Logo-like Learning of Basic Concepts of Algorithms - Having Fun with Algorithms

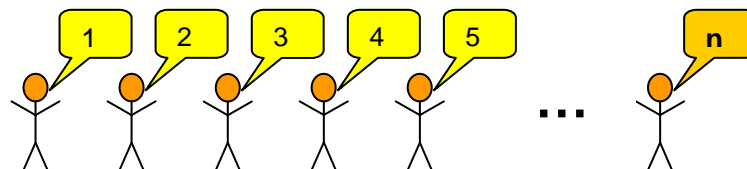**Gerald Futschek,** *futschek@ifs.tuwien.ac.at*
Institute of Software Technology and Interactive Systems, Vienna University of Technology

Abstract

The ability of algorithmic thinking is one of the major goals of informatics education. But learning algorithms is very often hard and boring for many students.

We show in this article that learning algorithms can be made easier and can be a lot of fun. The more the students are actively involved in the design and application of algorithms the more fun they have. We use constructivistic learning methods so the students learn how to design and detect algorithms without using a programming language.

Although we are not primarily interested in programming the way of learning is highly influenced by the Logo style of learning: creative learning by inventing algorithms and performing the algorithms themselves in smaller or even larger groups.

The students are counting themselves

The simply looking task "determine the number of students in a room" is used to demonstrate how all students of a course can be involved to learn how to design efficient algorithms.

Keywords
Logo-like learning, algorithms, group learning

Learning Algorithms

In schools and universities "algorithms" is a key topic in all informatics curricula. Although this topic is very essential and there exists much teaching experience, most of the students think that the algorithms stuff is very hard and boring. Usually there is a set of standard algorithms to be imparted to the students. Well elaborated algorithms that are sometimes very sophisticated are presented and analysed and should be understood by the students so that they are able to program them.

In this article we oppose this attitude and claim that learning algorithms can be a lot of fun even for students that have never done it before. The author gained experience in teaching algorithms in several courses for students that had no pre-knowledge in computer programming. He tried to transfer his experience in Logo education to teaching and learning algorithms.

The here presented algorithms are not intended to be programmed by the students. The students play the algorithms themselves. So they are part of the algorithms and can find easier and with more fun arguments for correctness and efficiency of the played algorithms.

Example: Determine the Number of Students

The students should determine their number. This task is easy to be understood and a first algorithm is easily found, but it is not so trivial if we want to perform this task fast also for a larger group of students. It can be used at the very beginning of an informatics course to demonstrate some basic properties of algorithms. Usually the following two solutions are proposed by the students:

1. **A counting person**: One student starts to count all the other students in a specific order. At the end the student reports the result after hopefully adding one for herself/himself.
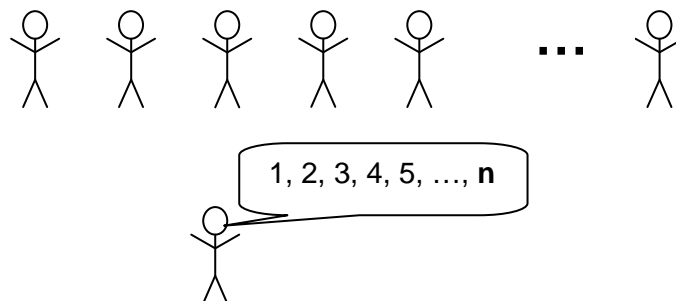

Figure 1. A student counts the other students

2. **Counting students**: Each student adds one to the previous student and tells the result to a student that has not yet counted. The last student, that has no next student, reports the result after adding one for herself/himself.
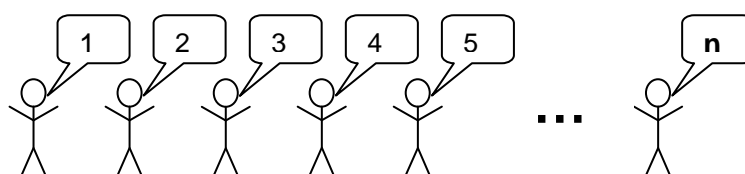

Figure 2. The students are counting themselves

These algorithms should be played by the students and the teacher may discuss the different approaches with her/his students: The first solution needs an algorithm for the counting person. The second solution needs an algorithm for each student! All students have the same algorithm except the first and the last one. The first student passes number one to the next student and the last student reports the total result of this commonly executed algorithm. In both cases the duration of the algorithm is proportional to the number of students.

A faster solution

Usually the students are satisfied with these solutions, so the teacher has to make progress proposing a challenging problem: Imagine there are some hundreds of students. Are the so far discussed solutions still practical to achieve an exact solution in a very short time? As we can observe most of the time the students are inactive and wait for their turn. There should be a possibility to do it faster with an algorithm where the students are more actively involved.

Often the students propose in this situation to count the number of students for all rows of an auditorium in parallel and then add the results of the rows. This is in fact much faster although only the students at the end of the rows have additional work to do. If we assume an auditorium of size i x j there are only i + j instead of i x j steps to do.
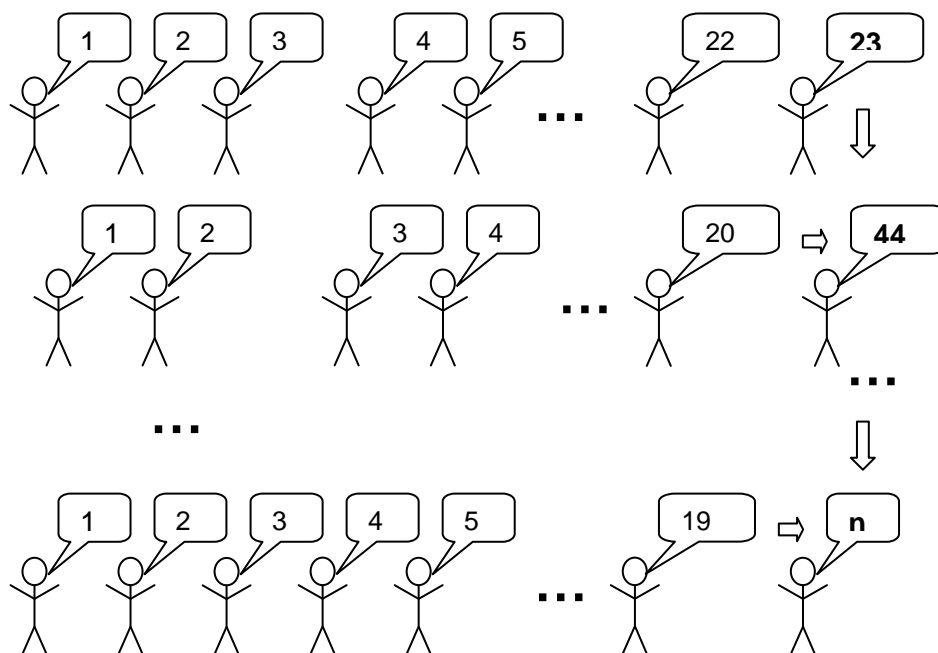

Figure 3. The students of different rows are counting independently and in parallel. The last student in a row has to add 1 (for himself) to the sum of the values from the previous student and the result from the previous rows.

The students at the end of the row have three things to do. First they have to wait until they can complete the number of students of their row. Secondly they have to wait for the number they receive from the student behind them. Thirdly they add these numbers and pass the result to the student in front. They can do these activities in this order or they may change the order of the first two actions.

Necessity of Synchronization

In practice this algorithm is not as easily performed as it looks like. Try to play this with your students! One can observe that the different rows have different speed in processing the numbers per row. So a student at the end of a row may get the result of his row before, after or

at the same time as the number of the student sitting behind him. Since he can understand and process only one of the two results at the same time, there is a need for synchronization in passing information to a student. In my experience a very good and natural way to synchronize two students is looking in the eyes of each other. Looking in each others eyes should be a prerequisite for passing a number.

Using this way of synchronization the algorithm of counting the number of students of a row can be formulated in a Logo-like pseudo-code. It depends on the sitting position of the student which one of these procedures he executes.

```
to beginOfRow
   look at next student in row and wait until he looks at you
      pass 1 to him
end

to countInRow
   look at previous student in row and wait until he looks at you
      receive number
   look at next student in row and wait until he looks at you
      pass (1 + number) to him
end
```

At the end of a row we have three different procedures depending on the sitting position of the student. In the last row there is no student in the back and in the first row the final result of the total number of all students is achieved.

```
to endOfLastRow
   look at previous student in row and wait until he looks at you
      receive number
   look at student in front of you and wait until he looks at you
      pass (1 + number) to him
end

to endOfRow
   look at previous student in row and wait until he looks at you
      receive number1
   look at student in back and wait until he looks at you
      receive number2
   look at student in front of you and wait until he looks at you
      pass (1 + number1 + number2) to him
end

to endOfFirstRow
   look at previous student in row and wait until he looks at you
      receive number1
   look at student in back and wait until he looks at you
      receive number2
   report (1 + number1 + number2)
end
```

The synchronization with another student and the action that is performed during this synchronization may be described also with the following subroutine, but due to better readability for students that are not familiar with programming we abstain from using this subroutine.

```
to synchronizeWith :student :action
   look at :student and wait until he looks at you
   run :action
end
```

Of course the whole algorithm can be adapted to other special arrangement of rows within a classroom or auditorium. In larger auditoria there are several rectangular blocks of rows that can calculate their numbers in parallel.

Although this version of the counting-algorithm is much faster than the first sequential algorithm, there is a huge potential for a still much faster algorithm. One can observe, as earlier mentioned, that most of the students are waiting for just two synchronizations and only once they had to add something. Only the students at the end of the row are waiting for synchronizations with 3 different students.

Counting with a "Divide & Conquer" Strategy

The students are now motivated to think about a still faster algorithm that is fast also in a very large group of people.

One idea can be that all students try repeatedly to synchronize with any other arbitrary students. When two students synchronize, one of the two students leaves the game (and sits down) the other adds the number of the leaving student to his number and starts again to synchronize with any other still active student.
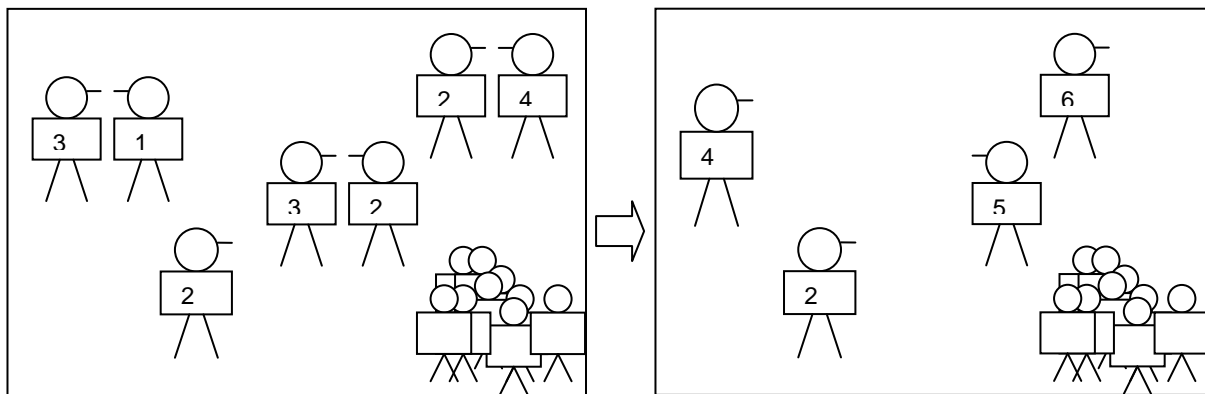

Figure 4.One step of the "Divide & Conquer" Algorithm: The students try to synchronize with a partner. One student of each pair gets inactive and sits down (lower right corner). The other adds the number of the now inactive partner to his own number.

At the beginning all students are standing and hold the number 1. At the end only one student is standing and he holds the total number of all students.

An analysis of the algorithm shows that the time effort can be log n, what is in fact much more efficient than the previously presented algorithms.

The strategy used is also called "Divide and Conquer". Each pair of students holds the numbers of two groups that represent a **division** of the students in two partitions. These two numbers are **conquered** (by adding) to yield the number of students of both groups.

Further Examples

Similar algorithmic tasks that involve all participants in the solving process are

- Distribute information material to all participants
- Determine the youngest student
- Determine the city, in which most of the students are born

In Futschek, G (2006) also a parallel version of the well-known BubbleSort algorithm is presented as a good example of a sorting algorithm that can be played by students, because the sum of the distances for walking is minimal. In general the advantages of efficient sorting

algorithms like QuickSort, HeapSort and MergeSort cannot easily be understood by playing them. They involve much more organizational work and also many exchanges of values over larger distances, that cannot efficiently be played by persons.

Good algorithmic tasks for pupils and beginners are usually not solvable in a trivial way but they must have an easy understandable problem statement. It is very important that the students invent and improve their algorithms themselves. They can do this alone or better in groups. Playing the algorithms is very important to clarify what is exactly intended by the various steps of the algorithms. Playing gives also an insight in the intermediate goals of an algorithm. This gives advantages in understanding correctness and efficiency considerations.

Summary

By detecting algorithms students can learn not only basic concepts of algorithms but also more advanced concepts like parallel programs, synchronization, divide & conquer strategy, etc.. Even complete novices are able to understand concepts of parallel algorithms, which are often easier to understand for them as sequential algorithms.

Challenging tasks with easily understandable problem statements are important to motivate students to find algorithmic solutions.

Playing algorithms is also a lot of fun for the students. Fun is motivation and motivated students have much better learning progress.

References

Futschek, G. (2006) *Algorithmic Thinking: The Key for Understanding Computer Science*, Lecture Notes in Computer Science 4226, Springer, pp. 159 - 168.