by Constructive Induction

Before starting to solve algorithmic problems by constructive induction let us summarize what the simplest version of this algorithm design method is about. Let us have a parametrized problem, i.e., we can assign to each problem instance its size. First, one has to solve the problem instances of the smallest size (usually 1). Then, one has to find a general strategy how to solve any instance of size n if the solutions of instances of size $n-1$ are available. The most transparent problems for novices in algorithm design are problems having exactly one instance of size n for any natural number n. We will start with exactly such problems.

We present a sequence of problems that can be transparently solved by constructive induction. A detailed description of the lessons for high school students for some representative tasks are presented in Chapter 5 (see Gallenbacher *et al.* (2023) for a larger set of tasks and also for the introduction to mathematical induction).

The number of decimal numbers with at most n digits. The question is how many decimal numbers of length at most n exist. The length of a number is the number of digits in its representation, which is the parameter of this problem. This is a very simple introductory task.

The base of the induction is easy. We have 10 digits $0, 1, 2, \ldots, 9$, and so we have 10 integers of length 1. To discover the general step, we always encourage students to go from size 1 to size 2, from size 2 to size 3, etc. until they recognize a pattern in these concrete steps. Going from size 1 to size 2 here means to build a table of size 10×10. In the rows there are 10 digits 0, 1, 2, …, 9, and the columns contain the 10 decimal numbers of length 1. Combining the labels of the rows with the labels of the columns we get 100 sequences of 2 digits. Any row estimates the first digit of the two digit representation and the label of each column estimates the second digit. Removing the leading zeros, we have all 100 decimal numbers of length at most 2. The generalization of this strategy is transparent. For the representation length at most n we take a $10 \times 10^{n-1}$ table. In the rows we have all 10 digits, and, in the columns, we have all sequences of $n-1$ digits.

Note that this problem can be solved easier by asking which is the largest integer of length n. The answer is the number consisting of digits 9 only. But the advantage of using this task is to get a very simple introduction to constructive induction.

Another approach based on constructive induction is using trees in which each tree level is used to choose the digit on the corresponding position in the number representation (for details see Gallenbacher *et al.* (2023)).

***Lines in two-dimensional Euclidean space*.** Suppose we have n different lines in two-dimensional Euclidean space, and we are asked to place them in such a way that the number of crossing points is as large as possible (see Fig. 1 for an example). The task is to estimate the maximal number of crossing points for n lines in two-dimensional Euclidean space for every n. Note that the minimal number of crossing points is 0 if the lines are parallel to each other. Obviously, the parameter n is the number of lines.

For $n = 1$ there is no crossing point. For $n = 2$ there is exactly one crossing point if the lines are not parallel. The strategy is to place the n-th line in such a way that it crosses all $n-1$ already placed lines in new $n-1$ crossing points. This allows us to count the maximal number of available crossing points of n lines. It is $1 + 2 + 3 + \cdots + (n-1)$. Exactly this sum is the product of using constructive induction and can be also derived by using the recurrence $L(n) = L(n-1) + (n-1)$.

Using the "small Gauss" one can get an explicit formula. But we can already use the sum above to develop an algorithm (program) counting the maximal number of crossing points of n lines for any given n. The program simply computes in a for-loop the sum $1 + 2 + \ldots + (n-1)$ for a given integer n.

***The introductory example of Poya*.** The previous problem is only a preparation of the following problem used by Poya when introducing the power of induction. There are n lines in two-dimensional Euclidean place. What is the maximal number of areas in which the space can be partitioned in this way?

One can start with 1 line getting 2 areas. Taking 2 lines one can get 4 areas. Taking 3 lines we will get 7 areas (see Fig. 2) if no two lines are parallel.

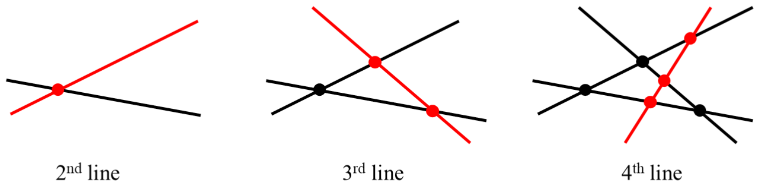


Fig. 1. Crossing points when lines are added in the two-dimensional Euclidean space.

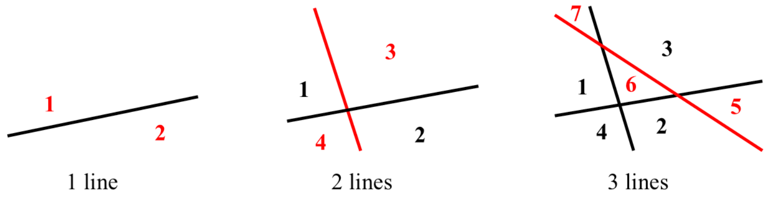


Fig. 2. Number of areas created by 1, 2, or 3 lines.

Clearly, as pointed by Poya in his lesson, the solution cannot be 2^n, and the task starts to be challenging. For finding the solution, solving the previous task can be helpful. With the n-th line added to the previous $n-1$ lines we can get $n-1$ new crossing points.

Each of the $n-2$ segments of the n-th line between two new crossing points partitions an area into two subareas. The segment of the n-th line "before" the first crossing point and the segment after the "last" crossing point also partitions areas into two subareas. Hence, adding the n-th line to the already placed $n-1$ lines increases the number of areas by n. Therefore, the maximal number of areas obtained by placing n lines is

$$2+2+3+4+\cdots+n\ .$$

To see this, denote the number of areas for n lines by $T(n)$ and consider $T(1)=2$ and the derived recursion $T(n)=T(n-1)+n$.

If you are searching for a challenge, consider the number of subareas of three-dimensional Euclidean space obtained by placing two-dimensional planes. As an exercise for students, you can take circuits instead of lines (or other suitable geometric objects) and ask to solve the problem by constructive induction. For more details see Chapter 5.

Number of triangles in a pyramid. One can build a pyramid from equilateral triangles. In Fig. 3 we see the four smallest pyramids with the heights 1, 2, 3, and 4. The question is how many triangles $PN(n)$ are in the pyramid of height n for any positive integer n. After previous experience, the students can compute $PN(1)=1$, $PN(2)=4$,

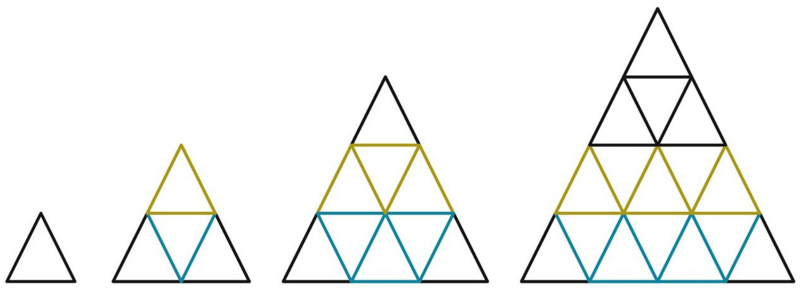


Fig. 3. Pyramid built from equilateral triangles.

and finally the recurrence $PN(n) = PN(n-1) + 2n - 1$. To make the task easier the teacher may first ask for the number $A(n)$ of triangles in the lowest level of the pyramid of height n. Here one can easily establish $A(1) = 1$ and $A(n) = A(n-1) + 2$, and so $A(n) = 2n - 1$. Then the formula $PN(n) = PN(n-1) + A(n)$ follows. If you decide to search for an explicit formula, you can derive $PN(n) = n^2$. We call attention to this because in this way you get the classical example of the sum of the first n odd numbers presented by Francesco Maurolico (Vacca (1909)).

An interesting point here is that one can calculate the number of triangles in a pyramid by dividing the area of the pyramid by the area of the triangle (for instance taking 1 as the size of the equilateral triangle as the basic building stone). One can generate similar tasks by building pyramids from squares or hexagons.

Coloring regions of a map. The famous *four color theorem* states that 4 colors are always sufficient to color a two-dimensional map consisting of regions in such a way that no two neighboring regions have the same color. Two regions are considered to be neighbors if they have a common continuous border consisting of infinitely many points. One could ask whether less colors are sufficient if the partition of the map into regions is done in some restricted way, for instance by placing some regular geometric objects into the plane. Using our experience with the task from Poya, we can pose the following question: A map is partitioned into regions by n lines. How many colors are sufficient to color the map?

If you start with 1, 2, or 3 lines (see Fig. 4), after some attempts the students can discover that for small parameter values two colors are sufficient. So we have a hypothesis, but the question is how to prove it: The natural way is by constructive induction. First, observe that exchanging the two colors in a valid coloring leads again to a valid coloring. Secondly, add a new line to a map colored by two colors (see Fig. 5 left). Again, we can view the new line as a sequence of segments between the crossing points with other lines. All these segments have the same colors on both sides now and all other (previous) boundaries between two regions have different colors. Keep the coloring on one side of the new line and flip the colors on the other side. In this way we get a valid coloring of the new map by two colors (see Fig. 5 right).

The strength of this task is that you can generate plenty of similar ones to train the class. You can take circles to partition the plane into regions, or triangles (rectangles,

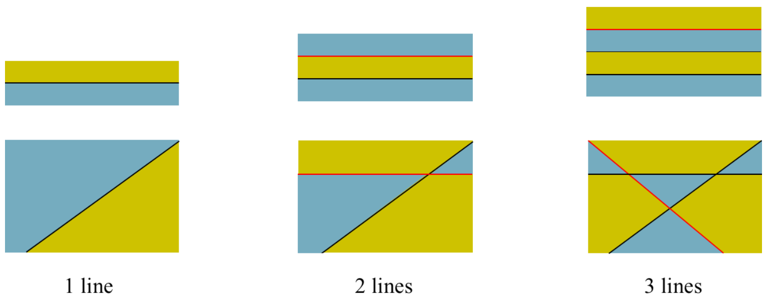


Fig. 4. Two colors are sufficient to color all areas created by 1, 2, or 3 lines.

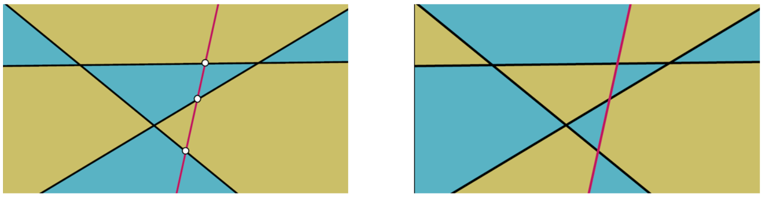


Fig. 5. Adding a 4-th line and changing the color of all areas on one side of the new line.

etc.) with and without the request that these geometrical objects intersect in a finite number of points only. If the request is satisfied, two colors are always sufficient. The strategy is to put the next object into the plane, and then flip the colors either inside of the new object or outside. If the constraint is not satisfied, one can ask students to construct examples that need at least 4 colors to be properly colored. Then the students can be asked to find the reason why the previous strategy with flipping colors inside of the new object does not work. More details to this coloring problem we give in Chapter 5.

4. Designing Algorithms by Constructive Induction

Here we can start with the last task of the previous Chapter, which is a good example for moving from problem solving to algorithm design.

Coloring regions of a map. The approach to solve the problem of dividing a plane by lines (circles, etc.) into regions offers an algorithm for two-coloring. Place the first line and color the plane with 2 colors. Then add one line after another and always exchange the colors on one side of the line.

Discovering the multiplication algorithm. Only few people are aware of the fact that the multiplication algorithm currently used in schools was designed by constructive induction. If you want to compute the product $a \cdot b$ you can parameterize by the length of the representation of b. The length of a does not matter. Suppose b consists of n +1 digits and assume we can multiply by b with n digits. Let

$$b = b_n b_{n-1} \dots b_1 b_0 \,.$$

We can then write

$$a \cdot b = a \cdot (b_n b_{n-1} \dots b_1 b_0) = a \cdot b_n b_{n-1} \dots b_1 \cdot 10 + a \cdot b_0 \,.$$

So, we see that one multiplication by a number consisting of n digits, one shift by one position (multiplication by 10), one multiplication by one digit, and one addition are sufficient to compute the multiplication by a number consisting of $n + 1$ digits. This is exactly the base of our school multiplication algorithm.

To train this concept one can multiply in other number systems or take other arithmetic operations (further examples are in Gallenbacher *et al.* (2023)).

Horner's method. The development of Horner's schema for evaluating a polynomial is a show case of applying constructive induction. Better than by any formula, the algorithm is explained by Fig. 28. We discuss this task in detail in Chapter 5.

Who is the agent. We have n people and want to discover who of them is an agent. We know that there is an agent among those n people. How to recognize her or him? The agent is the person who knows everybody (all other $n - 1$ people), but nobody knows her or him. We are allowed to pose the following questions: "Person A, do you know person B?" and will always get the correct answer. The task is to find the agent with as few questions as possible.

First we observe that there can be at most one agent in the group of people. If there would be two, then each of them would know the other, and so none of them could be an agent. The induction base for the group size of one person is simple, no question is needed. For the induction step, the point is to recognize that one question is sufficient to reduce the number of candidates by 1. If we ask whether A knows B, then the answer "yes" excludes B as an agent candidate. If the answer is "no," then A cannot be the agent. In this way we see that $n - 1$ questions are sufficient to find the agent if one knows in advance that there is an agent in the given group of people.

There is a wonderful extension of this task by allowing some restricted cheating (one or more wrong answers). This extension leads to the development of self-verifying codes for the corresponding numbers of errors.

Sorting and searching. Also in this fundamental area of algorithm design the concept of constructive induction is very fruitful. One can start with searching for the minimum (or for the maximum) of n elements and design an algorithm with $n - 1$ comparisons by constructive induction.

The next step could be to do *binary search*. One can take the length of the number representation of n (approximately the discrete logarithm of n) as the parameter for the size of sorted sequences of n elements. The induction step then shows that, if one can find an element in a sorted sequence of m elements, then one more comparison is sufficient to find an element in a sorted sequence of $2m$ elements.

For sorting algorithms one can consider *insertion sort*. If one has a sorted sequence of n elements and takes a new element, then one has to find its position in the sorted sequence. One can get different algorithms depending on the induction strategy used. If one uses *bubble sort* for placing the new element, the resulting algorithm has a quadratic number of comparisons. If one uses *binary search*, the complexity is in $O(n \log n)$.

Another sorting algorithm can search for the maximum (as bubble sort does) and then sort the remaining $n - 1$ elements.

There are plenty of possibilities to create further exercises here (see Chapter 5). For instance, one can search for the k largest elements.

Winning strategies for games. Constructive induction is a genius strategy for searching for winning strategies for finite games. You label the configuration with the following very simple labeling rules: Start with your winning configurations and losing configurations defined by the game, and in each constructive step you label your further winning or losing configurations by going one step back in the game.

You continue to repeat the induction steps until all configurations of the game are labeled. You label a configuration as your winning configuration if you can move from this configuration in one step to a configuration that is already labeled as your winning configuration (if the turn is yours). If its your opponent's turn, you label a configuration as your winning configuration if she or he can only move to one of your winning configurations. When it is your turn, you label a configuration as your losing configuration (winning configuration of your contrary) if all your possible moves end in an already marked winning configuration of your opponent. Also, you label a configuration as your losing configuration, if your opponent on turn can reach in one step a winning configuration for her or him.

There are many simple games that can be completely analyzed by this strategy (for a variety of examples see Gallenbacher *et al.* (2023)).

Dijkstra's shortest path algorithm. This is an advanced example. Especially since the number of different paths between two vertices in a network can be exponential in the size of the network, and we want to avoid looking at all possible paths in order to find the shortest one.

The key issue when developing Dijkstra's algorithm by constructive induction is the choice of the parameter. First, one defines the problem of finding the shortest paths in a network from the source to the k closest vertices. Then k is the parameter, not the size of the network. The base for $k = 1$ is easy. One takes that neighbor of the source that is connected to it by the cheapest edge. The induction step takes the tree with the k shortest paths to the k closest vertices and argues that the shortest path to the $(k + 1)$-th closest vertex must go via the edges of the tree of the k shortest paths (for details see the implementation for high school students in Gallenbacher *et al.* (2023)). Then one can efficiently estimate the next closest vertex by considering only edges of the tree for the prefix of the path, and for the last edge on the shortest path the edges leading from the tree to the vertices not belonging to the k closest ones.

If one wants to start teaching or discovering Dijkstra's algorithm with an easier task, then one can search for shortest paths from a source to all other vertices in a network whose edges all have the same value and so develop the *breadth-first-search* algorithm.

5. Teaching Constructive Induction in Classrooms

The goal of this section is to show how to successfully implement the ideas presented in the two previous sections. We are far away of aiming only to explain famous algorithms in order to be able to execute them on arbitrary problem instances. We want to involve students as much as doable into the process of discovering solution methods, and so to reach reasonable competences in designing algorithms for given problems. This is related to our definition of a competence which is frequently not well understood in didactic literature and textbooks.

"*A competence is not an ability (a skill) to act (to handle) by following a given pattern, does not matter how complex this pattern*

> *(algorithm) is. A competence is the ability to intelligently apply the acquired knowledge and experience in order to master new situations, problems and challenges.*"

Particularly in algorithmics education it means that we have to present such sequences of examples and challenges that pupils will start to develop own algorithmic solutions for new, given tasks. Our way of teaching competences uses the so called "historical (or genetic) Socrates method" (see Hromkovic and Lacher (2024)).

> "*Do not teach the products of science and technology (facts, theorems, models, methods, research instruments, and tools) and how to use them, but teach students the processes of their discovery and development.*"

This concept is called historical (genetic) Socrates method because it focuses on teaching to study the genesis of the products of science and technologies. Teachers have to understand the original motivations, see how to learn from failures when trying to make progress, and how to get the right ideas due to the acquired experience. Students have to learn repeating attempts to master the given challenge and evaluate the products of their own work (which of the goals have or have not been achieved and to which extent). The finest art of applying historical Socrates method consists of partitioning the discovery process into a sequence of such small and simple challenges (steps) that students are able to master these particular steps to high extent by their own. Note that discovering algorithms in such a way, instead of learning to execute them only, offers a completely different level of sustainability and contributes to reaching true competences in computational thinking.

In what follows we present our attempts to apply historical Socrates method for designing lessons for training algorithm design for some representative high school classroom tasks. We use the **QT** (Questions and Tasks) to activate the students and **ED** (Explanations and Definitions) to introduce new objects, concepts or methods.

5.1. *Combinatorial and Geometric Tasks by Pattern of Poya*

The simplest task suitable as an introduction to this set of tasks is to study the number of crossing points of n lines in the two-dimensional Euclidian space. This task combines geometry with combinatorics and the solutions can be nicely visualized. We can start with the following tasks denoted as **QT** (questions and tasks) in what follows.

QT 5.1.1 What is the minimal number of crossing points of $n = 1, 2, 3, \ldots$ different lines in two dimensional Euclid space (plane)?

Solutions and expectations The students have to recognize that the answer is 0 for all n, because one can place all the lines in such a way that each one is parallel to each other. One could ask students to describe the set of lines explicitly for a given concrete n. For instance for $n = 5$ one can take $y = 0$, $y = 1$, $y = 2$, $y = 3$, and $y = 4$ (or

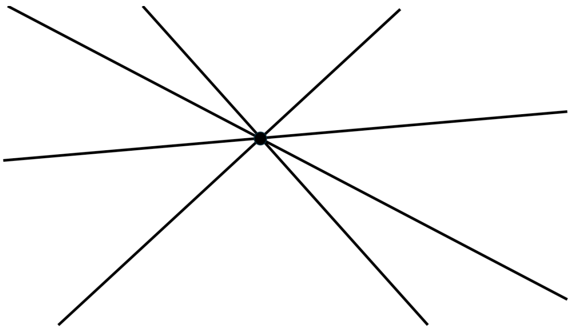


Fig. 6. Minimal number of crossing points for lines that are not parallel

alternatively $y = x$, $y = x + 1$, $y = x + 2$, $y = x + 3$, and $y = x + 4$). Another gain of this task is to recognize that for a given number n of lines the number of crossing points can be different depending on the placement of the lines.

QT 5.1.2 We have n lines in a plane, no two are parallel to each other. What is the minimal number of crossing points?

Solutions and expectations The students have to recognize that the number is 1 for all n, because all n lines can can cross in one point (see Fig. 6). Students can also provide explicit description as $y = x + 1$, $y = 2x + 1$, $y = 3x + 1$, ... Solving this task should help students to discover that to get the maximal number of crossing points no two lines are allowed to be parallel, and no three lines are allowed to cross each other in the same point.

QT 5.1.3 a) What is the maximal number of crossing points for $n = 1$, 2, and 3 lines?

b) You know the maximal number of crossing points for 4 lines. How to use it to estimate the maximal number of crossing points for 5 lines?

c) Describe a general strategy how to estimate the maximal number of crossing points of $n + 1$ lines if the number of crossing points of n lines is known.

Solutions and expectations a) Students develop solutions for $n = 1$, 2, and 3 as depicted in Fig. 1. Teacher should introduce the notation $CL(n)$ for the maximal number of n crossing lines and fix in this notation: $CL(1) = 0$, $CL(2) = 1$, $CL(3) = 3$.

b) Students can do it in an abstract way, but preferably by drawing. If students have a solution for $CL(4)$, they know that to achieve the maximal number of crossing points they have to draw a new line (red in Fig. 7) that crosses all the already placed 4 lines in new crossing points. Students can discover that $CL(5) = CL(4) + 4$ (i.e. old crossing points plus 4 new ones). If students describe this solution in words, teacher has to describe it as a recurrence as above.

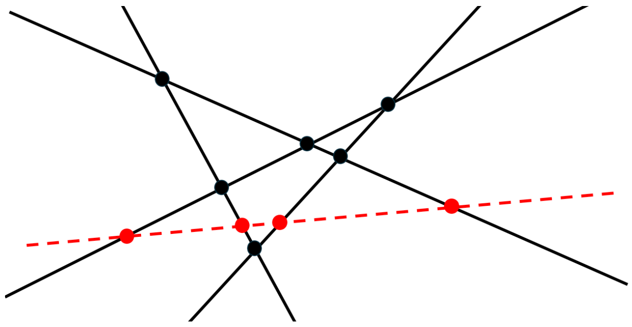


Fig. 7. Minimal number of crossing points.

c) Using the experience from solving b), we expect that students can formulate a general strategy for estimating $CL(n+1)$ from $CL(n)$. The $(n+1)$-th line is placed in such a way that it crosses each of the already placed n lines in a new crossing point, and so the number of all crossing points is now $CL(n+1) = CL(n) + n$. Again, teacher has to bring the corresponding recurrence equation above and discuss with the class what this recurrence expresses.

We use the notion **ED** for explanations and definitions that teacher offer to the class in order to introduce new objects, concepts or methods. After the experience with QT 5.1.1, 5.1.2 and 5.1.3 teacher can introduce recurrence equations and constructive induction as follows.

ED 5.1.1 Functions are models describing the relationship between the values of two or more attributes (variables). For instance $y = f(x) = 2x^2 + 7$ determines the relationships between y and x. If the value of x is known, one can use the equation $y = 2x^2 + 7$ to estimate unambiguously the value of y. If y is known, one can estimate all possible values of x satisfying the equation $y = 2x^2 + 7$ (for instance, if $y = 15$, one can look for which x the equation $15 = 2x^2 + 7$ is satisfied). Such description of a function ($f(x) =$ *formula with unknown* x) is called an ***explicit representation (description)*** of function f.

In QT 5.1.3 we discovered a new representation of functions from natural numbers to natural numbers. We have $CL(1) = 0$ and $CL(n+1) = CL(n) + n$. This description of the function $CL(n)$ consists of two components. The first component $CL(1) = 0$ estimates the value of the function for the smallest argument, and is called the ***base*** of the description. The second component $CL(n+1) = CL(n) + n$ explains how to compute $CL(n+1)$ if the value of $CL(n)$ is known, and is called a ***recurrence equation***. We say that we have a ***recursive description*** of function $CL(n)$.

A recursive description of a function is complete in the sense, that this description suffices to compute the value of the function for any natural number n. In our case it works as follows:

$CL(1) = 0$

$CL(2) = CL(1) + 1 = 0 + 1 = 1$
$CL(3) = CL(2) + 2 = 1 + 2 = 3$
$CL(4) = CL(3) + 3 = 3 + 3 = 6, \ldots$

For our function $CL(n)$ one can derive the explicit representation (description) from its recursive description $CL(n) = CL(n-1) + (n-1)$.

$$\begin{aligned} CL(n) &= CL(n-1) + (n-1) \\ &= CL(n-2) + (n-2) + (n-1) \\ &= CL(n-3) + (n-3) + (n-2) + (n-1) \\ &= \ldots = CL(n-i) + (n-i) + (n-i-1) + \ldots + (n-1) \\ &= \ldots = CL(2) + 2 + 3 + \ldots + (n-2) + (n-1) \\ &= 1 + 2 + 3 + \ldots + (n-2) + (n-1)\ . \end{aligned}$$

This description as a sum of numbers from 1 to $n-1$ can be simplified as follows. One writes this sum twice in different order and adds the values on the same position in the sums.

$$\begin{array}{ccccccccccccc} 1 & + & 2 & + & 3 & + & 4 & \ldots & + & n-2 & + & n-1 \\ n-1 & + & n-2 & + & n-3 & + & n-4 & \ldots & + & 2 & + & 1 \\ \hline n & + & n & + & n & + & n & \ldots & + & n & + & n \end{array}$$

Hence, the result is $(n-1) * n$. Because we doubled the sum, we obtain the explicit representation $CL(n) = (n-1) * n/2$ of function CL.

QT 5.1.4 The following functions are given by their recursive descriptions. Compute their values for their 5 smallest argument, and find their explicit representations (descriptions).

a) $F(n) = F(n-1) + 2(n-1)$ and $F(1) = 1$
b) $F(n) = F(n-1) + 2$ and $F(1) = 5$
c) $F(n) = 2 * F(n-1)$ and $F(0) = 1$
d) $F(n) = F(n-1) + 2(n-1) + 1$ and $F(1) = 1$

Solutions and expectations We expect that students convince themselves that a recursively described function can be computed iteratively for any value of its argument and can execute the computation. Optionally, teacher can ask to implement the computation by a program. Because we do not want to start teaching general recursion at this point, we have chosen simple recursive function for which the value of $f(n)$ depends only on the value of $f(n-1)$ (the previous value of the function for $n-1$). The functions are chosen in such a way that they can be used by solving algorithmic problems presented in the following parts of this Chapter.

a) $F(1) = 1$
$F(2) = F(1) + 2 * 1 = 1 + 2 = 3$
$F(3) = F(2) + 2 * 2 = 3 + 4 = 7$
$F(4) = F(3) + 2 * 3 = 7 + 6 = 13$
$F(5) = F(4) + 2 * 4 = 13 + 8 = 21$

Similarly as in ED 5.1.1 one can observe that
$F(n) = F(n-1) + 2*(n-1)$
$= F(n-2) + 2*(n-2) + 2*(n-1)$
$= F(2) + 2*2 + 2*3 + 2*4 + \ldots + 2*(n-2) + 2*(n-1)$
$= F(1) + 2*1 + 2*2 + 2*3 + \ldots + 2*(n-2) + 2*(n-1)$
$= 1 + 2(1 + 2 + 3 + \ldots + (n-2) + (n-1))$
$= 1 + 2((n-1)*n/2)$
$= 1 + (n-1)*n$

Now, teacher may ask to use the explicit representation (description) of $F(n)$ to compute the values $F(1)$, $F(2)$, $F(3)$, $F(4)$, and $F(5)$, and compare them with the values computed by the recursion.

In general, if deriving the explicit representation (description) of F is too hard for the class, it can be omitted.

b) $F(1) = 5$
$F(2) = F(1) + 2 = 5 + 2 = 7$
$F(3) = F(2) + 2 = 7 + 2 = 9$
$F(4) = F(3) + 2 = 9 + 2 = 11$
$F(5) = F(4) + 2 = 11 + 2 = 13$

Students have to recognize that the function computes odd numbers starting from 5.
$F(n) = F(n-1) + 2$
$= F(n-2) + 2 + 2 = F(n-3) + 2 + 2 + 2$
$= F(n-m) + m*2$
$= F(1) + (n-1)*2$
$= 5 + 2*(n-1)$

c) $F(0) = 1$
$F(1) = 2*F(0) = 2*1 = 2$
$F(2) = 2*F(1) = 2*2 = 4$
$F(3) = 2*F(2) = 2*4 = 8$
$F(4) = 2*F(3) = 2*8 = 16$
$F(5) = 2*F(4) = 2*16 = 32$

One can observe, that $F(n) = 2^n$. Because we do not aim to teach complete induction as a proof method, we do not need to strive making a formal derivation of the explicit representation of the function F and are satisfied with the achieved intuition.

d) $F(1) = 1$
$F(2) = F(1) + 2*1{+}1 = 1{+}2{+}1 = 4$
$F(3) = F(2) + 2*2{+}1 = 4{+}4{+}1 = 9$
$F(4) = F(3) + 2*3{+}1 = 9{+}6{+}1 = 16$
$F(5) = F(4) + 2*4{+}1 = 16{+}8{+}1 = 25$

Now, the students may guess $F(n) = n^2$. A helpful intuition can be obtained from $(n+1)^2 = n^2 + 2n + 1$, i.e. the next quadratic value can be obtained from the previous one by adding $2n + 1$ as the $(n+1)$-st odd number. Combining approaches from a) and b), one can even derive $F(n) = n^2$, but this is for strong students only.

ED 5.1.2 ***Constructive induction*** is a powerful method for solving problems and designing algorithms. One can use constructive induction if one is able to ***parametrize*** the set of all instances of the problem considered. Parametrizing a problem means to split the set of all its input instances into disjoint classes C_1, C_2, C_3, $\ldots$ or in other words to assign to each problem instance a natural number called the size of the instances. If a problem is parametrized, one can try to solve the problem by the following inductive strategy.

1. **Induction base** Solve the problem for the instances of the smallest size.
2. **Induction step** Discover how to solve any instance of size $n + 1$ if you have solutions for instances of size n. In general one can use the solutions of all instances of size smaller than $n + 1$ when solving an instance of size $n + 1$.

QT 5.1.5 a) What is the minimal number of crossing points of n circles in a plane?

b) What is the maximal number of crossing points of $n = 1, 2, 3$ circles in a plane?

c) Find a recursive description of function $CC(n)$ counting the maximal number of crossing points of n circles in a plane.

d) Find an explicit description of $CC(n)$.

Solutions and expectations Students have to be able to follow the solution schema of QT 5.1.3 and FD 5.1.1 and 5.1.2 to solve this task by their own.

a) The answer is 0. One can place arbitrary many non overlapping circles in a plane by different strategies. For instance placing the next circle inside of the smallest circle or placing in some distance to the right from the previous circle.

b) Students have to discover solutions as in Fig. 8. Students have to recognize that two circles can cross in at most 2 points; does not matter whether they have different sizes or not. To draw several circles in such a way that each circle crosses each other in exactly 2 points it is sufficient to have a point laying inside of all circles. Students fix that $CC(1) = 0$, $CC(2) = 2$, and $CC(3) = 6$.

c) Students can develop a general strategy as described in Fig. 9. A new circle is added in such a way that is crosses each already placed circle in exactly two new points. In this way one obtains the recurrence:
$CC(1) = 0$ and $CC(n + 1) = CC(n) + 2n$

d) Students can apply the solving strategy of QT 5.1.4 b) to get
$CC(n) = n * (n - 1)$

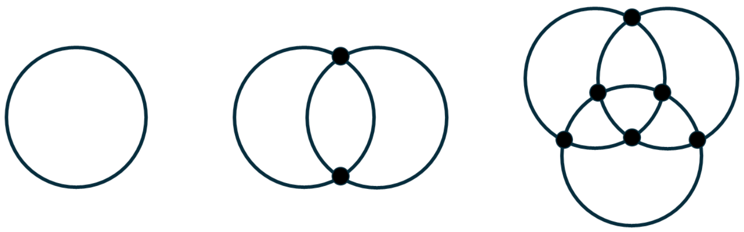


Fig. 8. Crossing points of circles.

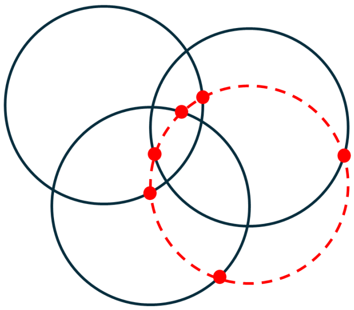


Fig. 9. Crossing points when a fourth circle is added.

QT 5.1.6 What is the maximal number of crossing points of n ellipses in a plane?

Solutions and expectations In this exercise we do not guide students by splitting the task into a sequence of more simple steps as in QT 5.1.4 and 5.1.5. We expect that the students manage the process of problem solving by their own.

In comparison to QT 5.1.5 students have to discover first, that two ellipses can cross in at most 4 points (see Fig. 10). In this way they get the base of the induction

$EC(1) = 0$ and $EC(2) = 4$

The next discovery (analog to circles) is that one can draw the $(n + 1)$-th ellipse in such a way that it crosses all other ellipses (previous ones) in 4 new crossing points. Consequently, students derive $EC(n + 1) = EC(n) + 4n$. Optimally, students can find the explicit description $EC(n) = 2 * n(n - 1)$.

Additional exercises A benefit of our combinatorial, geometric tasks is that a variety of similar tasks exist and are suitable to train applying constructive induction. We list some of the possibilities here.

1. What is the maximal number of crossing points of n broken lines? Two broken lines (see Fig. 11) can cross in a most 4 points and the solution is the same as in the case of the ellipses.

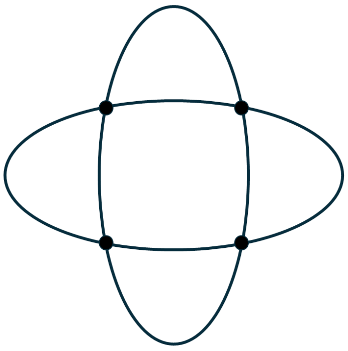


Fig. 10. Two ellipses with four crossing points.

2. What is the minimal number of crossing points of n triangles, if any two triangles have at most finitely many common points (there is no common line belonging to both triangles). Following Fig. 12, two triangles can have 6 crossing points, and so $TC(1) = 0$ and $TC(2) = 6$. The corresponding recurrence is $TC(n+1) = TC(n) + 6 * n$.

   One can take rectangles instead of triangles and will obtain the already known function $RC(1) = 0$ and $RC(n+1) = RC(n) + 4 * n$.
3. A pyramid is built from quadratic stones as in Fig. 13. The height of the pyramid is the number of its levels. From Fig. 13 we see $QuadC(1) = 1$, $QuadC(2) = 3$, $QuadC(3) = 6$, and $QuadC(4) = 10$. If we increase the height by 1, we have to add the base level. The number of stones in the base of the pyramid of height $n+1$ is $n+1$, and so $QuadC(n+1) = QuadC(n) + (n+1)$.

One can consider pyramids built from triangles (see Fig. 3). Here the recurrence is $TriangleC(n+1) = TriangleC(n) + (2n-1)$. The explicit solution is $TriangleC(n) = n^2$ because it is the sum of the first n odd numbers, the famous historical example of induction proofs by Francesco Maurolico (Vacca (1909)).

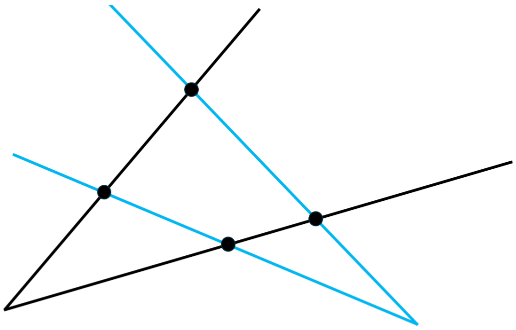


Fig. 11. Two broken lines with four crossing points.

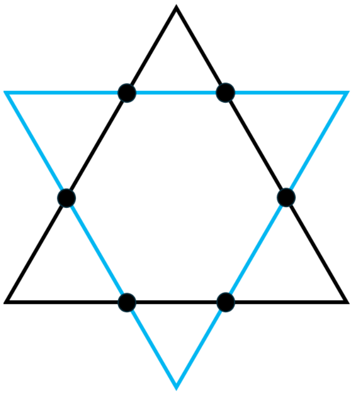


Fig. 12. Two triangles with six crossing points.

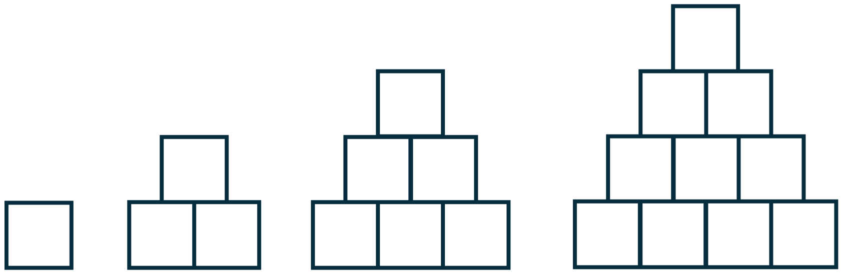


Fig. 13. Pyramid built of quadratic stones with height $n = 1, 2, 3, 4$.

After these initial pure counting tasks, we can move to more advanced geometrical tasks.

QT 5.1.7 If we place lines in a plane, we cut the plane into several subareas.

a) What is the minimal number of areas generated by placing n lines into a plane?
b) What is the maximal number $A(n)$ of areas generated by placing $n = 1, 2, 3$ lines in a plane?
c) How to estimate the maximal number of areas generated by 4 lines, if the number of areas for 3 lines is known?
d) How to estimate $A(n + 1)$ when $A(n)$ is known for any n?
e) Can you find an explicit description of $A(n)$?

Solutions and expectations We have formulated this task as a sequence of smaller tasks in order to ask students to acquire some experience step by step before being able to find the final solution.

a) Here is the answer easy. Drawing n parallel lines into a plane, we get exactly $n + 1$ areas. One can use induction here by arguing that adding one new line parallel to already placed n parallel lines increases the number of areas by 1 (dividing one area into two subareas).
b) In Fig. 4 we see the solutions for $n = 1, 2,$ and 3. $A(1) = 2$, $A(2) = 4$, $A(3) = 7$.
c) Similarly as in QT 5.1.3, adding one new line to three lines we get three new crossing points (see Fig. 5). These three crossing points divide the new line into 4 segments. Two segments of finite length are laying between two neighboring crossing points. Another two segments are infinite, running from a crossing point to infinity (see Fig. 5). Each of these 4 segments divides an area into two sub-areas, and so one gets 4 sub-areas more. Hence, $A(4) = A(3) + 4$.
d) Applying the strategy from c) students can discover $A(n + 1) = A(n) + (n + 1)$.
e) Applying the derived recurrence, one obtains $A(n) = A(1) + 2 + 3 + \ldots + n = 1 + (1 + 2 + 3 + \ldots + n) = 1 + (n + 1) + n/2$.

QT 5.1.8 What is the maximal number $CA(n)$ of areas obtained by placing n circles in a plane?

Solutions and expectations Now, we expect that students can solve this step by step using their experience from QT 5.1.5 and QT 5.1.7. Following Fig. 14 we see that $CA(1) = 2$, $CA(2) = 4$, and $CA(3) = 8$. Students know that they can add a new circle crossing each of the already placed circles in 2 new crossing points. In this way the $(n + 1)$th circle has $2n$ crossing points, and so is partitioned into $2n$ segments. One segment is the part of the circle between two consecutive crossing points, and all segments are of finite length. Each segment of the new circle partitions an area into two subareas. Hence, $CA(n + 1) = CA(n) + 2n$. Students already know this recurrence equation (QT 5.1.5 c)) and can derive $CA(n) = n(n - 1) + 2$. Note that this additional 2 is coming from $CA(1) = 2$ ($CC(1) = 0$ (in QT 5.1.5 c)).

Additional exercises One can pose the question about the maximal number of areas generating by placing different geometrical objects. If one takes n ellipses, the recurrence will be $EA(1) = 2$ and $EA(n + 1) = EA(n) + 4n$ by the same argument as in QT 5.1.8. If one takes triangles crossing each other in finitely many points only, one obtains $TA(1) = 2$ and $TA(n + 1) = TA(n) + 6n$.

If one takes broken lines, one gets $BLA(1) = 2$ and $BLA(n + 1) = BLA(n) + 5n$.

One can relate solving tasks by constructive induction to programming. All recursive descriptions of functions can be used for developing programs computing $f(1)$, $f(2)$, $f(3)$, ... $f(n)$ in this order for a given n. Later it is a good preparation for implementing f as a recursive function.

Up to this point teacher can move to algorithmic tasks with the aim to design algorithms solving a problem.

QT 5.1.9 Teacher introduces the coloring of maps in a plane and explains that four colors are always sufficient for coloring a map in such a way that two neighboring countries have always different colors (no curve as a border between two countries has the

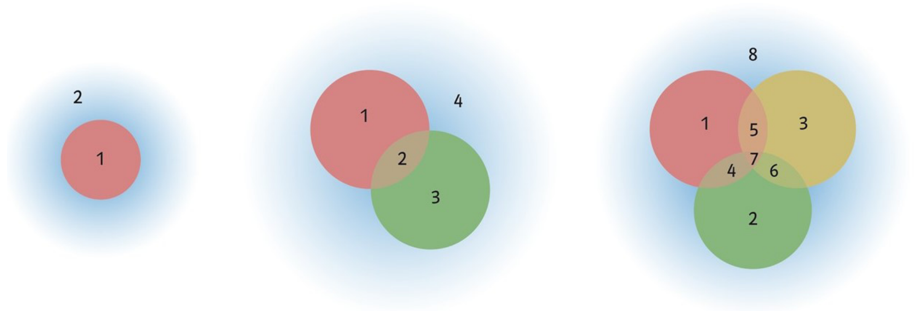


Fig. 14. Maximal number of areas obtained by 3 circles.

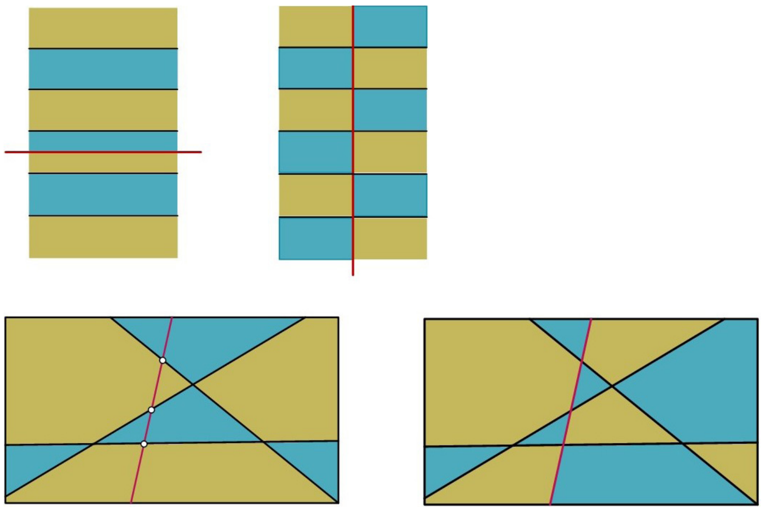


Fig. 15. Two colors are sufficient to color maps generated by 3 lines.

same color on both sides). Students can do coloring for several examples of maps. Now students have to consider only special maps that are generated by placing some lines into a plane. Students know already that for n lines the number of subareas must be between $(n + 1)$ and $n(n + 1)/2$ (see QT 5.1.7).

a) How many colors are sufficient to color maps generated by $n = 1, 2, 3$ lines?
b) If one knows the solution for three lines, how can one use it to color the map after adding one more line?
c) How to use a coloring with minimal number of colors for maps generated by n lines to find coloring for maps generated by $n + 1$ lines?

Solutions and expectations a) Students can discover that surprisingly for small n's two colors always suffice (see Fig. 4 and Fig. 15).

b) To master the step students frequently need some support. Teacher can give the following two hints. First, teacher can ask: "If you have a valid coloring by two colors, will you get a valid coloring if you exchange the colors?" Students would immediately observe that the answer is "Yes". Now the teacher places a new line into a valid coloring of a map and asks to fix all segments (borders between two subareas) that have the same color on both sides. Students will recognize that all such segments are segments of the new line, and that all segments of the new line have this property. Now students have a good chance to discover the strategy exchanging the coloring on one side of the line. In this way students can show, that two colors are sufficient.

c) To generalize the idea of *b)* to a general case of arbitrary many lines is easy.

QT 5.1.10 How many colors are sufficient to color maps generated by placing n circles (ellipses) in a plane?

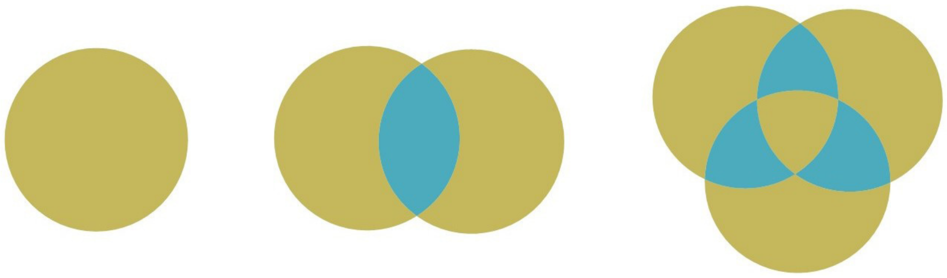


Fig. 16. Two colors are sufficient to color maps of three circles.

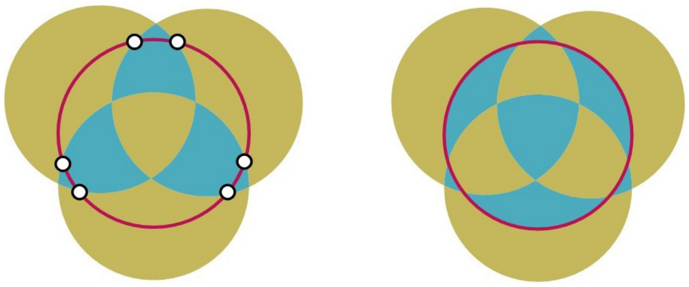


Fig. 17. A fourth circle added and two colors are still sufficient to color this map.

Solutions and expectations We expect that students will recognize that two colors are sufficient in this scenario as well (see Fig. 16). Placing a circle (ellipse) into a valid coloring all segment of the new circle (ellipse) have the same color on both sides (see Fig. 17). Hence it is sufficient to exchange colors inside of the circle (ellipse) and keep the coloring outside. For sure, one can as well exchange the coloring outside and keep the coloring inside. Hence, two colors are sufficient.

Additional exercises Instead of placing one kind of objects as lines or circles one can consider any combination of lines, circles, ellipses at once, coloring with two colors will always work. It does not matter whether the next object is a line or an ellipse, we always master to create a valid coloring.

This works also for broken lines, if we forbid common segments, i.e. two broken lines may cross only in finitely many points. In Fig. 18 we see that without this property already two broken lines may require at least 3 colors. Subareas 2 and 3 must have a different color than 1, because both are neighbors of 1. But subareas 2 and 3 must have different colors, because they are neighbors. Hence, 3 colors are necessary. Now one can ask students to draw broken lines in such a way that 4 colors are needed. The students have to be motivated to discuss why it is so. The reason is that the segments of the last placed object have different properties. Some of them have the same color on both sides and some of them have different colors on the sides of the segment.

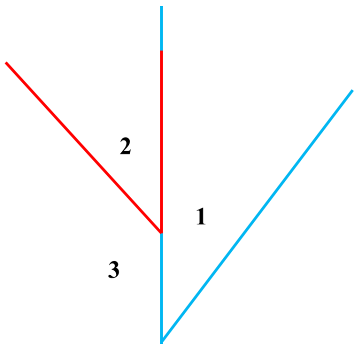


Fig. 18. Broken lines with common segments.

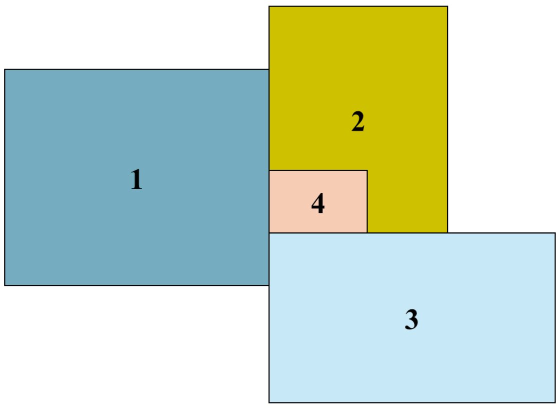


Fig. 19. Rectangles with common segments.

Hence, one cannot use the developed strategy of exchanging colors on one side of the new object.

This task can be extended to triangles, rectangles or other geometric objects. As long as we require a a finite number of crossing points between two objects, two colors are always sufficient. If we allow common segments of two objects, then 4 colors are necessary. In Fig. 19 we see an example of a map generated by rectangles that requires 4 colors. One can argue for that by the fact that each subarea is a neighbor of each other. One can ask students to place triangles in such a way that 4 colors are needed as well.

5.2. *Searching and Sorting by Constructive Induction*

After training constructive induction by solving geometric problems as in Chapter 5.1 it is quite easy to apply constructive induction to searching and sorting. Alternatively, one is also allowed to introduce constructive induction by solving searching problems and sorting because these tasks can be naturally solved by induction.

We recommend to start this topic by the following funny searching problem.

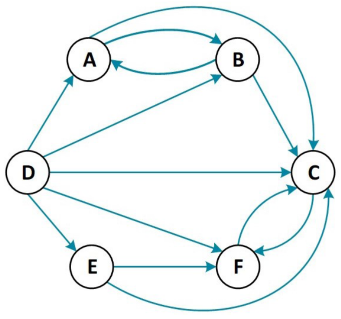
QT 5.2.1 Who is the agent? We have n persons and we know that one of them is an agent. An agent is a person who knows everybody, bot nobody knows him (in a part of the year, you can exchange "agent" for "Saint Nicolas"). You are allowed to ask any person A whether A knows person B, and you will get the correct answer. The challenge is to find the agent by posing as few questions as possible.

Solutions and expectations A good idea is to start dealing with this problem by abstract, but transparent data representations (as an example see Fig. 20 left). One can represent the n persons by vertices of a graph and draw an arrow from A to B if A knows B. In this way the relation of knowing persons is represented by a directed graph. The agent is the vertex X from which there are arrows to all other vertices, and there is no arrow to X. The challenge is to ask for the existence of arrows until the agent is recognized in such a way that the number of questions is minimized.

Another representation of the problem is the adjacency matrix of the graph (Fig. 20 right). If there is 1 on the crossing of row X and column Y, then X knows Y. If 0 is in this crossing, then X does not know Y. The agent (D in Fig. 20) is W, if all values in row W are 1's and all values of the column of W are 0's. The task is in asking for values on different positions of the matrix until the agent is fixed, and the challenge is in minimizing the number of questions. We see that the number of possible edges (items of the adjacency matrix) is exactly $n^2 - n$, i.e., $n^2 - n$ questions are always sufficient for finding the agent among $n\ persons$. The challenge is to do it with essentially smaller number of questions.

After getting a good feeling about the problem teacher could help by asking whether it is possible to have 2 or more agents. Students have to recognize that this is impossible. If X and Y would both be agents, then X would know Y and Y has to know X. But then none of X and Y could be an agent, because somebody else knows each of them. Note, that in our task we assume that one of the persons is an agent. Without this assumption the problem would be a little bit different.

Now, the crucial point is to recognize that one question can reduce the number of candidates for the agent by 1. If you ask "A, do you know B?", you can get two



	A	B	C	D	E	F
A	▨	1	1	0	0	0
B	1	▨	1	0	0	0
C	0	0	▨	0	0	1
D	1	1	1	▨	1	1
E	0	0	1	0	▨	1
F	0	0	1	0	0	▨

Fig. 20. Directed graph and adjacency matrix for the example of 6 persons.

answers. If A knows B, then B cannot be the agent. If A does not know B, then A cannot be the agent. Hence, one question can reduce the size of the problem (the number of candidates) by 1.

Two different algorithms can be developed. The recursive one poses a question and reduces the problem instance to the call for solving a problem instance of size smaller by 1. The algorithm based on dynamic programming will start with two persons and pick one of them as a candidate for the agent by one question. Alternatively one can start with one person and saying that he is the agent without posing any question. After that you repeat the induction step. In one step you add one new person X into the game and ask whether X knows the current candidate. Depending on the answer, you fix the new candidate. Both solving strategies need $n - 1$ questions to find the agent among n persons.

QT 5.2.2 We have n cards with a well defined linear order. One could also create cards that have an integer on one side and on the other side they are identical. All cards are covered and the task is to find the card with the maximal value. The only one allowed operation called "comparison" is to reveal two cards and compare their values. The challenge is to find the card with the maximal value by the smallest number of comparisons. Another implementation of the game can be done with n identical objects with different weights. The only operation allowed is using a scale to compare the weights of two objects.

Solutions and expectations After solving QT 5.2.1 this task is easy. Students immediately observe that comparing the values of two cards one can exclude one of the cards as a candidate for the maximum and so reduce the size of the task by 1. Again, using this fact one can design a recursive algorithm or comparing the winner of the last comparison with the next object.

QT 5.2.3 One has n covered cards as in QT 5.2.2 distributed chaotically on the table. The task is to find the card with a concrete value x. Students have to play this search several times for different values and note the number of cards revealed until the card with the correct value x has been found.

Solutions and expectations Students will recognize that this kind of searching in chaos is a matter of luck. Sometimes, the card searched for is found quickly, sometimes almost all cards have to be revealed. On average, about half of the cards are revealed until the card searched for has been found. The conclusion is that if one is frequently searching for something, it is reasonable to invest effort in creating a order of all objects to make searching more efficient.

QT 5.2.4 Binary Search We have n covered cards in a line on positions 1 to n and we know that the cards are ordered from the smallest one on the left side and the largest one on the right side. We hold one revealed card in our hand and are searching for the position containing this card. The challenge is to find this position by revealing as few cards as possible.

Solutions and expectations For this task it is important that the values of the cards are unknown and may be very different. If one uses only cards of values 1 to n, then the card i is on the position i and we do not need to search for it.

To involve students in the discovery of binary search, teacher can ask the following question. "Reveal one arbitrary card and compare its value with the card you are searching for. What can you conclude from this comparison?" Students have to recognize that if x is larger than the number on the revealed card, then the position of the card with the value x must be placed to the right of the revealed card and the cards (positions) to the left of the revealed card are out of the game. If x is smaller than the value of the revealed card, then the position of the card with value x must be to the left of the revealed card.

After this discovery students have to recognize that the best way of choosing a card to be revealed is to take the position in the middle. This way, revealing one card narrows the search space to half it's former size, or offers the card searched for. Students have to play with numbers to see that for $n = 1'000$ cards 10 revealed cards always suffice, for $n = 1'000'000$ 20 revealed cards are enough, and for n =$1'000'000'000$ at most 30 revealed cards guarantee a successful searching. We recommend here to use the opportunity (if students do not know logarithmic functions) to introduce the discrete logarithm of n by base 2 as the smallest number k such that $2^k \geqslant n$. The only pre-knowledge required is to know potence functions c^n for any constant c. One can introduce notation $dislog_c(n)$ for the discrete logarithm of n by base c.

Additional exercises The following task is a generalization of QT 5.2.2 (searching for the maximum or for the minimum in n unsorted cards). The task is to find the three cards with highest values from the set of covered cards. The simplest idea based on induction is to take three arbitrary cards and sort them by 3 comparisons. Let $MAX_3(n)$ denote the number of comparisons of searching for the three cards with highest values. Hence, the base is $MAX_3(3) = 3$. Now, one can take one card after another and compare, if necessary, with all three up to now maximal cards. This way one could get $MAX_3(n + 1) = MAX_3(n) + 3$, which offers finally $MAX_3(n) = 3 * n - 6$.

Now, one can improve this algorithm by the idea of binary search. If $max_3 \leqslant max_2 \leqslant max_1$ are the three maximal values up to now, then the new value x is compared with max_2. If $x < max_2$, then x is still compared with max_3. If $x > max_2$, then x is still compared with max_1. In this way one obtains the recurrence $MAX_3(n + 1) = MAX_3(n) + 2$, and so a faster algorithm with the complexity $MAX_3(n) = 2n - 3$. For strong students teacher may still ask to improve this algorithm. Consider the following strategy for searching for two largest elements. Let be $n = 2^k$ for some k. To estimate the maximum one can play a tennis tournament with $k = dislog_2(n)$ rounds (see Fig. 21). After that it is obvious that the second largest value must be among the k elements (red in Fig. 21) that directly lost in the comparison with the maximum. To find the maximum of these k elements costs $k - 1$ comparisons. So, the number of comparisons altogether is $(n - 1) + (k - 1) = n + dislog_2(n) - 2$, which is much

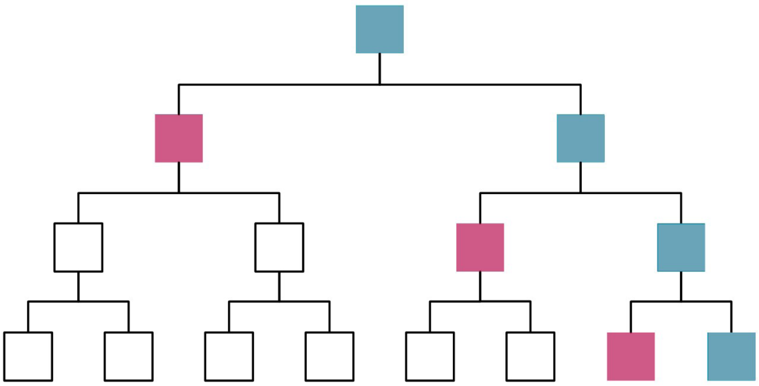


Fig. 21. Improving the efficiency of an algorithm with the graph of a tennis tournament.

better than $2n$. If one uses this strategy for searching for the 3 maximal elements, then the sufficient number of comparisons is

$$\begin{aligned}&(n-1)+(k-1)+(k-2)+dislog_2(k)\\&=n+2k+dislog_2(k)-4\\&=n+2dislog_2(n)+dislog_2(dislog_2(n))-4\end{aligned}$$

which is much better than $3n-6$.

Another interesting task we formulate so, that one does not need to work with covered cards. We have n runners taking part in an endurance race. All finished the race and no two have the same time because the measurement is very exact. You are allowed to ask questions such as ”Runner A, have you been faster that runner B?”, and you will get the correct answer. The challenge is to simultaneously estimate the winner as well as the last runner by as few questions as doable. The motivation is that the last runners will get a special price as the strongest fighter (some endurance races really do it). Using constructive induction you ask the question for arbitrary two runners A and B. Max is the candidate for the winner and Min is the candidate for the last runner. If A was faster than B, then Max = A and Min = B after the first question. Then one asks the next runner and after comparing him with Max and Min, one can update Max and Min. Hence, $MinMax(2)=1$ and $MinMax(n+1)=MinMax(n)+2$, which offers $MinMax(n)=2n-3$. This task can be solved better without induction. Consider n is even. One partitions n runners into $n/2$ pairs and compares the runners inside of pairs by $n/2$ comparisons. Then one can estimate the absolute winner by $n/2-1$ comparisons from the winners of the first $n/2$ comparisons. Analogously, one can estimate by $n/2-1$ comparisons the very last runner from the $n/2$ losers of the first comparison round. Altogether the number of comparisons is $n/2+2(n/2-1)=3/2n-2$. This is the fastest possible algorithm. Interestingly one can get this optimal complexity by ”divide and conquer”. One splits n runners in two equally-sized groups and estimates recursively the minimum and the maximum in each of the two groups. Finally, the minima and the maxima are compared by two comparisons. This offers the recursion $MinMax(n)=2MinMax(n/2)+2$ and $MinMax(2)=1$. We do not recommend to go from this recurrence to the explicit representation of the function because it is too difficult for high school pupils. On the other hand, one can easily verify that $3/2n - 2$ is the solution of the recurrence equation above. One inserts $3/2n - 2$ for $MinMax(n)$ on the left side of the equation and then replaces $MinMax(n/2)$ by $3/2(n/2) - 2$ on the right side of the equation. A simple calculation shows that the left side is equal to the right side.

QT 5.2.5 a) There are 9 objects that all look the same, but one of them is heavier than all other ones, and all others have the same weight. How many comparisons on a scale (see Fig. 22) do you need to find the heaviest one?

b) The same but you have 27 identically looking objects. Can you fix the heaviest one by three times weighing?

c) You are allowed to weigh k times. What is the maximal number of identical looking objects for which you can determine which is the one that is heavier than all the others?

Solutions and expectations a) This task opens a new dimension when taking the complexity as the number of comparisons. It is not required in the task formulation that you are only allowed to compare one single object with another single object. One can weigh a collection of objects against another collection of objects. This can be a hint if the class does not master this challenge. Another help could be to start with 3 objects with one comparison and then to extend it to 6 objects with 2 comparisons. The solution is that two comparisons for 9 objects are sufficient. In the first weighing, one compares the weight of two groups each of which has three objects. If one group is heavier than the other one, then the heaviest object is one of the 3 objects of the heavier group. If the weight of both groups is equal, then the heaviest object is among the 3 objects not included in the comparison. This is the classical constructive induction (note, that the parameter is the number of comparisons), because by one comparison we reduce the size of the problem from 9 objects to 3 objects. Hence, $W(1) = 3$ and $W(2) = 9$ if $W(i)$ is the number of objects from which the heaviest can be found by i comparisons.

If the class does not discover the above strategy by its own, the teacher can help by the following questions. Put 4 objects on each side of the scale. If the weights are equal, what can you conclude? If one group of 4 objects is heavier than the other, what can you conclude?

b) With the experience with solving a), the class will recognize that $W(3) = 27$ (or even $W(3) = 3 * W(2)$). One compares two groups of size 9 and this comparison

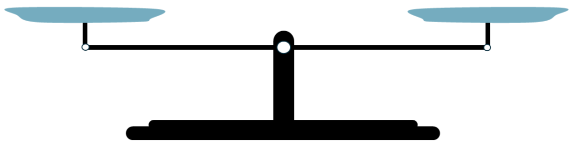


Fig. 22. Scale to compare the weight of objects.

reduced the number of objects candidating for the heaviest one from 27 to 9, i.e., to one third.

c) c) Now the inductive strategy is already discovered and the class can establish $W(n + 1) = 3 * W(n)$ and take $W(1) = 3$ or $W(2) = 9$ as the base. It is not hard to derive the explicit solution $W(n){=}3^n$. The class has to be able to explain the algorithm for all numbers of objects, not only for the numbers of the form 3^k. k comparisons are sufficient and necessary for the number of objects between $3^{k-1} + 1$ and 3^k.

After mastering different searching tasks as above teacher can move to sorting. If one wants to discuss partial order and linear order before dealing with sorting we recommend an appropriate design of lessons in Gallenbacher *et al.* (2023).

QT 5.2.6 Teacher proposes the following sorting inductive strategy taking one element after another and bringing it to its correct position. Start with one element that is already sorted. Having a sorted sequence of n elements take the next element and place it on the correct position in the sorted sequence.

a) Implement the placement of the new element as follows. Compare the new element with the smallest (the leftmost) element of the sorted sequence. If the new element is larger, continue to compare it with the next smallest element and move from the left to the right until the new element is larger.

   In which order 10 numbers $1, 2, 3, \dots, 10$ have to come that the number of comparisons is the smallest possible?

   In which order 10 numbers $1, 2, 3, \dots, 10$ have to come that the number of comparisons is the largest possible?

   What is the largest number of comparisons executed by the algorithm when sorting n elements?

b) Implement the placement of the next element by using binary search. How many comparisons this algorithm executes in worst case?

Expectations and Solutions a) First, students have to recognize that the number of comparisons executed depends heavily on the order of elements to be sorted. They can execute the algorithm for a few sequences of elements to see this fact.

Taking the order $10, 9, 8, 7, 6, 5, 4, 3, 2, 1$ students have to discover that this is the easiest case because the next element is always smaller than the smallest element of the already sorted subsequence. In this way 9 comparisons are sufficient to get the sorted sequence $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$. We expect that students can generalize this experience for arbitrary n, and so fix that $n - 1$ comparison suffices for elements in the opposite order.

Taking the elements in the already sorted order $1, 2, 3, 4, \dots, 10$ will cause the most work because each next element will be compared with all elements of the already sorted sequence. In this case the number of comparisons is $1 + 2 + 3 + \cdots + 9 = 45$. After that students should be able to generalize this worst-case scenario to n element and establish the upper bound on the number of comparisons

$$1 + 2 + 3 + \cdots + (n-2) + (n-1) = n * (n-1)/2.$$

b) Students start with executing this algorithm with covered cards and recognize how the placement of the next element in a sorted sequence of k elements can be done by binary search by at most $dislog_2(k)+1$ comparisons. After that they see that the maximal number of comparisons executed is

$$1 + 1 + dislog_2(3) + \cdots + 1 + dislog_2(n-1).$$

Now teacher can help to upperbound this formula by $(n - 1) * (1 + dislog_2(n-1))$.

QT 5.2.7 Teacher proposes sorting by repeatedly using the already known algorithm for finding the maximum in an unsorted sequence of elements. The induction step is obvious. One finds the maximum of the current unsorted sequence and moves it as the smallest element to up to now (already) sorted sequence. In this way estimating the maximum decreases the number of elements in the unsorted sequence by 1.

a) How many comparisons always suffice to order n elements by the strategy described above?
b) Execute this algorithm with numbers written on covered cards. Are there some unordered input sequences that use less comparisons than in the worst case established by a)?

Expectations and solutions a) Students already know that estimating the maximum of n unsorted elements (n covered cards) costs exactly $n - 1$ comparisons. Repeating this algorithm for sequences of length n, $n - 1$. $n - 2$, ... , 2 causes altogether

$$(n-1) + (n-2) + (n-3) + \cdots +1 = n * (n-1)/2$$

comparisons.

b) If this algorithm is executed with covered cards, the number of comparisons is always the same, i.e., the order of elements in the input sequence does not matter. Students have to be able to discover it by their own. For students it is interesting here to see the contrast to QT 5.2.6 a), where the number of comparisons executed depends heavily on the order of elements in the input sequence.

Additional exercises Teacher can allow only one very simple operation on a sequence of elements. One can compare two neighboring elements only and depending on the result of the comparison exchange their order. Students are asked to develop an algorithm that starting with an arbitrary potentially unordered sequence of elements finishes with the sorted sequence of these elements. In this way students are forced to develop Bubblesort or something very close to that. After developing Bubblesort, students can optimize their algorithm by formulating a criterion for stopping the work as soon as the sequence is already sorted. All this can be implemented by a program. Additionally students can search for initial sequences causing the smallest possible number of comparisons and for the opposite worst case input sequence.

5.3. *Arithmetics*

One of the fundamental tasks of informatics is the development of abstract representations of some objects in such a way that the work with these abstract representations is efficient and transparent. The development of number representations and design of algorithms for executing fundamental arithmetic operations is the most impressive historical example of this kind. Constructive induction has played a majorrolein the development of arithmetics.

If we look at arithmetics in primary school, we see that

$$\begin{aligned}
&a + b = a + (b - 1) + 1,\ a + 0 = a\\
&a - b = a - (b - 1) - 1,\ a - 0 = a\\
&a * b = a * (b - 1) + a,\ a * 1 = a\\
&a : b = (a - b) : b + 1,\ b : b = 1,\ k : b = 0 \text{ for } 0 \leqslant k < b
\end{aligned}$$

We see that in this inductive descriptions of the basic arithmetic operations the parameter is always the value of one of the two operators (b for addition, subtraction, and multiplication, and a for division). Here we see that adding $1\ (+1)$ and subtracting $1\ (-1)$ is sufficient to compute any addition and any subtraction of two integers. Multiplication can be executed by additions only, and division can be calculated by repeated subtraction. Teacher in high school has the opportunity to discuss it and let the pupils program the execution of the four basic arithmetic operations by only using operations $+1$, -1, and the test whether the value of some variable is 0 in a while-loop.

When dealing with arithmetic operations in high schools we recommend to take the length of the number representation (the number of digits in the decimal representation) of one of the arguments as the parameter. Here we are showing how our school multiplication algorithm has been developed by constructive induction.

QT 5.3.1 Assume, one can multiply by any digit (a number of the representation length 1) and by 10.

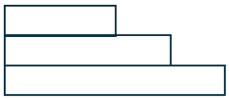
a) Consider the following two multiplications:

$$\begin{array}{r}
1856 * 7 = \\
\hline
12992
\end{array}
\qquad
\begin{array}{r}
1856 * 37 \\
\hline
12992 \\
5568 \\
\hline
68672
\end{array}
\qquad
\begin{array}{l}
\boxed{1856 * 7} \\
\boxed{(1856 * 3) * 10}
\end{array}$$

Explain how the multiplication by a digit can be used for the multiplication by numbers of length 2 (consisting of two digits).

b) Consider the multiplication by a three-digits number.

$$\begin{array}{r}
1856 * 937 \\
\hline
12992 \\
5568 \\
16704 \\
\hline
1739072
\end{array}$$



Write in the empty boxes which calculation has been executed in the corresponding row. Explain how the multiplication by a two-digits number can be used to calculate the multiplication by any three-digits number.

c) Let $y = dcba$ be the representation of a number of length 4, where a, b, c, and d are digits. Explain the meaning of the follwoing abstract alculation for an arbitrary number x.

$$x * y = x * dcba = x * cba + x * d * 10 * 10 * 10$$

Use this formula to compute $1856 * 2937$ ($x = 1856$, $y = 2937$).

d) Explain in general for any length $n + 1$ of $y = y_n y_{n-1} \cdots, y_1, y_0$, how to calculate $x * y$ when one can calculate multiplication by numbers of length n.

Solutions and expectations a) We expect that the students recognize that multiplication by a two-digit number can be executed by two multiplications by a digit, one multiplication by 10 and one addition. At this point teacher can introduce the following abstract description of this calculation. First starting with the given example,

$$1856 * 37 = (1856 * 7) + (1856 * 3) * 10$$

Then taking $y = ba$ for digits a and b as an abstract representation of a two-digit number, we can describe the calculation as follows:

$$x * y = x * ba = (x * a) + (x * b) * 10$$

b) Students can easily insert the missing calculations in the empty boxes:

$$1856 * 7 = 12992$$
$$1856 * 3 * 10 = 55680$$
$$1856 * 9 * 10 * 10 = 1670400$$

In general, the students can claim that the multiplication by a three-digit number can be calculated by three multiplications by one digit, three multiplications by 10 and two additions. But teacher asks students to describe the process of multiplying by three-digit numbers by multiplying by a two-digits number. For the concrete example it looks like:

$$1856 * 937 = 1856 * 37 + 1856 * 9 * 10 * 10$$

In general taking $y = cba$ students should be able to derive the following description of the multiplication:

$$x * y = x * cba = x * ba + x * c * 10 * 10$$

c) After the experience with the abstract representations from b), students should be able to understand this calculation description and use it for concrete numbers.

$$1856 * 2937 = 1856 * 937 + 1856 * 2 * 10 * 10 * 10$$

$$= 1739072 + 3712000$$
$$= 5451072$$

d) This task is only for strong students. But all students can try to solve the task for multiplyers of length 5 as follows:

$$x * y = x * edcba = x * dcba + y * e * 10 * 10 * 10 * 10$$

In general it looks as follows:

$$x * y = x * y_n y_{n-1} \cdots y_2 y_1 y_0 = x * y_{n-1} \cdots y_2 y_1 y_0 + x * y_n * 10^n$$

Additional exercises If the abstract representation of the calculation is too hard for the class, then teacher can omit it and use it for concrete numbers only. For instance:

$$4268 * 31759 = 4268 * 1759 + 4268 * 3 * 10^4$$

For strong classes teacher can introduce another way of expressing the calculation of $x * dcba$, and ask for explaining the difference to the previous one.

$$x * dcba = x * dcb * 10 + x * a$$

Now teacher can ask which of these calculations is more efficient. The answer is the new one, because we have only one multiplication by 10 in this induction step. Both calculations use one addition, one multiplication by a digit, and one multiplication by a number with a shorter representation.

One can also use constructive induction to develop the common division algorithm for $a : b$, where the parameter is the representation length of a (for details see Hromkovic and Lacher (2024)).

QT 5.3.2 One has to design a strategy for computing x^6 (the sixth power of x) for an arbitrary number x. For the multiplications the only allowed arguments are x itself and the results of previous multiplications. The following three strategies are proposed:

Strategy 1:	Strategy 2:	Strategy 3:
$x^2 = x * x$	$x^2 = x * x$	$x^2 = x * x$
$x^3 = x^2 * x$	$x^3 = x^2 * x$	$x^4 = x^2 * x^2$
$x^4 = x^3 * x$	$x^6 = x^3 * x^3$	$x^6 = x^4 * x^2$
$x^5 = x^4 * x$		
$x^6 = x^5 * x$		

a) We are searching for a strategy using a minimal number of multiplications. Which of these strategies is the best? Does there exist a strategy for computing x^6 with less than 3 multiplications?
b) Find two different strategies for computing x^{24} by 5 multiplications only.
c) Find the best strategy for computing x^{64}.

d) Find a strategy to compute x^{13} by 5 multiplications.
e) Find a strategy for computing x^{45} by 8 multiplications.

Solutions and expectations a) Students easily observe that both strategies 2 and 3 use 3 multiplications, and so they are better than strategy 1. To answer the second question may be hard for the students and teacher may help here. For next tasks it is helpful to see the argument that 3 multiplications are necessary to compute x^6. If we have only x, the only one multiplication we can do as first is $x^2 = x * x$. Now we have two numbers, x and x^2. The biggest number we can obtain is x^4 as $x^2 * x^2$. So, one cannot compute any number larger than x^4 by two multiplications and the given rules.

b) Using the ideas of strategies 2 and 3 from a) students can compute x^{24} by 5 multiplications in two different ways:

$x^2 = x * x$	$x^2 = x * x$
$x^3 = x^2 * x$	$x^4 = x^2 * x^2$
$x^6 = x^3 * x^3$	$x^8 = x^4 * x^4$
$x^{12} = x^6 * x^6$	$x^{16} = x^8 * x^8$
$x^{24} = x^{12} * x^{12}$	$x^{24} = x^{16} * x^8$

c) Here students should discover the following strategy. Try to calculate as largest number as possible (to get the highest power of x) until your results remain smaller than x^{13}. Then try to get x^{13} by multiplying some of the numbers (powers of x) already calculated.

The first phase here is:

$x^2 = x * x$
$x^4 = x^2 * x^2$
$x^8 = x^4 * x^4$

Because x^{16} is bigger than x^{13} we stop the first phase here.
The second phase uses the fact that $13 = 8 + 4 + 1$.
$x^{12} = x^8 * x^4$
$x^{13} = x^{12} * x$

Teacher has to introduce and discuss this strategy if students do not discover it or present their solutions as an ad-hoc solution only. But this strategy for computing x^{13} by 5 multiplications is unique, so students must calculate as above if they found a solution.

d) After discussing the general strategy in c) students have to be able to apply it again for another power of x.

The first phase is:

$x^2 = x * x$
$x^4 = x^2 * x^2$
$x^8 = x^4 * x^4$
$x^{16} = x^8 * x^8$
$x^{32} = x^{16} * x^{16}$

The second phase uses $45 = 32 + 8 + 4 + 1$.

$x^{40} = x^{32} * x^8$
$x^{44} = x^{40} * x^4$
$x^{45} = x^{44} * x$

QT 5.3.3 Use the constructive induction to explain why x^{2^i} can be always computed by i multiplications.

Solutions and expectations If this formulation of the task is too hard for the class, teacher can split the task in the sequence of subtasks. How many multiplications are sufficient to compute x^2, $x^4 = x^{2^2}$, $x^8 = x^{2^3}$, $x^{16} = x^{2^4}$, ….

x, $x = x$, $x = x$, $x = x$, …. For the concrete cases students see immediately that x^2 can be computed by 1 multiplication, $x^4 = x^{2^2}$ by 2 multiplications, $x^8 = x^4 * x^4$ by 3 multiplications, etc.

In general $x^{2^{n+1}} = x^{2^n} * x^{2^n} = x^{2 * 2^n}$

Hence one multiplication suffices to move from x^{2^n} to $x^{2^{n+1}}$.

QT 5.3.4 Explain why for $n \geqslant 4$, x^n can be always calculated by $2 dislog_2(n) - 2$ multiplications.

Solutions and expectations Students already know the general strategy of two phases and so can try to estimate the number of multiplications for each phase. If n is a power of two, i.e., $n = 2^m$, then $m = dislog_2(n)$ multiplications are sufficient to compute x^n in the first phase and there is no second phase. If $2^{m-1} < n < 2^m$, then the first phase consists of $m - 1 = dislog_2(n) - 1$ multiplications.

For the second phase the m values x, x^2, x^4, … , $x^{2^{m-1}}$ are available. The key point now is that each of these values can occur at most once in a product of the second phase, and so the number of multiplications is at most $m - 1 = dislog_2(n) - 1$. Why each computed value (a power of x) is used at most once? Because it is the same as paying the value n by binary coins of values 1, 2, 4, 8, … , 2^{m-1} in such a way that the number of coins used is minimal. In this case we cannot use any coin twice because we can exchange two coins of the same value by one coin with the double value.

QT 5.3.5 Find the n-th powers of x, i.e., x^n for any n such that our strategy uses exactly $2dislog_2(n) - 2$ multiplications.

Solutions and expectations One can start with small powers. Students can find that x^7 is such a case $dislog_2(7) = 3$ and so $2dislog_2(7) - 2 = 4$. Our strategy uses 4 multiplications.

$x^2 = x * x$
$x^4 = x^2 * x^2$
$x^7 = x^4 * x^2 * x$

Another power is 15. $dislog_2(15) = 4$, and so $2 * dislog_2(15) - 2 = 8 - 2 = 6$. We need correspondingly 6 multiplications.

$x^2 = x * x$
$x^4 = x^2 * x^2$
$x^8 = x^4 * x^4$
$x^{15} = x^8 * x^4 * x^2 * x$

After this students can recognize that the powers n with the maximal number $2 * dislog_2(n) - 2$ of multiplications have the form $n = 2^m - 1$. This is because their binary representations consists of 1's only, and so we have to sum all binary coins to get the value n.

Additional exercises After QT 5.3.2 teacher can offer several challenges asking for computing different powers of x by a minimal number of multiplications in order to get more experience. One can look also for powers for which more than one optimal strategy exists. Another alternative starting point may be calculating $n * x$ by using the maximal number of additions. For instance for $10 * x$ one can proceed as follows:

$2x = x + x, 4x = 2x + 2x, 8x = 4x + 4x, 10x = 8x + 2x$

or alternatively

$2x = x + x, 4x = 2x + 2x, 5x = 4x + x, 10x = 5x + 5x$

We finish our teaching sequence in calculations with Horner schema – a prime example of using constructive induction in algorithm design.

Explanation A transparent way of describing calculation processes are circuits. An algorithm for computing x^7 can be represented as the circuit in Fig. 23.

$x^2 = x * x, x^4 = x^2 * x^2, x^6 = x^4 * x^2, x^7 = x^6 * x$

One can build circuits also for more than one input (x in our example). One can compute the value of $ax^2 + bx + c$ for four input values a, b, c, and x by the following algorithm.

$x^2 = x * x,\ ax^2 = a * x^2, bx = b * x,\ ax^2 + bx + c$

This algorithm uses 3 multiplications and 2 additions and is represented by the circuit in Fig. 24. The elements of the circuit computing the arithmetic operations are called multiplication-gate and plus-gate.

In Fig. 24 we see that we have four inputs $x,\ a,\ b, c$ (instead of only one as in Fig. 23). This circuit computes the value of $ax^2 + bx + c$ for all infinitely many possible values of input parameters $x,\ a,\ b$, and c with 3 multiplications and 2 additions. In this sense circuits in Fig. 24 is universal because it works properly and with the same complexity (number of operations executed) for all possible values of a, b, c, and x.

QT 5.3.6 Improve the universal circuit from the pervious example for evaluation $ax^2 + bx + c$ in such a way that it will have only two multiplications gates and two addition gates.

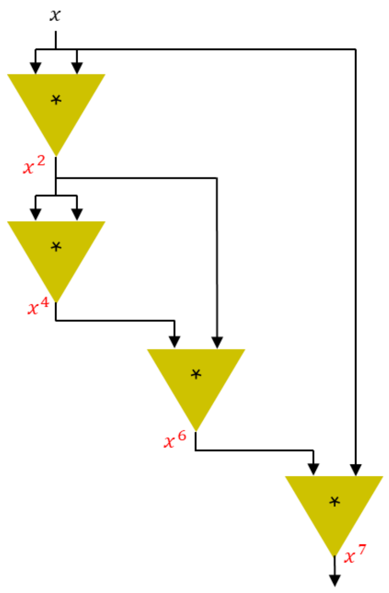


Fig. 23. Circuit to compute x^7 using 4 multiplications from the input value x.

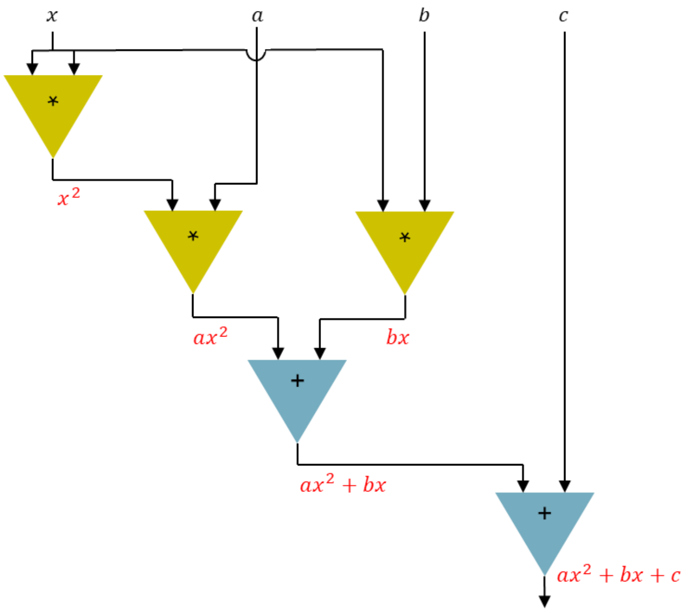


Fig. 24. Circuit to compute the value of $ax + bx + c$ from four input values.

Solutions and expectations If the class does not progress by its own, teacher can help with two hints. The first one is: ”Try to transform the expression $ax^2 + bx + c$ into an equivalent expression with only two multiplications”. If the first hint does not help, teacher may indicate that this transformation could use the distribution law. At latest after the second hint the class can discover

$$ax^2 + bx + c = (ax + b)x + c$$

and discuss the optimal circuit for the desired expression (see Fig. 25).

Teacher can make this task easier for students, if instead of working with the abstract expression $ax^2 + bx + c$, teacher works with a concrete expression like $7x^2 + 13x + 27$.

QT 5.3.7 Transform the following expressions in a form that minimizes the number of multiplications.

a) $7x^3 + 5x^2 + 28x + 13$
b) $29x^4 + 113x^3 - 52x^2 + 17x - 113$
c) $a_3x^3 + a_2x^2 + a_1x + a_0$
d) $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$

Solutions and expectations Students already know that the main idea is to use the distributive law, and so one can expect that students even find different transformations leading to the optimal solutions.

a) $7x^3 + 5x^2 + 28x + 13 = (7x^2 + 5x + 28)x + 13 = ((7x + 5)x + 28)x + 13$
$7x^3 + 5x^2 + 28x + 13 = (7x + 5)x^2 + 28x + 13 = ((7x + 5)x + 28)x + 13$

There are exactly three multiplications $7x$, $(7x + 5) * x$, and $((7x + 5) * x + 28) * x$ to be executed. For training reasons teacher may ask to draw the circuits that correspond to the desired expression.

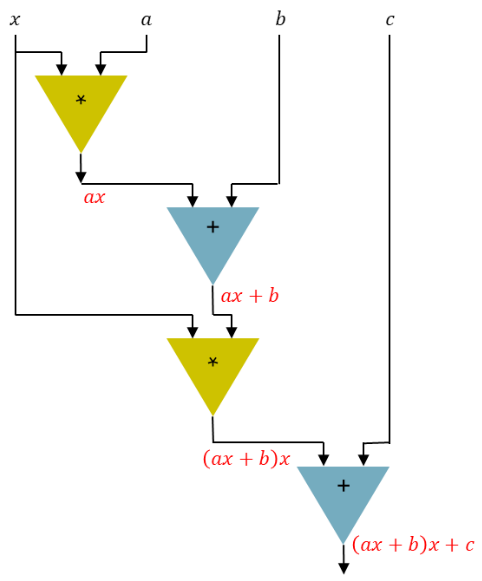


Fig. 25. Circuit to compute the value of $(ax + b) * x + c$.

b) $29x^4 + 113x^3 - 52x^2 + 17x - 113$
$= (29x^3 + 113x^2 - 52x + 17)x - 113$
$= ((29x^2 + 113x - 52)x + 17)x - 113$
$= (((29x + 113)x - 52)x + 17)x - 113$
$29x^4 + 113x^3 - 52x^2 + 17x - 113$
$= (29x + 113)x^3 - 52x^2 + 17x - 113$
$= ((29x + 113)x - 52)x^2 + 17x - 113$
$= (((29x + 113)x - 52)x + 17)x - 113$

Students see that the sufficient number of multiplications is 4, and could get already the general idea that polynomials of degree n can be evaluated by n multiplications. Teacher can ask students to count the number of additions in this polynomial evaluation form and fix that the number of executed additions is also four.

c) Students have to do the same as in a) but in an abstract setting

$a_3x^3 + a_2x^2 + a_1x + a_0$
$= (a_3x^2 + a_2x + a_1)x + a_0$
$= ((a_3x + a_2)x + a_1)x + a_0$

d) $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$
$= (a_4x^3 + a_3x^2 + a_2x + a_1)x + a_0$
$= ((a_4x^2 + a_3x + a_2)x + a_1)x + a_0$
$= (((a_4x + a_3)x + a_2)x + a_1)x + a_0$

Explanations Converting the evaluation of the polynomial $a_3x^3 + a_2x^2 + a_1x + a_0$ of degree 4 into a more efficiently computable expression can be viewed as the reduction of evaluating polynomials of degree 3 into the evaluation of polynomials of degree 2.

$a_3x^3 + a_2x^2 + a_1x + a_0$
$= (a_3x^2 + a_2x + a_1)x + a_0$

This can be transparently expressed by Fig. 26

QT 5.3.8 Express the evaluation of polynomials of degree 4 as the reduction to evaluations of polynomials of degree 3 in terms of expressions as well as in terms of circuits as in the explanation above.

Solutions and explanations After the experience with QT 5.3.7 and explanations students can master this step in the development of schema of Horner easily by their own.

$a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$
$= (a_4x^3 + a_3x^2 + a_2x + a_1)x + a_0$
This is expressed in Fig. 27.

QT 5.3.9 Express in terms of circuits the evaluation of polynomials

$a_{n+1}x^{n+1} + a_nx^n + \cdots + a_2x^2 + a_1x + a_0$

of degree $n+1$ by the evaluation of polynomials of degree n. How many multiplications and additions are sufficient to evaluate an arbitrary polynomial of degree n?

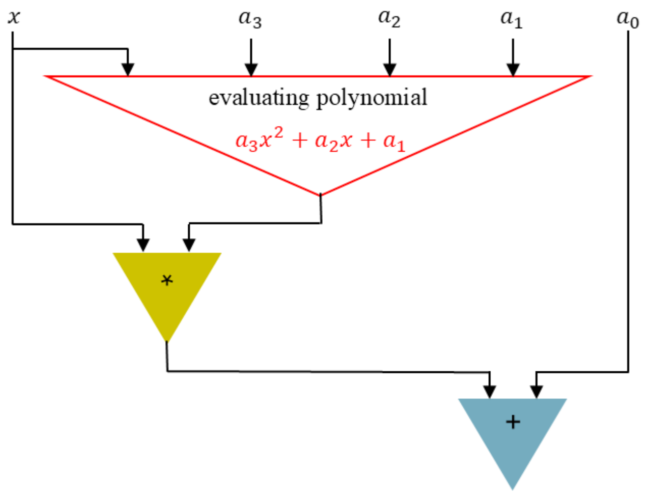


Fig. 26. Circuit reducing a polynomial degree 3 to degree 2.

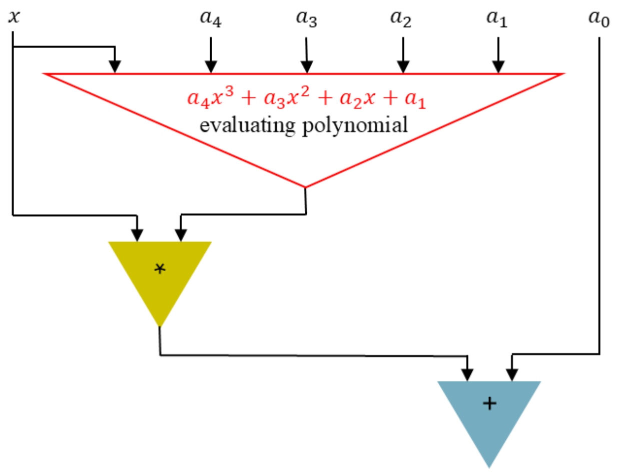


Fig. 27. Circuit reducing a polynomial degree 4 to degree 3.

Solutions and expectations After solving QT 5.3.8 students must see how the reduction from evaluating polynomials of degree $n + 1$ to evaluating polynomials of degree n works (see Fig. 28).

Teacher can additionally support students to express this reduction in terms of algebraic expressions. The main reason for this support is not related to the understandability of the reduction itself, but in missing experience with working with parametrized expressions.

$$a_{n+1}x^{n+1} + a_nx^n + a_{n-1}x^{n-1} + \cdots + a_2x^2 + a_1x + a_0$$
$$= (a_{n+1}x^n + a_nx^{n-1} + a_{n-1}x^{n-2} + \cdots + a_2x + a_1)x + a_0$$

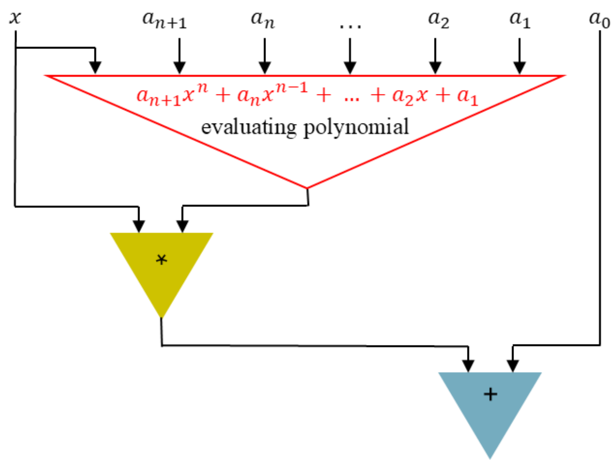


Fig. 28. Horner's schema: Circuit evaluating polynomial of degree $n + 1$.

From the presented reduction students see that by increasing the degree of polynomials by 1 the number of multiplications in their evaluation is increasing by 1, and the same is true for the number of additions. If $M(n)$ is the number of multiplications in the evaluation of polynomials of degree n, and $A(n)$ denotes the number of additions, then we showed that

$M(n + 1) = M(n) + 1$ and $A(n + 1) = A(n) + 1$

Obviously $M(0) = 0$, $A(0) = 0$, and $M(1) = 1$, $A(1) = 1$

From this students can already conclude $M(n) = n$, and $A(n) = n$.

It is well known that Horner's schema is optimal, i.e., there is no general strategy evaluating every polynomial of degree n with less than n multiplications and less than m additions. But this does not exclude that there exist special concrete polynomials of degree n that can be evaluated with less amount of work. The following task is for strong students.

QT 5.3.10 Find some polynomials such that can be evaluated with less multiplications than their degrees. Can you solve this task also if all coefficients of the polynomial are different from 0?

Solutions and expectations An easy solution is to take the polynomial $x^8 + 1$ for instance. One can express $x^8 + 1$ as $((x^2)^2)^2 + 1$ and evaluate this polynomial of degree 8 with 3 multiplications only (see Fig. 29).

If one takes polynomials with coefficient 1 by the highest degree, then one always saves one multiplication. For instance

$x^3 + 7x^2 + 15x - 6 = ((x + 7) * x + 15) * x$

and the evaluation works with two multiplications only.

A more advanced example starts with an expression that can be evaluated by two multiplications and one addition: $(x - 1)^4$ (see Fig. 30).

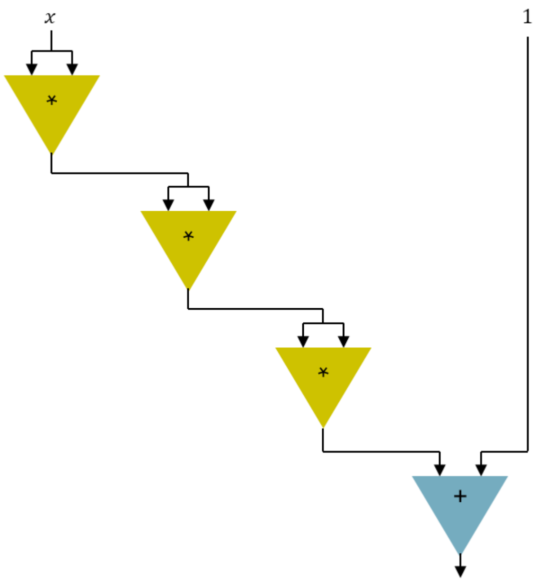


Fig. 29. Circuit with a polynomial of degree 8 that can be evaluated with 3 multiplications.

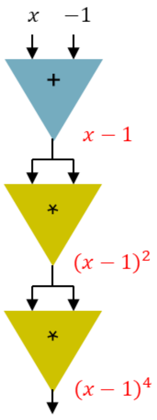


Fig. 30. Circuit for one addition and two multiplications.

But

$$\begin{aligned}(x-1)^4 &= (x-1)^2 * (x-1)^2\\ &= (x^2-2x+1) * (x^2-2x+1)\\ &= x^4-4x^3+6x^2-4x+1\end{aligned}$$

is a polynomial of degree 4.

Students can be asked to develop more such examples and increase the gap between $M(n) = n$ and really needed number of multiplications for concrete polynomials of degree n. For instance the polynomial of degree 2^n

$$(x - 1)^{2^n}$$

can be evaluated by 1 addition and n multiplications instead of n additions and 2^n multiplications.

Additional exercises Already before developing the general Horner's schema one can ask students to find strategies for the evaluation of many concrete polynomials by as few multiplications as possible.

After presenting Horner's schema as a universal strategy for evaluating polynomials that cannot be improved in general, teachers can deal with polynomials for which evaluation one can save some operations. A funny example is $2x^5 + 113x^4 + 7x^3 - 213x^2 - 71x + 29$. Here one case save one multiplication (by exchanging it for one addition) by simple modification of the polynomial to

$$(x^5 + x^5) + 113x^4 + 7x^3 - 213x^2 - 71x + 29$$

and then applying Horner's schema.

$$(((((x + x) + 113)x + 7)x - 213)x - 71)x + 29$$

Another funny task for students can be to minimize the number of multiplications when evaluating polynomials $2x^8 + 2x^6 - x^4 + 6x^2 - 7$ (4 multiplications and 5 additions) or $2x^{12} - 6x^8 + 3x^4 - 7$ (by 4 multiplications and 4 additions).

Teacher can help students to discover that one can save the execution of one multiplication if a polynomial has as many real roots as its degree. For instance $x^2 + 2x - 3 = (x - 1) * (x + 3)$, and so one multiplication is sufficient. In this way $(x - 7) * (x + 6) * (x - 1) * (x + 2)$ is a polynomial of degree 4 that can be evaluated by 3 multiplications, does not matter which are the coefficient of the polynomial. This can work also for $(x + x - 5) * (x + 2) * (x + x - 6) * (x + 3)$ if one wants to generate polynomials with leading coefficient different from 1. Students have to discuss it with teacher to understand that this does not lead to a schema better than Horner's one. There are several reasons for that. First, there are polynomials that do not have real roots. Secondly even for those that have real roots, one cannot achieve any real leading coefficient in the proposed way. Finally, and most importantly, getting the coefficient of a polynomial as inputs, one needs first compute the roots of this polynomial before applying the knowledge about the roots to evaluate the polynomial efficiently. But estimating the roots of a polynomial is usually much more computationally intensive than evaluating the polynomial. Moreover, for polynomials of degree 4 and more there is no formula one could use in order to discover the roots. Still worse, there is even a mathematical proof that for polynomials of degree higher than 3 such formula does not exist, and so nobody could discover it in order to apply it for evaluating polynomials.

6. Conclusion

If one wants to strengthen students in algorithmic (computational) thinking, one should not train to correctly execute presented algorithms only. Instead, one has to push the students to use and develop abstractions and problem solving competencies in order to discover algorithms by their own. This requires developing robust strategies for designing algorithms for a large variety of problems. Teaching in high school asks for natural and well understandable strategies.

Constructive induction is the oldest robust strategy for solving problems and it is very transparent and comprehensible to students. But the key point is that constructive induction has a very high educational value with respect to the development of the student's way of thinking. Except the potential of introducing induction as a proof method, constructive induction offers a special case of recursion and so can be used as an easy introduction to the recursive algorithm design method "divide et impera" (divide and conquer). The induction step results in a recurrence that corresponds to the reduction of solving problem instances of size n to solving problem instances of size smaller than n, and so to the concept of "divide et impera." Also note that using constructive induction in problem solving consequently from smaller instance sizes to larger ones is also the base of "dynamic programming," another fundamental algorithm design technique.

In this paper we only outlined what kind of problems can be approached by constructive induction in high schools and showed how to involve students in searching for solutions in classrooms for some representative algorithmic problems. A careful implementation of teaching solving further problems by constructive induction is presented in our textbook Gallenbacher *et al.* (2023), which shows how to guide students such that they discover solutions and develop algorithms to a large extend on their own.

Acknowledgements

We are deeply indebted to Peter Widmayer who called our attention to the great potential of constructive induction for teaching algorithm design. We would like to thank Dennis Komm for his comments and improvements on the extended abstract of this paper.

References

Arnold, J., Donner, C., Hauser, U., Hauswirth, M., Hromkovic, J., Kohn, T., Komm, D., Maletinsky, D., Roth, N. (2019). *Programmieren und Robotik*. Klett und Balmer AG, Baar, Switzerland.

Bussey, W.H. (1917). The origin of mathematical induction. *The American Mathematical Monthly*, 24(5), 199–207.

Cypher, A. (1993). *Watch what I do: Programming by Demonstration*. MIT Press, Cambridge, MA, USA and London, UK.

Dagiene, V., Hromkovic, J., Lacher, R. (2021). Designing informatics curriculum for K-12 education: From Concepts to Implementations. *Informatics in Education*, 20 (3), 333–360.

du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Reserarch*, 2(1), 57–73.

Fincher, S., Jeuring, J., Miller, C.S., Donaldson, P., du Boulay, B., Hauswirth, M., Hellas, A., Hermans, F., Lewis, C., Mühling, A., Pearce, J.L., Petersen, A. (2020). Notional machines in computing education: the education of attention. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR 2020*. ACM, New York, NY, USA, pp. 21–50. 1-58113-828-8.

Gallenbacher, J., Hromkovic, J., Lacher, R., Komm, D., Pierhöfer, H. (2023). *Algorithmen und Künstliche Intelligenz*. Klett und Balmer AG, Baar, Switzerland.

Hromkovic, J., Lacher, R. (2023). How teaching informatics can contribute to improving education in general. *Bull. EATCS*, *139*.

Hromkovic, J., Lacher, R. (2024). Teaching Tangible Division Algorithms or Going from Concrete to Abstractions in Math Education by the Genetic Socratic Method. In: *Proceedings of the 7th International Conference, CMSC 2024*. LNCS 15229. Springer, Cham, Switzerland, pp. 136–144. 978-3-031-73257-7 (eBook).

Hromkovic, J., Kohn, T., Komm, D., Serafini, G. (2016). Combining the power of Python with the simplicity of Logo for a sustainable computer science education. In: *Proceedings of the 9th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives (ISSEP 2016)*. LNCS 11169. Springer, Cham, Switzerland, pp. 155–166.

Kohn, T. (2017). *Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment*. ETH Zürich, Zürich, Switzerland.

Kohn, T., Komm, D. (2018). Teaching programming and algorithmic complexity with tangible machines. In: *Proceedings of Informatics inSchools: Fundamentalsof Computer Science and SoftwareEngineering (ISSEP 2018)*. LNCS 11169. Springer, Cham, Switzerland, pp. 68–83.

Lieberman, H. (2001). *Your Wish is my Command*. Morgan Kaufmann, San Francisco, CA, USA.

Papert, S. (1980). *Mindstorms: Children, Computers and Powerful Ideas*. Basic Books, Inc., New York, NY, USA.

Tanton, J.S. (2021). *Mathematical Induction*. Encyclopedia of Mathematics, Online: `http://encyclopediaofmath.org/index.php?title=Mathematicalinduction`. Accessed September 20, 2024.

Vacca, G. (1909). Maurolycus, the first discoverer of the principle of mathematical induction. *Bulletin of the American Mathematical Society*, 16(2), 70–73.

J. Hromkovič has been professor of Information Technology and Eduation at the Department of Computer Science at ETH Zurich since January 2004 to January 2025. He is member of Academia Europea since 2010. His research and teaching interests focus on informatics education, algorithmics for hard problems, complexity theory with special emphasis on the relationship between determinsm, randomness, and nondeterminism. One of his main activities is writing textbooks which make complex recent discoveries and methods accessible for students and practitioners, and so contributing to the speed up of the transformation of new paradigmatic research results into educational folklore. In order to introduce the subject informatics to the school education, he founded the Centre for Computer Science Education in 2005. He is responsible for the master program Lehrdiplom Informatik at ETH devoted to the education of computer science teachers.

R. Lacher is with the ABZ (Center for Computer Science Education at ETH) and worked for more than 30 years in large organizations as quality manager or consultant on quality management. Regula has completed three educations: She is a geographer (master in natural sciences of the University of Zürich, 1990), a Quality Manager (accredited by the European Organization for Quality Management in 1993) and a physics laboratory technician (apprenticeship plus professional baccalaureate, in 1982). All these different backgrounds together with the interest in the concept of constructivism proved to be useful in her work in informatics education. Regula enjoys contributing to computer science education as a co-author to a series of textbooks, research papers and creates tasks for the Informatics Beaver Competition.