Iteratively Intervening with the “Most Difficult” Topics of an Algorithms and Complexity Course

EMMA ENSTRÖM, KTH Royal Institute of Technology
VIGGO KANN, KTH Royal Institute of Technology

When compared to earlier programming and data structure experiences that our students might have, the perspective changes on computers and programming when introducing theoretical computer science (TCS) into the picture. Underlying computational models need to be addressed, and mathematical tools employed, to understand the quality criteria of TCS. Focus shifts from doing to proving. During several years, we have tried to make this perspective transition smoother for the students of a third year, mandatory algorithms, data structures and computational complexity course. The concepts receiving extra attention in this work are NP-completeness, one of the most central concepts in computer science, and dynamic programming, an algorithm construction method that is powerful but somewhat unintuitive for some students.

The major difficulties we attribute to NP-completeness is that the tasks look similar, but have a different purpose, than in algorithm construction exercises. Students do not immediately see the usefulness of the concept, and hence motivation could be one issue. We have partly studied the teaching of NP-completeness in collaboration with a professor giving a similar course at a university in another country. One line of attacking NP-completeness has been to emphasize its algorithmic aspects, using typical tools for teaching algorithms.

Potential difficulties associated with dynamic programming is that it is based on a known difficult concept – recursion – and that there are many ingredients in a dynamic programming solution to a problem.

For both dynamic programming and NP-completeness, we have invented several new activities and structured the teaching differently, forcing students to think and adopt a standpoint, and practice the concepts in programming exercises.

For students, one particularly difficult area is the proving of correctness required in all tasks in the course, and one area of uncertainty for the students is presentation of algorithms in pseudo code (which lacks formal syntax). These issues have also been addressed.

The approach to improving the course is action research, and the evaluation has been done using course surveys, self-efficacy surveys, rubrics-like grading protocols, and grades. We have also interviewed teaching assistants about their experiences.

Categories and Subject Descriptors: F.1.3 [**Computation by Abstract Devices**]: Complexity Measures and Classes—*Reducibility and completeness*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms: Algorithms, experimentation, theory

Additional Key Words and Phrases: Algorithm visualization, assessment, NP-completeness, self-efficacy

ACM Reference Format:
Emma Enström and Viggo Kann, 2014. Iteratively Intervening with the “Most Difficult” Topics of an Algorithms and Complexity Course. *ACM Trans. Comput. Educ.* ?, ?, Article ? (October 2014), 36 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

This work is funded by RCN at KTH School of Education and Communication in Engineering Science and GRU at the KTH School of Computer Science and Communication, partly from the CPU fund.
Author’s addresses: Department of Theoretical Computer Science, School of Computer Science and Communication, KTH Royal Institute of Technology, Sweden.

1. INTRODUCTION

The ACM Computing Curricula [ACM 2013] stipulates which "bodies of knowledge" should be included in an undergraduate degree program in computer science. The first of these is *Algorithms and Complexity*, and the main part of it is usually taught in a separate course containing subjects like algorithmic strategies and computational complexity. It is obvious that algorithm design is an important skill for computer scientists, but what about computational complexity?

In his 1997 invited talk at a theoretical computer science conference Christos Papadimitriou [Papadimitriou 1997] observed how the notion of NP-completeness has become a pervasive and influential concept in many diverse disciplines, ranging from "statistics and artificial life to automatic control and nuclear engineering", and gave some reasons for this success. One of these is that NP-completeness is a "valuable intermediary between the abstraction of computational models and the reality of computational models." As a consequence, it is now a widely accepted fact that NP-completeness is a fundamental concept that "any CS professional should understand and be able to apply" [Lobo and Baliga 2006].

On the other hand, it is also quite well-known that computational complexity is not easy to teach or to learn and, in general, the problem of presenting theoretical foundations of computer science in an integrated and motivating way has been studied for decades (e.g., [Mandrioli 1982]) and is still a rich, though somewhat under-explored, research area for computer science education (e.g., [Goldreich 2006]).

The main algorithmic strategies that should be taught in an Algorithms and Complexity course, according to the ACM Computing Curricula 2013, are brute force algorithms and recursive backtracking (i.e. exhaustive search algorithms), greedy algorithms, divide-and-conquer, and dynamic programming. Of these strategies, dynamic programming (dynprog in the following) is perhaps the least straight-forward one. The bottom-up approach of dynprog is a strategy for "reversing" the evaluation order of sub problems and turning recursion into iteration. There is also a top-down approach to dynamic programming, which is called *memoization* in which the recursive calls are complemented with a lookup table where already calculated values are stored, effectively reducing the number of recursive calls that are made. The problem is still solved recursively, though. This version does not appear to confuse as many students, but it is lacking one of the ingenious features of dynprog. Memoization algorithms are recursive and recursive algorithms are not as easily analyzed as iterative algorithms. For an introduction to dynamic programming, see for example [Kleinberg and Tardos 2006].

This article is about improving learning of the concepts of NP-completeness and dynprog in an algorithms and complexity course for third-year undergraduate computer science students at KTH. We have chosen to concentrate on these concepts because they present difficulties of different kinds for students, according to our experience.

In the course, both concepts are assessed with problems where the student is facing the task of devising some algorithm, present it nicely (generally in pseudo code), analyze its complexity, and prove that it works, i.e. solves the problem. All steps apart from actually designing the algorithm are, from the students' perspective, absent when programming – after constructing a program that solves a task, nothing remains. This might be the reason so many students are either not performing well on, or complaining about, these steps in the solution. The students expect something different from the task and their solution, than does the teacher. These circumstances are also addressed in the last cycle of this project.

The task "write a program that solves this problem:. . . " does not seem to pose any difficulties to our students apart from how easy or difficult it is to come up with an idea of how to solve the problem, in a programming setting, provided that all you have to do is write the program.

In a theoretical computer science setting, implementation of algorithms is less important than *design* of algorithms. The task is phrased “design an algorithm that solves this problem:. . . ”, and the student is expected to describe the algorithm (preferably both in pseudo code and with some accompanying explanation in natural language), analyze its complexity, and prove that it actually solves (all legal instances of) the problem. “Look, I have attached the code that implements this algorithm, and it works” is, by the standards of theoretical computer science, not a proof. (On the contrary, this actually introduces new possible errors due to the implementation itself, the language chosen, the computer architecture, and so on, and it is *harder* to prove that the implementation is correct.)

We have seen many students being unprepared for arguing correctness when presenting their homework solutions, despite the fact that this is often explicitly mentioned as a requirement. Nevertheless, students do present solutions to the problem, let it be with methods from another context than the course, and the task “solve this problem” is self-evident and does not need any particular presentation or persuasion to tackle.

The results described in this article are partly based on four previous papers [Enström and Kann 2010; Enström et al. 2011; Crescenzi et al. 2013; Enström 2013] : three about the course improvements, and one about automated assessment. Additional results regarding students self-efficacy beliefs concerning proof and pseudo code, additional data supporting or contradicting previous findings, and extended discussions on the communicative characteristics of the teaching and learning situation, are also included.

1.1. Potential NP-completeness related difficulties

When, on the other hand, the task is “Prove that the problem X is NP-complete”, in addition to the problems students are facing in algorithm tasks, they also need to know the particular purposes of NP-completeness reductions. These reductions work just as algorithmic reductions in general, but we are concerned with which properties of the original problem are “preserved” through the reduction. In addition, we also use the reduction in a proof by contradiction setting, which means that we start with a known problem and reduce it to a new problem. This is certainly *not* how we solve problems in general! These are potential difficulties with the NP-completeness reduction tasks, but another possible obstacle is motivation. Unlike the task of solving a problem, the task of proving a problem to belong in a particular problem class and the usefulness of the result of those efforts, is not self-evident to our students. This either causes students to lose motivation, or to try solving the problem instead, as this type of task seems more useful, natural or recognizable than “only” showing that the problem is difficult. There are grains of truth in this view – certainly a rich field of TCS is concerned with solving the difficult problems as well as possible. Lacking good ways of approximating or heuristically solving the problems, TCS goes back to classifying the difficult problems again, this time according to how well they can be approximated. We suspect that we need to motivate the usefulness of classifying problems according to computational hardness to the students.

Motivation of the usefulness of a subject is important for learning. For example, Light et al. [Light et al. 2009] write:

> In professional courses, the match between a student’s understanding of what it means to be an engineer or a doctor and what the course seems to be providing can be crucial both for motivation and intellectual development.

So if the students have trouble seeing the usefulness of computational complexity they might have less incentive to learn. However, as already observed, NP-completeness and computational complexity) has had a huge impact in many different application areas, and it should be possible to motivate the concrete usefulness of this concept to students.

Besides possible failures to expose the rich applications of reductions, one missing ingredient while teaching these concepts is that the concepts themselves are just another way of usefully applying the algorithmic way of thinking, which is usually taught to all computer science students. In other words, a further motivation to learn NP-completeness is that designing reductions can be, in a certain sense, exactly the same as designing algorithms: a student enjoying this latter activity should also enjoy the former one.

The reduction is supposed to be used in a proof, and the motivation for doing so is the same for all NP-completeness proofs. Many students do not see the purpose of designing a reduction in this context and end up constructing exhaustive search algorithms while working on their proofs, or produce a "proof" with a reduction in the wrong direction, from the new problem to the known one. This is in line with previous research on reductions, which has found that when confronted with the task of constructing an algorithm based on a reduction, students tend to try reducing abstraction by "opening up the black box" [Armoni 2008; Armoni et al. 2006], which leads to algorithms for solving the original problem instead of reducing it. In this case, it happens already in the context of "normal" problem solving through reductions. We believe that this tendency can be more prevalent in our setting, when the purpose and context of the reduction algorithm is less familiar to the students.

Finally, if the hardness is due to lack of experience with proofs, this is a larger issue that possibly needs a cross-curricular approach and is better attacked by special courses like the ones described by [Muller and Rubinstein 2011]. Also the attempt to make reductions "a habit of mind" [Armoni et al. 2006] might fit into that approach.

We decided to make use of two typical tools for teaching the design and the analysis of computer algorithms; an automated program assessment system (Kattis) and an algorithm visualization system (AlViE[1]). It is worth observing that even though NP-completeness is one of the concepts students typically struggle with, as far as we know this concept is not well covered by educational software: the only other experiences we are aware of are the ones described in [Brändle 2006; Pape 1998]. We also supported the use of the two tools with other activities mainly devoted to highlighting the usefulness of learning computational complexity and to forcing students to think and to adopt a standpoint. This paper describes the activities that we introduced in our courses and how the students responded to them.

1.2. Potential dynamic programming difficulties

As for dynamic programming, one thing that could contribute to making it difficult is that recursion, a well known "difficult task", is part of the problem solving strategy of dynprog. The subject of recursion has attracted educational researchers' interest for a long time. In the 1980s Kahney and Eisenstadt [Kahney and Eisenstadt 1982] contributed by studying students' and other programmers' answers to problems involving recursion. Their results are further described by Kahney in [Kahney 1983], and describe a set of "mental models" of recursion that students had, out of which one was capable of capturing the things instructors want students to know about recursion, and some were not only incomplete, but misleading. Around the same time, Ford [Ford 1982] concludes that iteration is really a special case of recursion, and that recursion is

[1]http://alvie.algoritmica.org/

a generalized control structure in programs. Similar arguments are also used by others: recursion is an example of the paradigm “Divide, Conquer and Glue”, and when using iteration, the glue step is missing and students get erroneous pre-images of the paradigm [Turbak et al. 1999]. Scholtz et al. [Scholtz and Sanders 2010] claim that the difficulties with recursion are connected to understanding the passive flow, whereas other authors are investigating misunderstandings that can occur around base cases [Haberman and Averbuch 2002]. They also note that recursion *is* a difficult topic for students. Many authors dwell on the topic of where in the first course recursion should appear, or how recursion is related to iteration, the computational model, or similar issues.

Beyond the possible difficulties associated with recursion, previous years' experiences suggest that two difficult parts of dynprog are finding a recurrence relation based on some structure of the desired solution, and tackling the complexity of solving a problem completely from scratch, without hints; remembering to perform all steps that need to be taken when constructing and analyzing the algorithm. Many students also struggle with, or avoid, any tasks involving “argue correctness” or “prove”. Being able to cope with many dimensions – problems where a simple two-dimensional matrix is not sufficient to hold every subproblem that matters to the solution, is necessary for some problems.

Ginat et al [Ginat and Shifroni 1999] suggest that less focus on the computational model, and more focus on the abstract level and the algorithm in theory can help students not to mistrust recursion as a method or their own abilities on recursion. As an algorithm construction method, dynprog follows that recommendation. It does not deal with passive flow, and the algorithm construction task indeed treats recursion as an abstract phenomenon.

1.3. On proofs

Proofs have been known to be a troublesome area of mathematics for students for decades. Balacheff [Balacheff 1988] devised a taxonomy for various approaches students can have to proof, out of which the two first levels are only more or less random testing with examples. In order to make this taxonomy more useful for teachers, Varghese [Varghese 2011] made complementary notes and constructed examples of the taxonomy.

In a study on Taiwanese undergraduates, Ko and Knuth [Ko and Knuth 2009] classify their subjects' answers as *no response*, *restatement*, *(invalid) counterexample*, *empirical*, *non-referential symbolic*, *structural* and *completeness*, and find depressingly many answers ending up in the two first categories. For counterexamples, the results look somewhat better. They also raise the question whether the teaching assistants understand proofs well enough. Jones [Jones 2000] also lists several previous papers where students are found to perform unsatisfactory when producing proofs, and investigate how prospective UK teachers, who have finished undergraduate mathematics degrees, perceive proofs. Jones finds that there are many students who have good grades in mathematics, but little explicit, reflected knowledge *about* proofs. They are not good at describing them in a rich and detailed way. Technical excellence is not always followed by a rich conceptual understanding.

Stylianides and Al-Murani [Stylianides and Al-Murani 2010] were looking for a particular misconception about proofs: that a proof and a counterexample can coexist, for the same assertion, among secondary students in the UK. When surveying, it seemed like students could have this misconception, while when interviewing, there was not evidence of anyone having it. The hypothesized misconception stemmed from combining results by Balacheff [Balacheff 1988] and Fischbein [Fischbein 1982].

Similar results, suggesting that students did not understand the role of proof, were in 1993 obtained by Chazan [Chazan 1993]: "there could always be a counterexample".

In a paper from 2011, Zerr and Zerr [Zerr and Zerr 2011] describe how the students of their study are far better at correctly classifying correct proofs as correct, than incorrect ones as incorrect (96 % success rate vs. 62 % when only some errors had to be spotted, and 35 % if we require students to find all errors). They argue that this is likely if all practice students have regarding proofs is to understand those in the text book and those shown by the teacher. In the study, peer-assessment of proofs and grading of both initial proof, assessment, and revised proof are performed.

2. THE COURSE

Algorithms, Data structures and Complexity (ADC) is a compulsory, third-year course in the 5-year Computer Science and Engineering program at KTH, Stockholm. The prerequisites for this course are *programming* (CS1), *algorithms and data structures* (CS2), *computer architecture*, *discrete mathematics*, *probability theory*, and *logic*. About 150 students follow the course every year.

ADC consists of 32 lectures (the first three are 90 minutes, and the rest 45 minutes each), given by the second author, 12 two hour tutorials in three groups, given by PhD and master students, and 4 compulsory computer labs. The assessment consists of two graded homework assignments and a written theory exam only assessing the lower grades assessment criteria. There is also an optional oral exam for students aspiring to get the highest grades. The mandatory computer lab exercises are performed in pairs, automatically checked for correctness and verbally presented to a teaching assistant (TA) in lab, and only graded pass/fail. ADC uses continuous assessment.

The first 19 lectures, 7 tutorials and 3 computer labs cover construction and analysis of algorithms and data structures. Out of these, lectures 9 and 10 and tutorials 3 and 4 plus one computer lab exercise deal with dynprog. The lab exercise, together with an individual, written home assignment which the students afterwards discuss with teacher or TA, constitute the major part of the assessed student work on the topic. There are also supplementary tasks for those who are unhappy with their performance and want to improve their grades.

The following 12 lectures cover reductions (1), introduction to complexity (1), Turing machines and undecidability (2), Cook-Levin theorem (1), NP-reductions and NP-completeness (3), approximation algorithms and heuristics (3), other complexity classes (1). The tutorials in parallel cover reductions and undecidability (2), NP-reductions and NP-completeness (4), approximation algorithms (2), solution to the complexity homework (1), and complexity classes (1), all of which are relevant NP-completeness.

The course is graded on a scale from A to E, or F for fail, and there are intended learning outcomes and assessment criteria connected with each grade. These are presented in a matrix and a flow chart on the course home page, and addressed on lectures and during peer review. In the end, how well your work meets the criteria decides your grade, so *what* work you have fulfilled, and how well you performed, matters – not just the percentage of total work.

The second author was the teacher responsible for the course and the first author has been one of the TAs.

2.1. The homework assignments

The individual homework assignments are very important as both formative and summative assessment in the course. They also played an important rôle for the studies described in this paper. Therefore we will explain how they work in more detail.

The first homework assignment assesses the learning outcome on algorithm design and the second assesses the learning outcome on complexity. Each homework assignment consists of three tasks assessing the grading criteria for the grades E, C and A, respectively. A student passing only the fist task has shown knowledge on the E level. A student passing two tasks has shown knowledge on the C level, and a student passing all three tasks has shown knowledge on the A level.

Students have two weeks to work on each homework assignment. No cooperation at all is allowed. In fact, the students are not allowed to discuss the assignments with anyone except the teachers of the course. Each student should hand in solutions as a written report. A few days after the due date of the written reports, each student has a 15 minute oral presentation with a TA. During the oral presentation minor errors can be fixed, and the TA can explain misconceptions. It is the combined written and oral presentation that is assessed. The feedback (including the grade) to the students is given orally and immediately.

Before the oral presentations the teaching assistants are instructed how to assess the assignments, and they are given some time to read the written reports in advance.

When all students have presented their solutions the teacher publishes correct solutions on the course web. In order to give more feedback, 45 minutes of the next tutorial are spent on explaining the solutions.

3. METHOD

We tried an action research approach to our course improvement work, each cycle consisting of the following steps:

(1) Identify which are the hardest things (concepts, methods, skills etc) for the students to grasp.
(2) Try to change teaching and/or assessment in order to improve the learning of the identified problem areas. Adding teaching and learning activities addressing these potential hardest things.
(3) Evaluate the results. Find new indications of “hard” content.

For each cycle, we also included new evaluation points, and additional topics to be considered: first NP-completeness, then dynprog and lastly proofs and pseudo code. These roughly correspond to first, second and third cycles, but the NP-completeness work started earlier [Enström and Kann 2010] and had several iterations before the “real” first cycle began. During the second iteration of complexity theory, we collaborated with a professor giving a similar course at a university in Italy [Crescenzi et al. 2013]. Together, we discussed which things we believed to be hardest for students to learn, and our previous experiences. We also discussed how we at our different universities had tried to mitigate these difficulties, and tried each other’s methods. We found this type of course development activity very fruitful. It had no permanent funding, so it has not been a long-lasting part of planning the course. One prominent source of feedback into the action research loop has been the somewhat ill defined “What students complain to their teachers about being extra difficult or not practiced enough.” Assuming that this is not due to observation bias, the fact that students usually complained about complexity, and then one year mainly complained about dynamic programming, could mean one out of two things. Either that dynprog was looking like a more challenging subject than complexity that year, or possibly that complexity was already recognized by the teachers as hard. However, the changes in teaching have not been aimed at describing complexity as hard, but rather at showing that it is a different topic, with similar methods, used for different purposes, compared to algorithms.

4. ACTIVITIES IMPROVING THE COURSE

Throughout the cycles, the silver thread through our plans has been to explicitly address aspects of the topics that we suspect students are not seeing. This has been done both by modifying existing activities, and by adding new activities to the course.

4.1. Programming assignments

When the course was given for the first time, in 1999, a decision was taken to include mandatory programming lab assignments in the otherwise theoretical course. The reason for this decision was that we suspected that the students would learn the theory better by practicing it in a familiar way, and thereby easier seeing that the theory is useful, which should increase their motivation.

Originally there were three lab assignments: implementing some data structures on disk, optimizing a given algorithm, and designing an algorithm for bipartite matching in a graph using a max-flow algorithm, that also should be implemented. Since 2006 the automated programming assessment system Kattis[2] has been used in ADC. Kattis was developed at KTH 2005 to assess programming exercises. It has later been commercialized. The typical way of using this system in teaching is described in [Enström et al. 2011].

During the process of solving a task, students have access to Kattis 24/7. Source code is submitted to Kattis, and the system runs secret test cases and interprets the output of the submitted code, and then reports the results to the students via email and/or a web interface.

4.1.1. The reduction computer lab. The first change to the course was to add a programming exercise on NP-completeness reductions. This was in order to tie that part of the course closer to the other parts, and to emphasize the algorithmic side of reductions. It was also supposed to prevent people from making their first "reduce in the wrong direction" mistake on their individual homework assignment, where it affects the grade more severely.

The specific exercise that the students were presented with was a reduction task, where they could choose between two NP-complete problems, and then produce code that reduces input for that problem to input for a new problem that was described in their instructions.

The feedback from Kattis for this problem consists of information on whether the solution was working or not, and in case it was not, whether a "yes" instance had been transformed to a "no" instance or vice versa and occasionally other hints on what could have happened. This is a slight change in the feedback compared to [Enström and Kann 2010], where this particular exercise is described further.

As a contextual note, for an ADC Kattis programming assignment, the students are given the text of the laboratory exercise in advance. The text also contains some theory questions that the students are asked to answer before solving the exercise. After the first 10–12 hours of lectures on this part of the course, the students are asked to perform a 15 minutes peer review of the answers given to the theory questions, and during the next week, the students work with the actual programming both on scheduled lab hours with teachers available, and on their own time. Finally, the students also have to discuss their solution with a TA during the scheduled computer lab hours.

4.1.2. The dynprog programming assignment. After adding a mandatory programming assignment to the course, right before the individual homework on the same topic, students were asking to be able to practice in the same way for their dynamic program-

[2]https://kth.kattis.com

ming algorithm design. The exercise where students optimized an existing program was replaced with an edit distance assignment utilizing dynamic programming, but with the same focus as the previous exercise: practicing code maintenance and rewriting other people's code. Students get a Java program where recursion is utilized directly and this in combination with some other less clever implementation details guarantees that the Kattis time limit will never be met – but the program works. The task is to speed it up significantly, which is mainly done by using dynprog.

4.2. Clickers and response cards during lectures

The use of clickers (or Audience Response Systems, or "voting" systems) for pedagogical or administrative purposes is widespread in the world today [Caldwell 2007]. Among the reasons for using them is "anonymity", but since the systems are often built in a way that admits excessive data collection and even scoring on an individual level, it is worth noting that this anonymity is only towards peers, not towards the teacher. In other words, if the student is afraid of speaking up and answering questions on lectures because it could expose him or her to *the other students*, the argument is valid. If students don't want to expose possible lacks of knowledge to teachers, the system can actually be a greater threat, as temporary misunderstandings might be recorded and later graded.

The advantages with clicker questions are well-known, especially for teaching physics [Duncan 2006]. For example, they provide means for student activity; everyone gets to think and answer the questions, not just the fastest responder; the results of the small polls allows the teacher to address misunderstandings directly; and they provide formative feedback to the students. They also incentivize the teacher to plan good questions that are suitable for this type of exercise. The use of response cards, predecessors of clickers of sorts, has been studied academically since the 1960s, and have proven to have effect on student grades and student participation [Randolph 2007].

At KTH, there was no clicker system, so we constructed our own low-budget system of response cards of three colors. We have gradually included response cards in the course during the two past years, and it has been much appreciated by the students. The cards are administered to the students in the beginning of lectures. The votes are still made simultaneously, and the feeling of being exposed seems not to be present among our students, according to their evaluation responses.

The first year, the clicker questions were introduced in an algorithm lecture before the complexity lectures. At the end of the lecture, we asked (using the cards) whether we should continue to ask such questions. Everyone answered yes! We then used clicker questions in most complexity lectures to reveal and remedy misconceptions on undecidability and reductions. Some questions required discussion among the students and some questions required fast response. The second year clicker questions using response cards were used throughout the whole course.

4.3. Visualizations

Visualizations are commonplace when teaching algorithms, and there is a multitude of tools and applets available for teachers for visualization purposes. For our course, no tools were originally suggested, and only a few algorithms were traced on lectures. Our Italian colleague had developed his own visualization tool AlViE, and during the exchange part of the project, we borrowed each others' tools and exercises. From that year on, computer aided visualizations were present in the course. We also linked to other algorithm visualizations on the internet from the course web pages.

For the dynamic programming methods, the Fibonacci sequence was visualized with both recursive calls, memoization and (bottom-up) dynamic programming. Some more complicated dynprog algorithms were also illustrated.

Why do you need to learn computational complexity?

1. Prerequisite to advanced courses (e.g. cryptography).
2. Will help you to attack hard real problems as an engineer.
3. Makes you more attractive to several companies.

Fig. 1. Arguments for learning complexity.

4.4. Visualizations of reductions

As stated in [Crescenzi 2010], a reduction is, for all purposes, an algorithm transforming instances of a starting problem into instances of a target problem: it is then natural to use algorithm visualization techniques while teaching reductions. We used the AlViE system to present the visualizations of two reductions in the following way.

Two of the three reductions from 3-satisfiability to Subset sum, 3-colorability and Vertex cover (inspired by [Kleinberg and Tardos 2006; Sipser 2012]) were selected and presented theoretically in a lecture. During the reduction explanation, the teacher made use of the visualizations, which were successively made available on the AlViE web site, so that the students could experiment themselves. The two visualizations not only show how the starting instance x is transformed into the target instance x', but they also show how a solution of x' can be transformed into a solution of x, thereby visualizing the proof of the correctness of the reduction.

4.5. Motivation and usefulness

As mentioned above, for algorithm design exercises students appear to be motivated by the task itself – to come up with a clever algorithm. For complexity, the situation is a bit different. The main motivational problem could be that the students do not think that they will benefit from learning complexity, outside of the course. In order to get proof of the industrial usefulness of computational complexity, we sent the following questions to some former students now working in industry:

(1) Describe a case where knowledge of computational complexity has helped you in your work.
(2) Do you regard algorithmic knowledge and knowledge of complexity as a qualification when recruiting computer scientists?

We got several positive answers, leading to a 5 minute motivational part of the lecture where complexity was introduced in ADC. Three arguments were presented to the students, see Figure 1.

The following example of attacking an NP-hard real problem was given:

> At Racasse we developed the price comparison service RedElvis. It was to find the cheapest way to order books, music and video on the web, considering delivery terms, discounts etc. We showed that the complexity of the problem is too high. Therefore we implemented some heuristics instead of an optimal algorithm, and found that simulated annealing gave solutions which in every test were equal to or better than the best solution a human could obtain [Grundin 2011].

Two employability (third argument) examples were given:

(1) Gustav Grundin at Vodaphone always gives an algorithmic/complexity problem at the employment interview.
(2) Lars Engelbretsen at Google: “Good knowledge of algorithmic problem solving is necessary for developers at Google.”

A, B, C, D and E are decision problems. Suppose that B is NP-complete and that there are polynomial-time Karp reductions between the problems in the following way:

$$\begin{tikzcd}
E & A \arrow[l] \arrow[r] & B \arrow[r] \arrow[d, leftrightarrow] & C \\
 & & D &
\end{tikzcd}$$

What will we know about the complexity of A, C, D, E? Mark with a cross each square that corresponds to something we know for sure.

	in NP	NP-complete	NP-hard
A			
C			
D			
E			

Fig. 2. The NP-reducibility assignment (ADC).

4.6. The NP-reducibility assignment

Text book assignments on complexity are often about solving a problem or proving something. There might be a lack of simpler assignments. We have introduced a new type of complexity assignment, which we have used both in the instruction and assessment, see Figure 2. The assignment hides all details about the involved problems and makes it impossible to tamper with the contents of the black box. Just asking the students this type of question, that they normally are not asked, directs their attention to the fact that this is important in itself. When we introduced the question, it was immediately appreciated by both students and teaching assistants.

4.7. Pattern Oriented Instruction

The second cycle of improvements also experimented with a new structure of the course contents, regarding dynamic programming.

Since coping with many aspects of a problem at the same time might be difficult, and since teaching everything at once might lead students to focus entirely on the algorithm that is constructed, and less on the construction process, we tried using Pattern Oriented Instruction (POI). POI is based on cognitive psychology theories on construction and organization of knowledge in schemata. A schema is a mental, connected chunk of “information”, that has been constructed by repeated experiences which share some common ingredients.

When, for instance, solving a complex problem, we are processing several different types of information at the same time. This is called cognitive load, and if the cognitive load gets too heavy, our problem solving skills are considerably reduced. When having seen the same type of problem many times before, we are able to process many interconnected pieces of information as one unit, a “chunk”, which reduces cognitive load. A pattern is either distilled from several different experiences with certain ingredients in common, or generalized from some specialized example of a phenomenon. It can be employed in teaching to enhance the learners’ ability to create new schemata. POI deals with structuring the teaching and the content in a way that facilitates the creation of schemata, which can later be processed as chunks.

POI preparations focus on identification, selection, progression and comparison of problems and patterns. The students are exposed to increasingly complex problems, encompassing different patterns already familiar. The comparison of problem characteristics is central, and patters are first introduced and later revisited in a different setting. The effect of POI on students' abilities in problem solving, abstraction and analogical reasoning in particular, are investigated by Muller and Haberman [Muller 2005; Muller and Haberman 2008; Haberman and Muller 2008] and the effect on problem decomposition and solution construction abilities by Muller, Haberman and Ginat in [Muller et al. 2007].

4.7.1. New structure of the course content . Together with the POI approach, we used Variation theory, based on the writings of Marton and Booth [Marton and Booth 1997]. This is a theory involving the prerequisites of learning: in order to learn something, we have to both acknowledge it as something important, and see that it could have been arranged in another way than it currently is. Everything that can vary, must be perceived to have the option of varying, for the learner to be able to discern that as a significant dimension of variation. When used for planning lectures – if the students notice – this can help to mitigate misconceptions about the topics learned. For instance, that the equal sign has the meaning of equality instead of only being a "do something now" operator, cannot be learned unless the equal sign sometimes is not used as that operator.

The description below is also given in [Enström 2013].

The task of solving a dynprog problem can be thought of as solving three different tasks:

(1) finding a structure for the solution,
(2) expressing a recurrence relation, and
(3) defining and proving an evaluation order with the properties that we will always have solved "smaller" subproblems whenever we solve a "larger" problem, and that the evaluation order renders the same result as the recurrence relation would produce.

To help comparing and distinguishing various characteristics of problems, we identified some features that could vary between different dynprog problems, and we hence needed examples of: whether the history needed to be saved during computations or not; whether the evaluation order was intuitive or more intricate; whether new values depended on the indices of the elements to be calculated, or on some input, or both; whether the previous subproblems to be used for each calculation were the most recently calculated subproblems, or if some special method was needed to find relevant subproblems (for instance, "jump", skip cells, in a matrix of previously calculated values); the number of dimensions for the subproblems (do they fit in a sequence, matrix, higher order); that there could be several different recurrence relations for the same problem and that the same recurrence relation could be the basis of several differently posed questions; the "location" of the base cases, "the location" of the answer after calculation; whether only a number, or the entire path to that number was needed for the solution (constructive solution), and different combinations and variations of these features.

These were only the dynprog specific variations. Another important variation is in the structure of a problem, or rather, of its solutions. If subproblems do not overlap, divide-and-conquer is generally a better method than dynprog, and if they do overlap, dynprog might be best. A greedy solution can sometimes be proven to be the superior option. These algorithm construction methods are also covered in the first part of the course, and assessed on the same written home assignment.

4.7.2. Phases, prototypes and patterns in dynamic programming . For the lectures, we decided to separate only two phases of constructing a dynprog algorithm:

— recognizing the structure of the problem and create a recurrence relation and
— finding and proving an evaluation order for a given recurrence relation,

and to deal with one phase at a time in our teaching.

Since we believed that the first phase was the hardest to learn, the course started with the second phase and let the students practice it in a new lab assignment, see 4.1.2. Then we proceeded to the first phase. This division into phases is also helpful when arguing correctness for the algorithm.

In the first phase we wanted to refer to prototype problems. Some of these had already been used in the course, and others were introduced as a complement to these. The prototype problems we chose were: Fibonacci (sequences), 2-dimensional recurrences without input (2-dimensional sequences), 2-dimensional recurrences with input, (values decided from input), Longest increasing subsequence (index and input dependent, need to save full history) Matrix chain multiplication (base cases on the diagonal, construction of the solution), Swamp walk (different questions, same “solution”), Coins (different recursions for the same problem, focus on proving correctness), Longest common substring (argue correctness of recurrence relation and algorithm, construct the solution, compare to 2D-with input that rendered the same recurrence relation) and Floyd-Warshall’s algorithm for finding all shortest paths in a graph (more than two dimensions used). We do not argue that these problems are *the best possible* set of prototype problems, but they contained examples of the variations we wanted to show and most of them had appeared in one form or another on the lectures previous years, only not as explicit representatives of some technique(s) each.

5. EVALUATION

We have evaluated the new activities in several ways, using course surveys, self-efficacy surveys, rubrics-like grading protocols, and grades.

5.1. Self-efficacy

In contrast with self esteem or confidence, self-efficacy is described as an individual’s confidence in his or her own ability to, at a given moment, perform actions in order to achieve some desired outcome. The term was introduced by Albert Bandura in the 1970s and is further developed by him in [Bandura 1986]. The phrasing of the self-efficacy items should be direct and not involve guesses about the future or some sort of inherent capabilities of the subject to the study. Self-efficacy beliefs are not static – on the contrary, they change with the individual’s experience. It is self confidence of a sort, but situated and very localized in time and subject area contents. These characteristics have given self-efficacy a role in education. The score of a self-efficacy test is known to be an important predictor of success [Pajares and Miller 1994]. There are studies in how self-efficacy correlates with performance [Iannone and Inglis 2010], how self-efficacy changes during studies[Ramalingan and Wiedenbeck 1998], and how it correlates with other factors around the individual [Askar and Davenport 2009].

We knew of no established instruments measuring self-efficacy for theoretic computer science. In mathematics, self-efficacy instruments have been developed [Iannone and Inglis 2010; Pajares and Miller 1994] and also in programming [Ramalingan and Wiedenbeck 1998; Askar and Davenport 2009], which is another related area.

Self-efficacy can either be considered as a learning outcome in itself (the purpose of some teaching could be to make students confident that they could perform certain tasks with desired outcome), or as an indicator of how well students are actually performing. The latter would require us to correlate self-efficacy to performance for our items and (grading) quality criteria. Self-efficacy like items can also be used to compare various tasks of the course and try to establish whether students are more

DP1. I could understand dynprog algorithms presented to me by a teacher or in a book.
DP2. I could recognize that an algorithm uses the dynprog strategy.
DP3. I could design a dynprog algorithm computing a recursively defined number sequence.
DP4. I could explain to an average CS student why dynprog is better than recursion when computing the Fibonacci sequence.
DP5. I could choose a dynprog order of computing for values depending on separate input (and not only on index).
DP6. I could explain why dynprog, using my order och computing, would solve the same problem as the recursion.
DP7. I could implement a dynprog algorithm presented to you in pseudocode.
DP8. I could decide, from the problem statement and given some time, whether a problem can be expressed recursively, without that being stated explicitly.
DP9. I could design a dynprog algorithm for a problem, given a problem statement without recursion.
DP10. I could construct a solution to a problem if I am given a dynprog algorithm deciding whether the solution exists.

Fig. 3. Self-efficacy items for dynamic programming, translated from Swedish.

C1. I could determine whether a decision problem lies in NP.
C2. I could determine whether a reduction is polynomial.
C3. I could explain the principles of an NP-completeness proof.
C4. I could determine in what direction a reduction should go in order to be able to use it positively or negatively.
C5. I could determine whether an NP-completeness proof is correct.
C6. I could choose a suitable NP-complete problem to reduce to a given problem.
C7. I could construct a simple NP reduction between two given problems.
C8. I could prove that my reduction is correct.

Fig. 4. Self-efficacy items for complexity, translated from Swedish.

comfortable with some tasks than with others, possibly indicating which tasks require more attention in teaching and more feedback during evaluation. This is the main use of the self-efficacy items in these studies.

5.2. The self-efficacy surveys

We would like to know whether students find any of the different tasks we believe are involved in dynprog especially hard, and whether students improve their self-efficacy during the course. Based on the ADC intended learning outcomes and the guidelines by Bandura in [Bandura 2006], we constructed self-efficacy items on dynprog, see figure 3, complexity, see figure 4, and proofs and pseudocode, see figure 5. The items are loosely held together by their theme, but it is not likely that either of them could contribute in the same way or amount to some score on the student's "total" self-efficacy on any of the themes. Hence the items should probably not be considered as comprising self-efficacy instruments for the various themes. Rather, the tasks we ask about are to some degree based on taxonomies such as Bloom's taxonomy [Bloom et al. 1956], dealing with separate tasks which could either be considered basic or something that students only need to learn if they aspire on higher grades. The surveys are further described in [Crescenzi et al. 2013; Enström 2013]. The students are for each item asked to grade their self efficacy on a scale from 0 to 100.

P1. I could implement an algorithm from a natural text description.
P2. I could implement an algorithm from a pseudo code description.
P3. I could understand what an algorithm does based on pseudo code.
P4. I could at an overview level describe what a program does, without going into implementation details, if I first read the code.
P5. I could identify which algorithm solves a given problem, if I was allowed to see the pseudo code for several algorithms.
P6. I could tell what problem an algorithm solves, given pseudo code for the algorithm.
P7. I could describe an algorithm with pseudo code.
P8. I could decide what is important to describe in the pseudo code, and what can be left out.
P9. I feel certain what format I would use for my pseudo code, and what symbols I should use.
B1. I know what needs to be included in the correctness argument for an algorithm.
B2. I could determine whether a correctness proof that was presented to me was complete and correct.
B3. I could understand why my algorithm is correct.
B4. I could explain orally why my algorithm is correct, to a teacher or another student.
B5. I could argue formally or prove that my algorithm is correct, in writing.

Fig. 5. Self-efficacy items for pseudocode and proofs, translated from Swedish.

The surveys were administered on lectures, before and after the sections of the course that was part of the experiments in 2012 and 2013. Apart from one survey – the proof and pseudocode survey of 2012, handed out together with the anonymous survey on the new activities of the course – the self-efficacy surveys were not anonymous, but the students were promised that it would not count towards grading, and that no one would read the material until after the course was finished. This procedure is naturally a limitation in the usability of the responses, possibly introducing errors in the form of more polished or adjusted answers from students. On the other hand, we wanted both to be able to see individual trends between the two occasions, and retain the option to later compare self-efficacy beliefs with results: both their predictive value from the first occasion, since it is known that self-efficacy affects motivation and possibly performance, and also possible correlations between results and self-efficacy beliefs after the teaching and learning activities.

5.3. Home assignment cover page

Together with the homework assignments, the students got cover pages to attach to their homework, with two types of questions. One half was a participation statement, and contained questions on what sessions or activities the student had attended in the relevant part of the course, and the other half was a "Where was what learned" part, and contained a matrix where students could mark where they had learned to master different aspects of dynprog and complexity, respectively: on their own, on lectures, on tutorials, from visualizations, from the peer assessed theory questions before the lab assignment, from the lab assignment, or if they still did not master it.

5.4. Course surveys

At the peer reviewed, pseudonymous theory exam, a final survey on the course was distributed in two versions, one with pre-defined answer options, and one with free-text questions. The students were randomly assigned one of these, and the one with pre-defined answer options is the one we present here. There was also, as always at our school, an online course survey, where some questions were about related issues. This survey is completely anonymous, done on the students' own time, and has higher non-response rate.

5.5. Results of the assessments

It is very hard to show that a change in a course has a positive effect by comparing assessment results, since there are so many variables involved in assessment. We do not believe in denying some of the students access to activities that we think are beneficial to them, so we cannot organize control groups. However, we know which activities the ADC students attended. Thus we can compare the number of activities attended to the student's performance on the two homework assignments.

5.6. Grading protocols from the individual assessments

To be able to view more detailed information on what students actually could do at their homework presentations, the grades being to blunt for determining this, a rubrics-like grading protocol was introduced. It allows the course responsible teacher to visibly (by marking some cells grey) communicate to TAs what performances are not sufficient for a passing grade, and at the same time makes the grading more transparent to students, who can be shown the protocol as example requirements or as motivation for an assessment.

When handing out homework, there probably remains some gaps between what students feel needs to be communicated, and what teachers find relevant. This mainly concerns the problem statement – naturally teachers expect students to do everything required by the problem statement, but also tasks that are *always* connected to the questions in the problem statement, but maybe not mentioned explicitly on each question. This is part of the socialization process into the role of a computer science professional (who knows about theoretical computer science). Sometimes, students do not accept that *any* requirement is not repeated for all tasks, while teachers tend to be more explicit for easier (corresponding to lower grade's criteria) tasks than for harder ones (where students are expected to be acting more independently as computer scientists with all tools and habits associated with this.) In the grading protocol, also many implicit criteria are present as to show that these are requirements and not a whim of the TA performing the grading.

This allows us to some extent to see whether some task was more often missed or not satisfiably accomplished compared to others. TAs were not required to complete the entire protocol if they encountered fatal errors, so we do not have a complete list of all errors made.

6. RESULTS

The findings of the previous studies are outlined and elaborated on here, and some additional data is included in the previous tables. Also, data on the proof and pseudocode self-efficacy surveys are presented and analyzed. The attempts to check for internal validity of the self-efficacy surveys, which was not done before, are also presented.

6.1. Automated assessment and programming assignments

The very first attempt at improving the ADC course included yet another, computer-assessed, programming assignment in the system Kattis. Students have since, according to course evaluation surveys and during-courses reports from student representatives about the course, come to perceive this assignment as a valuable practice occasion for the second homework of the course. The formative assessment side of the lab assignments is here emphasized – an opportunity to get feedback and practice without risking to lower your grade.

However, as described in [Enström et al. 2011], when the ADC course first introduced Kattis in 2006, the number of students who finished the lab assignments on time actually *decreased*. The passing rate of the lab exercises was presented in Table 1 in [Enström et al. 2011], but the grade administration system only returned data about the course in total, not the on time completion of assignments. More accurate data is presented here, in table I.

Table I. Percentage of first time students completing the three lab assignments of ADC; percentage completing them on time within parenthesis. The minimum admission grade for each year and the percentage of each year who were done with 2/3 out of the courses of the previous year are present for comparison. Those figures come from other sources. The order of the lab assignments have changed during the years, and in 2012 the rewrite lab was a new task. For the years 2011-2013, the number of students in the statistics is higher because other than CS major students are present. E1 is the maxflow algorithmic assignment, E2 is the refactoring/maintenance assingment and E3 is an natural language indexing assignment, which has never been transferred to Kattis.

Year	Nr. students	E1 (%)	E2 (%)	E3 (%)	Admission grade (min.)	Done with 2/3 of first year courses(%)
2000	115	96 (86)	94 (93)	94 (92)	18.28	–
2001	113	92 (78)	91 (90)	88 (87)	17.84	–
2002	103	89 (84)	90 (88)	88 (78)	17.55	–
2003	121	93 (91)	97 (97)	93 (87)	16.96	–
2004	129	94 (82)	92 (92)	88 (78)	15.33	76
2005	107	83 (67)	83 (78)	77 (64)	14.5	52
2006	92	71 (52)	83 (79)	84 (83)	16.18	55
2007	80	88 (66)	89 (88)	91 (88)	15.7	67
2008	106	81 (51)	85 (83)	83 (76)	11.52	66
2009	108	82 (49)	87 (82)	88 (82)	15.1	63
2010	113	84 (46)	88 (80)	89 (75)	15.43	64
2011	153	90 (54)	88 (83)	95 (78)	16.51	70
2012	183	80 (36)	95 (87)	95 (79)	18.77	64
2013	166	80 (69)	94 (90)	90 (77)	19.5	69

As mentioned by some student on the course evaluation that year, an assignment can be perceived as far more difficult due to the more thorough testing and less flexible assessment provided by an automated system. We believe that students previous years, erroneously, were passing the lab assignments only because TAs were not able to test the code thoroughly or spot some errors. They might also decide not to disappoint someone who had algorithms that solved the problem, but did not follow the output specification, for instance by presenting a path backwards, adding extra line breaks, etc.

We can se in Table I that especially the number of students who finished E1 (programming according to specification and implementing known algorithms), decreased, but also that there was actually a decrease the previous year. The numbers within parenthesis is the number of students who, according to the grade administration and reporting system, finished the assignment before deadline and hence received a bonus point for the exam. The E1 assignment consists of three separate tasks, in order to test various steps independently of one another, and is a practice exercise on the problem solving method of *reduction*.

The task E2 (maintaining and refactoring existing code to optimize it), was also affected, but not to the same extent. That assignment is simpler, and only consists of one task. E2 was in 2012 replaced by another task with the same structure and purpose, but requiring dynamic programming to be used. This task may be easier, or just better framed by the rest of the course, but completion rate is back to the level of before Kattis was introduced. The third task does not use Kattis at all, but still the completion rate dropped in 2005, 2006 and 2008.

It is possible that students' pre-requisite knowledge vary systematically between years. Therefore, for comparison, the lowest admission grade of each year (statistics from central admission office), and completion rate of the courses of previous year's courses (data from another administrative system at the university) are added. It is unknown whether the student with the lowest admission grade was an outlier, and if he or she was still studying, when the ADC course was given. However, around year 2000 the carrier option of computer science was very popular ("the IT boom"), and after that bubble crashed, fewer people wanted to study CS and the admission grades fell.

For teachers considering using automated assessment of programming assignments, this can be worth considering: Which assessment standards did the students previously have to conform with, and will these be changed? However, since the introduction of the tool on another, less advanced, course improved the student completion rate on that course, we believe that the changes we could see in the ADC were mainly due to more thorough checking of correctness than previous years, rather than due to the tool Kattis being difficult. More on this can be found in [Enström et al. 2011].

Later, when Kattis was more established, putting any assignment there makes the students feel more comfortable because they know how it works. Since there are new students each year, this knowledge seems to reside in the collective experience of CS students at KTH – in their culture. Kattis is now mostly experienced as a tool and a resource, and something taken for granted, rather than being experiences as an experiment that they need to deal with or, for those with that aim, something that makes it more difficult to mislead the teacher. (The latter has not been investigated, and I have no data on how common it is, but I have heard at least one narrative from former KTH students' personal experience, describing how they went about to distract a TA from noticing errors in a program, that the students were already aware of, with the purpose of not having to fix the errors.) The consequence of the greater acceptance of Kattis is that students wanted a practice exercise on dynamic programming on Kattis as well as a complexity practice exercise, which was introduced in the second cycle of the project.

Two things in particular are worth mentioning about the Kattis results of [Enström et al. 2011]: firstly, when assessed by an automated system, students cannot negotiate the quality criteria as they try to do with a human teacher. This means that the teacher's quality criteria are enforced, while at the same time Kattis takes the blame for being unreasonable. Secondly, when the feedback is limited, the feedback that is actually there gets more emphasized. Since Kattis in her "vanilla" version only responds with the result (Wrong answer, Run time error, Time limit exceeded, Compile error or Accepted being the most common outcomes), together with the CPU time, and given that there is a time limit that in the case of ADC can be difficult to come down to, students will start optimizing their code to reduce the CPU time. The competitive side of the Kattis assignments is encouraged both by Kattis' design and by TAs. In this course, for all assignments apart from the complexity reduction assignment, this is in line with the course goals. It is important, however, to clearly state if the received feedback is not worth optimizing for. For the complexity assignment, the only thing to optimize would be I/O, which is not among the course goals to optimize.

Kattis usually presents the CPU time of a submission to its author, and entertains a "high score list" of the submissions with the lowest CPU requirements. Kattis also offers teachers the option to provide other types of feedback to students, and in some cases this is more relevant. For instance, some assignments have a grading program that calculates a score depending on how good the solution was. This makes Kattis present a high score list based on the score instead of lowest CPU consumption, which is pedagogically preferable.

In conclusion, the result of introducing automated assessment can, as seen by this example, include decreased passing rates for the course, if assignments get more difficult, or rather, if negotiating the quality criteria of the course gets more difficult. It may also include a more positive relationship between students and TAs, if the latter are not responsible of failing programs that do not work, and it may increase time-on-task and engagement with the tasks if the feedback concerns some optimizable characteristics of the program.

6.2. Framing of exercises/Verbal and non-verbal communication

In this section, we mention some observations from all of the cycles, regarding communication and language use.

The Kattis complexity assignment, described in [Enström and Kann 2010], was introduced to help practicing theory in a more familiar setting: the lab assignment. It was perceived as an extra task, increasing the course work, the first year, but also as something that helped students learning complexity. At the same time, it was – and still is – common that some students complain about the exercise being underspecified, tricking them into spending too much time on something that was really simple if it had only been expressed from the beginning. It also happens that students shrug and say that the task was completed in 30 minutes and needed only half a page of code, a seemingly pointless exercise in comparison to the other tasks, or that TAs need to stop people from trying to receive a lower running time of their program, since it hardly does anything but reading input and writing output. A result of our approach is then the views that the task is underspecified, too easy, or should be optimized like the other assignments in the course. These are likely to be consequences of using this familiar setting for an unfamiliar task.

The number of students who handed in homework solutions with useless reductions (look, this problem can be solved if we can solve the hardest problems in NP!) seems to have decreased since this assignment and other complexity activities were introduced. Those who need hands-on experience to realize the implications of a reduction, can get that experience before jeopardizing their grades by failing the homework.

The students’ expectations on a task sometimes differ from ours, and this can be exemplified by language differences between teachers and students. Whereas teachers typically want a student to present an algorithm, show that it is reasonably efficient, show that it terminates and that it solves the problem it is designed for, as a response to a “devise an algorithm that. . . ” problem, our students tend to believe that the only thing needed is to present an idea for the algorithm. This misconception can in at least two ways be fostered by the teaching: sometimes, only the general idea and not the details can be discussed at lectures. This can be taken for an example of an accepted strategy. More often, lectures go through explaining the algorithm, writing pseudo code, analyzing it and showing correctness for it, but students might perceive most of these steps as some sort of pedagogical decoration of the main purpose of the task – to get an algorithm idea. Instead of seeing the lecture as an example of how to solve problems in the context of the ADC course, someone could believe that the teacher just was keen to have everybody understanding the algorithm. The *purpose* of the correctness arguments is not acknowledged.

Another language related, common issue is that for the complexity homework, when asked to show that a problem $X \in NP$, some students try to show that they can verify *one* solution in polynomial time, while clear on the fact that you cannot verify *any one* solution. In Swedish, the counting word for “one” and the definite article for common gender nouns both are “en”. So, if you believe that the task is specified word by word in the problem statement and that no terms are special to the field of study, it is sufficient to find one instance where the solution is verifiable in polynomial time.

There is another communicative feature of the research activities this course has been subject to: the many surveys. It is unlikely that it does not in any way affect students if we regularly ask them how certain they are that they can perform some list of tasks. The items on the survey could probably function as check lists for “things teachers want me to know”. This was brought up by some students during the first year of self-efficacy surveys, and therefore we investigated whether this was common, but most students did not consider the surveys part of the teaching and learning experiences. Regardless of that opinion, it is a way in which these surveys *can* function. To some extent, it depends on the students' mind set when completing the survey if it also becomes a moment of self-assessment, introspection and reflection. To some extent, it depends on how the teachers present and describe the surveys.

6.3. The survey results

At the final written theory exam, in 2011–2013, each student received one of two evaluation surveys. The first one was an open question survey with questions about the pedagogical purpose of each activity and whether this purpose was fulfilled. The second survey consisted of closed questions on the meaningfulness and usefulness of the activities. In 2011, all students who got this survey answered it except two students, a total of 59. In 2012, all but one answered the survey, a total of 70, and in 2013 74 students responded. In 2013, there was an additional question asking which year they took the course, and 68 of the responding students had participated in ADK 2013.

The results of the closed version of the surveys, for the questions concerning complexity, are summarized in Table II. Similar responses were also given for dynprog-related questions.

Each question also had “no opinion” as well as “did not participate” alternatives, and these are not presented in the summary.

Most activities seem to have been perceived in similar ways all three years, with the apparent exception of the motivational lecture, where more students were inclined to answer “don't know” in 2012 and 2013. Afterwards, many students expressed confusion over the term “motivational lecture”, which explains this. The role of the motivational lecture might have been most emphasized the first time it was given. Generally, when students were less positive than in 2011, they chose “don't know”, with the exception of the visualizations, which were seen as not contributing to learning reductions by almost half of the responding students in 2013. When asked whether visualizations helped them learn dynprog, 35 % of these students said no. This number is more similar to how large fraction of students in 2012 and 2011 claimed that they did not learn complexity.

Another change is that 40 % of the students in 2011 did not believe that the clicker questions helped them learn computational complexity, whereas this fraction decreased to less than half the relative size in the following years. This may caused by the questions improving, students getting more used to them, or their function becoming less of an experiment and more of a permanent arrangement. In 2011, the clicker questions were only used for the lectures in complexity, but for subsequent years, questions were prepared for all topics of the course. Still, already the first year, most of the students wanted more clicker questions [Crescenzi et al. 2013].

In 2012 and 2013 we also asked the students to indicate on a cover sheet for the homework assignments, where they learned what (multiple choice), see Table III and Table IV.

Table II. Activity survey results (2011-2013). Answers are presented as *first year/second year/third year* percentages. Similar results were obtained for dynamic programming.

1. Was the pedagogical purpose of the activity clear?

	yes	questionable	no
motivational lecture	86/23/32%	11/16/6%	4/4/3%
clicker questions	92/90/94%	8/6/4%	0/1/0%
visualizations	80/81/71%	16/10/16%	3/1/3%
reduction computer lab	90/81/85%	6/10/9%	4/0/0%

2. Did you find the activity meaningful?

	yes very	yes somewhat	not particularly	not at all
motivational lecture	21/10/9	61/17/21	7/10/7	-/-/1
clicker questions	38/59/54	44/36/32	13/3/9	4/0/1
visualizations	28/40/22	38/40/46	26/10/18	8/1/6
reduction computer lab	65/67/62	35/23/26	0/1/3	-/-/0

3. Did you learn some computational complexity by working with the activity?

	yes	no
clicker questions	50/69/65%	40/11/19%
visualizations	45/33/26%	34/34/46%
reduction computer lab	94/86/87%	4/6/6%

4. Do you think that activities like this one can make it easier to learn computational complexity?

	yes	no
clicker questions	69/73/74%	19/9/12%
visualizations	95/69/68%	5/7/10%
reduction computer lab	96/89/93%	2/3/0%

5. Did the activity add something to the course?

	yes	no
motivational lecture	75/27/28%	4/13/4%
clicker questions	83/94/91%	6/1/4%
visualizations	76/71/65%	13/6/13%
reduction computer lab	94/91/93%	2/1/3%

Table III. Where different dynprog tasks were perceived to have been learned in 2013. Data from the cover sheet of homework 1. Multiple choices possible. N=142.

Where did you learn to. . .
1. . . . decide if an algorithm is using dynprog?
2. . . . decide if a problem could be solved by dynprog?
3. . . . design a recurrence relation for a simple problem?
4. . . . choose an evaluation order, given a recurrence relation?
5. . . . construct a solution given a dynprog algorithm that returns the optimal value for the solution?
6. . . . motivate correctness for a dynprog algorithm?

	lectures	tutorial sessions	lab theory assignments	dynprog lab	homework 1	still haven't learned this	already knew or learned by myself
1.	51%	26%	54%	49%	14%	1%	21%
2.	49%	24%	45%	42%	18%	4%	22%
3.	34%	30%	24%	18%	10%	6%	49%
4.	37%	30%	25%	23%	10%	16%	29%
5.	35%	25%	32%	38%	16%	16%	18%
6.	43%	15%	9%	8%	31%	28%	19%

Table IV. Where different complexity tasks were perceived to have been learned in 2012/2013. Data from the cover sheet of homework 2. Multiple choices of answers available for respondents. N=148/N=126.

Where did you learn to…
1. …tell if a decision problem is in NP?
2. …describe the principles for an NP-completeness proof?
3. …choose a suitable NP-complete problem to reduce?
4. …construct a reduction between given problems?
5. …prove correctness of an NP reduction?
6. …reduce an optimization problem to a decision problem?
7. …reduce a constructive problem to an optimization problem?

	already knew or learned by myself	lectures	tutorial sessions	visualizations	lab theory assignments	reduction lab	homework 2	still haven't learned this
1.	20/19%	59/63%	37/42%	2/2%	43/41%	39/41%	43/43%	0/2%
2.	14/18%	59/56%	44/46%	1/2%	30/24%	34/40%	36/42%	0/1%
3.	17/20%	46/48%	36/29%	1/3%	14/13%	28/30%	51/55%	3/6%
4.	14/11%	41/44%	36/33%	0/2%	24/22%	57/54%	52/52%	1/10%
5.	16/17%	34/40%	32/37%	0/2%	13/9%	32/33%	43/46%	5/12%
6.	19/16%	36/57%	18/29%	0/1%	3/3%	5/6%	40/48%	18/6%
7.	11/13%	23/44%	15/29%	0/0%	1/1%	3/6%	38/36%	32/29%

These tables do not entirely evaluate the same phenomena as does the final survey, and should not be used to decide what activities should be part of the course. They ask about a number of tasks, sorted in increasing "quality" according to course grading criteria, and where students have learned these tasks. Students are to mark each activity that has helped them learn a task in a matrix on the cover page. The homework assignment is the final test on these tasks. For instance, according to table IV, the students did not perceive that the visualizations had contributed much to their present knowledge on the complexity items we asked about, yet 33% of the students on the final survey stated that they had learned complexity from the visualizations, and that these were meaningful and contributed to learning. Those answers were given later and in retrospect, while the results in Table IV were supplied together with the second homework assignment, and also on the latter, students might compare the relative contributions of also the usual course activities, and judge these as more useful than the visualizations. Apart from the visualizations, all activities were considered contributing to learning. Lectures are for most tasks considered beneficial by the largest fraction of the students. The lab theory assignments also seem to be useful for students, but to decreasing extent as the teacher estimated difficulty level of the task increases. The relative difficulty of the items seems to correspond well to the fraction of students who have still not learned each task – this fraction increases row by row in both table III and table IV, with the exception for complexity task 6. To reduce optimization problems to decision problems is, according to the grading criteria, something that lies on a reasonably high level (grade B(?)), but at least in 2013, more students perceived designing reductions between two given problems difficult, than performing optimization-to-decision problem reductions. The lab theory assignments are roughly contributing to the learning of both dynprog and complexity tasks to decreasing extent for each item in the matrix. This is consistent with the aim of the questions: to help students consider relevant aspects of the problem, and choose appropriate approaches and/or data structures for the task, rather than acquiring really difficult skills.

The tasks involved in dynamic programming were more often already familiar to students, before the course activities, than the complexity tasks. This is especially true for the design of recurrence relations, which was something we had expected to be perceived as more challenging.

Considering how many students claiming not to have learned each item, the most difficult dynprog task is to motivate correctness. The most difficult computational complexity task is, with the same interpretation, to solve construction versions of problems given algorithms for the optimization version. This latter is the second least learned task of dynprog – it is something that you may need to do regardless of whether you are working with tractable or intractable problems.

6.4. Visualizations, clicker questions and new structure

As seen in the survey results, the visualizations were the least appreciated new activity, yet most students were very positive towards them. Only a minority claimed to have learned from them. It is in our opinion a good idea to address learners with visual preferences as well as those preferring verbal instruction. Many complicated algorithms can be easier described with visualizations than by tracing execution on the board.

The questions at the lectures were extremely appreciated among the students, and when asked on the free text answers surveys for the likely purpose of these exercises, students mentioned activity, thinking, adopting standpoints, interaction, feedback to the teacher and to the students about the lecture’s progress, and so on. The teachers and students appeared to be in agreement on why the questions were there, and they were appreciated by both groups, although it is the *aggregated* responses of the student cohort that corresponds well to the teachers’ ideas. It did not put students off to have to raise a colored card in class (compared to pressing an anonymous button), but when a clicker system was borrowed and utilized, students liked it.

Once introduced, this type of questions during lectures are unlikely to disappear. Reports on experiments with clickers often involve teachers being most appreciative of the systems. This is likely not due to a feature of any particular system, but a consequence of the new lecture planning and lecture process, that can be much more rewarding for the teacher as well as for the students. Preparing good clicker questions, even in the absence of clickers and even in the absence of response cards, is a practice we recommend. The simultaneous vote is probably better than asking for one answer option at the time, so for instance having students point at the right wall for “yes” and the left wall for “no” seems to offer students less concerns about loosing their face than asking first for yes answers and then for no answers.

Regarding the new structure of the dynprog lectures, many students were (fortunately) unaware of this being an experiment. They did however appreciate all interventions we came up with, maybe consistent with the positive change bias.

6.5. Comparison to assessments

The homework assignments assessing the problem solving proficiency, the first is on algorithm construction and the second one is on computational complexity, were described in Section 2.1.

One of the main reasons for changing the course was to improve the performance ratio of the first time students at the complexity homework, that is, the share of the students passing the homework the first time it is given. The year before the project started the performance ratio was 73%. The first year of the project the ratio was the same, but the second year it was improved to 87%. In 2013, 69 % of the first time students passed the second homework. This year, the tasks of the homework might have been more difficult, or students less prepared for mathematical tasks.

Table V. Comparison of student performances at homework 1 and 2 (hw1 and hw2) depending on the number of new activities attended (ADC), years 2011-2013.

	Activities attended	hw1 grade <hw2 grade	mean grade hw1	mean grade hw2
year 1	>4	41%	3.2	3.6
	>2 and ≤ 4	36%	3.3	2.9
	≤ 2	20%	2.4	1.9
year 2	>4	49%	2.0	2.7
	>2 and ≤ 4	34%	1.7	2.1
	≤ 2	11%	1.7	2.0
year 3	>4	40%	2.5	2.8
	>2 and ≤ 4	25%	2.2	1.9
	≤ 2	24%	2.0	1.4

How *many* students are passing the assignments is not all there is to know. In [Enström and Kann 2010] we found that the ration between number of students failing the second assignment and those failing the first, was greater than one before the reduction lab assignment and smaller than one afterwards. This method, however, does not show a stable trend. It is also not as useful once the teaching of algorithms became the object of our refactoring efforts, since now both of the assignments are better framed by other course activities, than was the case in 2008. In [Crescenzi et al. 2013] we also compared the results on the complexity homework with the number of attended activities: motivational lecture, clicker questions, demonstration of visualizations, student use of visualizations on their own, peer review of lab theory, and submission of NP reduction computer lab to Kattis. The NP reduction computer lab was counted as 1.5 if submitted early (i.e. not later than the specified due date). Out of the students failing the complexity homework in 2011, only a fourth had attended three or more of the six activities.

Attending many activities might only indicate activity from the student's part, and active students are likely to receive better grades than those who are not studying as hard, regardless of the quality of the activities provided. Therefore we compared the attendance information on complexity activities to the two grades that a student received for homework assignments. 114 students in 2011, 136 in 2012 and 135 in 2013, handed in both homework assignments, 34/57/53 students received a better grade in the second homework than in the first, 46/10/42 students received a better grade in the first homework than in the second, and 34/69/40 students received the same grades. The percentage of students who had higher grade on homework 2 than on homework 1 for the years 2011–2013 are presented in table V, split up into sub categories based on how many activities the students had attended. The average grade (on a linear scale where A is 5 and F is 0) for each subgroup on each of the homework assignments are also present. For all three years, those attending many of the complexity activities were more likely to have a higher grade on the second, complexity, homework than on the algorithms homework. The average grade was consistently higher on the second homework among those who attended more than four activities. This remained the case also after the dynprog restructuring cycle of these action research interventions.

Note that the activity attendance information of 2011 is not complete. Some students in the two lower attendance categories ($\leq$2) might actually belong in the middle or upper category, since for those who supplied no attendance information, we only have information about the mandatory and assessed course activities. For 2012 we have full attendance information. Also, we know for 2012 that the correlation between the attendance of the algorithm and complexity parts of the course is high.

This indicates that the activities on complexity had a positive effect, or possibly that students benefit more from teaching in complexity than in algorithms. It is not possible to use the homework grades in the same way for evaluating the dynprog activities, as dynprog is only a minor part of the algorithms homework assignment, and the dynprog task sometimes is the easiest one, and sometimes is the hardest one, on that homework. It would however be surprising, and decrease the credibility of the method, if the same type of result would be visible for the dynprog activities. In [Enström 2013, Table 4], we performed that type of check for the data from 2012, and attending many dynprog activities did not correlate with a higher grade on the algorithms homework than on the complexity homework that year. (The average grades 2012 might have been slightly better, in total, if some students would not have used dynprog too often, choosing it in favor of a more efficient, greedy solution to the third task, but that is unlikely to affect the average grades enough to change this situation.)

6.5.1. Self-efficacy on complexity. In 2012 and 2013 self-efficacy surveys on the computational complexity part of the course were given. They consisted of 8 items, see figure 4. The differences in self-efficacy scores of these surveys are presented in table VI.

Too few students out of those who completed both surveys had attended few activities. Hence, the activities' possible impact on self-efficacy cannot be calculated from these results.

The total results for both years are presented in Table VI as 2012/2013 results. The items C4, C5, C6 and C8 relate closely to the three difficulties identified by us, and the average final score is lower on these both years.

The lowest mean increase was for C4, both years, indicating that students after the complexity teaching still were feeling less certain about directions of reductions for various purposes. However, the median score for C4 on the second survey was 90 in 2013, so a majority did not feel uncertain. It is also worth noticing, that the teaching assistants who were grading the homework and the oral exam where this assignment was presented, reported in 2012 that this year (unlike all previous years) there were no occurrences of reductions in the wrong direction. The students did not increase their beliefs in their own abilities on this one as much as on the other items, but they performed better than students had done before with respect to this particular item.

Items C5 and C8 concern correctness proofs, which we already believed to be difficult, and in the median values of 2013 it is particularly clear that these belong in another category than the other items, together with C6 (choosing a suitable problem to reduce), suggesting that comparing problems with respect to similarities and difficulties, as in POI, in structure and purposes could be useful also for the complexity part of the course.

Item C2 had a high starting score in 2012, and did therefore not increase much. Most students increased their self-efficacy during the period, but there were some occurrences of decreased self-efficacy, one for item C5 and two for item C4.

Table VI. Statistics over self-efficacy scores for students (N=35/N=72) participating in both complexity surveys 2012/2013. 79/78 students answered the first survey and 45/144 answered the second survey.

	C01	C02	C03	C04	C05	C06	C07	C08
mean, survey 1	39/32	49/35	30/24	42/34	28/13	25/16	33/18	27/15
mean, survey 2	89/90	88/92	85/90	65/81	68/74	70/75	87/84	75/75
mean difference	50/58	39/56	55/66	22/47	40/61	45/59	54/66	48/60
median, survey1	50/30	50/40	25/18	45/32	25/0	25/5	25/6	25/0
median, survey 2	95/98	90/95	90/95	75/90	75/75	75/75	95/88	75/75
median difference	50/55	40/52	50/72	25/50	50/65	45/70	50/75	50/65

6.5.2. Self-efficacy on dynamic programming. Also surveys on self-efficacy in dynamic programming were distributed both in 2012 and 2013. These consisted of 10 items, (see figure 3). Statistics for these surveys are presented in table VII.

Table VII. Statistics over self-efficacy scores for students (N=68/N=43) participating in both dynprog surveys 2012/2013. 110/73 students answered the first survey and 79/78 students answered the second survey.

	dp01	dp02	dp03	dp04	dp05	dp06	dp07	dp08	dp09	dp10
mean, survey 1	76/59	49/57	70/51	88/83	34/36	49/44	71/65	69/55	47/38	36/37
mean, survey 2	90/87	84/89	87/85	95/97	68/76	78/81	88/89	80/82	67/75	65/67
mean difference	14/28	35/32	17/34	7/14	33/40	29/37	17/24	11/27	20/37	30/30
median, survey 1	80/67	50/60	75/50	96/100	25/25	50/40	75/75	75/60	50/35	28/30
median, survey 2	95/90	88/90	92/90	100/100	75/80	80/85	90/90	80/90	72/75	75/70
median difference	14/25	30/25	20/35	0/0	32/40	28/40	10/20	6/25	20/35	32/30

DP4 has the highest starting *and* final value both years, where for instance in 2013 the median is 100, signaling that students are comfortable with recurrence relations, and contradicting the assumption that these might make dynprog seem difficult.

DP5 starts out at a low level, but the final average is comparable to other items. It has the highest increase, so the “evaluation order” is an unknown concept in the beginning, but is later not as strange. The same happens for DP10 and, in 2013, DP9.

The most difficult tasks are represented by items DP9 and DP5, related to seeing the structure of subproblems and their solutions, choosing an appropriate evaluation order, and solve an entire problem, and DP10, again indicating that constructing solutions by reduction to an algorithm solving the existence of a solution, i.e. reducing the construction version of a problem to the decision version, is difficult. This points at reductions being difficult *both* in proofs by contradiction and in solving problems, meaning that all difficulties with reduction in complexity might not be due to the “backwards” application of them.

If we compare the figures in table VII and those in table table VI, the dynprog items are both in the beginning and in the end, perceived as less challenging compared to the complexity items. This is confirming the observation that algorithms and problem solving are familiar to our students, while complexity is a new type of topic which they are less used to dealing with.

6.6. Self-efficacy on pseudocode and proofs

In preparation for the third cycle, the students received a self-efficacy survey on pseudocode and proof at the exam 2012, together with the anonymous course survey.

The same survey was then used, non-anonymously like the others, in 2013 both before and after the entire course. Unlike the previously investigated topics, these topics are considered pre-requisite to some degree, and are not explicitly taught in ADC.

The items are presented in figure 5 (note that items P9 and B1 are not phrased as self-efficacy items, which was commented by some respondents) and the results from 2013 in table IX. Since both new and older students completed the survey in 2012, all responses are counted, except for the rows about differences between the two 2013 occasions, where only the students who completed both surveys are counted.

The pseudocode items all start with higher values than does the complexity and dynprog items. This is reassuring, since students should already be used to pseudocode. The average increases are modest, with the exception of P9 which started at 49 and increased to 80.

Among pseudocode items, P5, P6, P8 and P9 seem to have been a bit more difficult than the other items. The two last items correspond to the choice of presentation means, and we had already noticed that students often seemed unsure of what they needed to include and how to present the algorithms. For P5 and P6, some spontaneous comments said that this would entirely depend on the difficulty level of the questions/algorithms.

P6, dealing with the ability to translate pseudocode information to the real world and possible macro perspective ideas like what task the algorithm is designed for, initially had the median value of 60, just above P9. “Translating” various algorithms to problem descriptions and decide whether any of them solves the described problem started out higher, but both had final median only 80, which for this set of items is low. Some students do recognize difficulties comparing understanding of an algorithm at micro level, row by row, with understanding *why* the algorithm works like that. Going in the other direction, from an algorithm solving a problem, to presenting it in pseudo code (P7), is not perceived as difficult – but the differences are small.

The proof items are more similar to the complexity items, or start out even lower. For B1 and B2, the increased self-efficacy both on average and as median, is large. Students are more certain of what needs to be proven, and feel more comfortable valuing proofs. Still, both median and mean values are around 70. For the two following tasks: understanding why an algorithm works, explain it orally, students feel better prepared already in the start. They *do* know about correctness, but are not certain what counts as a proof, and continue to be marginally more certain of why an algorithm works than what is required to prove it. They are also more comfortable actually attempting to prove correctness, than to judge whether a given proof is sufficient!

In table X, the responses about pseudocode from 2013 can be compared to the responses from the anonymous survey in 2012. The same comparison of proof self-efficacy beliefs are presented in table IX.

In table IX the responses on the proof items are presented.

Table VIII. Differences in self-efficacy scores for students (N=112) participating in both proof/pseudocode surveys S1 and S2 2013. 145 students answered the first survey and 144 students answered the second.

	p01	p02	p03	p04	p05	p06	p07	p08	p09	b01	b02	b03	b04	b05
mean, S1	76	80	78	72	67	61	69	64	49	17	18	52	47	33
mean, S2	85	90	87	84	80	78	88	81	80	68	63	78	76	67
mean S2-S1	9	9	8	12	13	16	19	17	31	51	45	26	29	34
median, S1	75	80	75	75	75	60	75	70	50	10	10	50	50	25
median, S2	90	90	90	88	80	80	90	82	85	70	70	80	80	75
median S2-S1	5	10	8	10	13	15	17	15	30	50	50	25	28	35

Table IX. Self-efficacy scores for proof study. The increases presented for 2013 are from before to after the course. In 2012, 130 students answered the anonymous final survey. In 2013, 145 students answered the first survey and 144 students answered the second. In total, 110 students responded to both surveys in 2013.

	B1	B2	B3	B4	B5
median 2012	60	60	75	75	65
mean 2012	57	53	72	74	62
median 2013	70	70	80	75	75
mean 2013	68	63	78	75	67
median increase 2013	50	50	25	28	35
mean increase 2013	52	46	27	30	34

Table X. Self-efficacy scores for pseudocode study. The increases from 2013 are increases from before to after the course. In 2012, 130 students answered the anonymous final survey. In 2013, 145 students answered the first survey and 144 students answered the second. In total, 110 students responded to both surveys in 2013.

	P1	P2	P3	P4	P5	P6	P7	P8	P9
median 2012	80	90	85	80	75	75	90	78	75
mean 2012	83	89	84	80	73	68	85	77	71
median 2013	90	90	90	85	80	75	90	80	80
mean 2013	85	90	87	84	80	77	89	82	80
median increase 2013	5	10	8	10	13	15	19	15	30
mean increase 2013	9	10	9	12	14	17	20	18	32

In 2013, students were somewhat more certain (on average) that they could perform the tasks compared to 2012. We can see that the increase in self-efficacy regarding pseudo code was very small during the course, and that the differences between 2012 and 2013 seem rather similar for both proofs and pseudo code. This *could* imply that the students in 2013 were "better" at these things than the 2012 students, or generally more confident, rather than that the teaching caused the difference between years.

6.6.1. Correlation of self-efficacy value and performance. A property that we might want the self-efficacy values to have is to correlate to the real mastery of the student's abilities, as measured by a classical assessment task. Therefore, we attempted to correlate self-efficacy beliefs with actual performance on the course. Out of our 32 items, we found one or several mandatory tasks, possible errors on homework assignments, and particular exam or homework questions, that could be related to half of them. The correlation between the second survey's answers and these data on students' performance were very small and the plotted data points did not manifest any pattern. The item with highest correlation was C8, to perform a correctness proof for a reduction, for which the correlation coefficient was 0.43 (Pearson). The plotted data is shown in figure 6.

Either the absolute values of the self-efficacy beliefs do not correlate with performance, in the current wording of the items, or the difference in scope between the examination tasks and the items is too large and the data therefore too uncertain. Also, the time between when students performed the assessed tasks and when they completed the surveys, might have brought about changes. Other steps need to be taken to resolve this.

For the possible negative correlations, i.e. errors reported by TAs that, if the students have "realistic" self-efficacy beliefs, could correlate negatively with the reported beliefs, there were so few data points that this method was not used.

6.7. Different years and different students

When comparing course results for different years, potential differences between the average student of each year is not taken into account. In Table I, some additional data about the students is included. We have also checked whether the admission grades and the grades on pre-requisite courses are correlated to ADC. It appears to be the case for pre-requisite courses, but not for admission grades.

In Table XI, the average admission grade for students with each of the possible ADC grades are presented for the years 2011-2013. Also for homework assignments, the pre-requisite courses' grades roughly correlate with performance, see Table XII.

Comparison with admission grades can be seen in Table XIII. The same trend is by no means present for this data. Admission grades do not seem to correlate with ADC grades. Similar results were obtained for homework grades. Students are competing to study at KTH either with their high school grades, or similar, or with their results

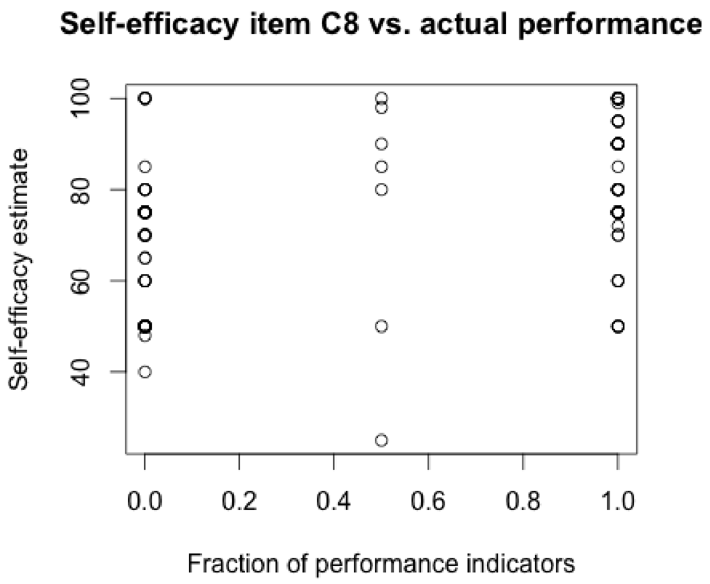

Fig. 6. Estimated self-efficacy for item C8 plotted against the fractions of (two) positive performance indicators out of the mandatory and documented tasks of the ADC course (2013).

Table XI. Comparison of ADC first time students' final ADC grades with their average grades from four pre-requisite courses. Maximum pre-requisite grade: 20.

Course grade	Average grade (# of students) 2011	2012	2013
A	14.43 (23)	14.67 (18)	16.86 (14)
B	13.15 (13)	16.17 (6)	11.00 (1)
C	12.36 (11)	14.35 (23)	15.52 (27)
D	14.71 (7)	7.67 (3)	12.36 (11)
E	9.44 (43)	10.13 (55)	10.69 (26)

on an admission test. In Table XIII, various types of grades have been normalized to the same scale, and two types of admission test scores, with maximum 2 and 2.5 respectively, are merged. Some students were admitted based on previous study achievements at KTH and some based on foreign countries' grading systems. The maximum scores for these students are unclear, and they are not counted in this statistics. Explaining different year's different ADC outcomes with the admission grades of that year's students is unsuitable. Some students may delay their study start at KTH, or have a years' leave for military service, pregnancy or other studies, which means that the individuals who are first time students on ADC any year are not necessarily included in the statistics, especially in Table I, but a large group of them are, and the statistics is valid for this group.

7. DISCUSSION

Some final thoughts on the three areas of this research.

Table XII. Comparison of ADC first time students' homework grades with their average grades from four pre-requisite courses. Maximum pre-requisite grade: 20

Homework	Grade	Average grade (# of students)		
		2011	2012	2013
1	A	12.60 (15)	16.00 (8)	18.00 (5)
1	B	13.73 (22)	14.40 (5)	–
1	C	11.41 (22)	13.54 (35)	13.94 (53)
1	D	12.88 (17)	12.40 (5)	10.75 (8)
1	E	7.86 (29)	9.67 (69)	9.88 (32)
1	F	5.80 (5)	12.00 (3)	7.00 (2)
2	A	14.53 (34)	15.12 (24)	16.68 (22)
2	B	11.00 (5)	16.11 (9)	13.29 (7)
2	C	12.18 (11)	12.19 (21)	13.77 (22)
2	D	12.62 (8)	10.78 (18)	12.00 (6)
2	E	9.14 (42)	9.41 (51)	9.94 (16)
2	F	6.00 (5)	7.00 (2)	9.78 (23)

Table XIII. Comparison of ADC 2011-2013 first time students' grades with their average admission grades (G) or admission test results (T). Maximum G: 22.5, maximum T: 2.0

Course grade	2011 average (#)		2012 average (#)		2013 average (#)	
	G	T	G	T	G	T
A	18.53 (17)	1.40 (5)	20.75 (14)	1.80 (4)	21.35 (9)	1.63 (3)
B	18.49 (9)	1.50 (3)	20.43 (5)	2.00 (1)	19.40 (1)	
C	18.02 (8)	1.60 (1)	20.77 (16)	1.66 (5)	21.22 (15)	1.61 (10)
D	18.55 (6)	1.70 (1)	20.10 (2)	1.30 (1)	20.49 (5)	1.20 (1)
E	18.13 (26)	1.55 (8)	19.89 (38)	1.55 (10)	20.51 (19)	1.64 (5)

7.1. The information conveyed by self-efficacy beliefs

The proposed method of showing correlation between self-efficacy values and performance by utilizing existing assignments in the course failed, see section 6.6.1. On the one hand, when describing the idea of self-efficacy, Bandura points out that the surveys should not be presented in such a way that they suggest to respondents that there is an objectively correct answer that their answers will be assessed with respect to. On the other hand, when investigating correlation between self-efficacy values and performance, it is likely to be important that not too much time passes by between the survey and a measurement of the investigated abilities.

Utilizing already present tasks for evaluating the predictive value of the reported self-efficacy beliefs conforms with the aim of not giving the impression of testing how "true" the self-efficacy beliefs are, but to evaluate very precise abilities by tasks involving many other abilities as well, is maybe too blunt a method.

One reason for the failure of our evaluation of the items, may be that we required too little information from those assessing the students. Preferably, short of designing specific tests for all students just to check their actual abilities on all these tasks, we could investigate a couple of items at a time, design course tasks that will provide suitable information, and require TAs to give detailed reports on these particular tasks and abilities.

Individual variation in the calibration of the self-efficacy values could also explain the lack of correlation – the value 80 might for example mean very different levels of mastering of the item for different students.

It is however also a possibility, that the self-efficacy items themselves are of too low a quality, containing impossible ambiguities and making the responses arbitrary. A couple of students experienced them in that way, and invented their own ways of answering, for instance by assigning binary values to each item, or to respond with intervals. The task an item concerns, can be of varying difficulty level depending on how difficult the problem that the task is “illustrated with” is. There is bad pseudo code to try reading, there are inherently tricky problems to solve, and so on.

7.2. The complexity assignment versus other lab assignments

The Kattis complexity assignment, described in [Enström and Kann 2010], is unusual for the ADC course in several ways. Firstly, the time limit requirements are not difficult to meet with, provided you try solving the correct task. If you try an exhaustive search approach to the new problem, instead of reducing something to it, it will of course time out. Secondly, the programming in itself is not difficult. The task is actually only there to support the process of constructing a valid reduction, and the main purpose is not to practice programming. Thirdly, the purpose of the task is more “constructed” and needs more explanation, than what is needed to motivate “solve this problem”. Why do we need to show that this problem solves another problem?

This is not obvious to all students. As mentioned above, it is common both to receive complaints because the assignment is too easy, and complaints about it being unclear and underspecified. It *is* underspecified, if viewed within the “regular” Kattis context. Usually, an assignment has a problem statement, input and output specifications, and some sample inputs with corresponding sample answers. This is reasonable and the right thing to include if the purpose is to solve a programming problem, or to practice programming according to specification. However, for this particular exercise, the task of constructing a (very simple) program that transforms the input of problem A to input of problem B would be extremely decontextualized, and pointless.

To practice NP-completeness reductions, at least some trace of the “choose a problem to reduce” must be present. Hence, the input cannot be definitely specified. Instead, input of two problems is presented, and the students should choose which one resembles the casting problem, that is the new problem with unknown complexity that is introduced in the lab, the most – or look at the choices of most other people.

Then there is the problem of knowing what a reduction can be used for, and therefore in which “direction” to reduce. This means that we do not want to specify what the output format is, either. Students need to read the statement and understand what the task is, which makes this assignment different from all other Kattis problems, but similar to the type of problem dealt with in the course at this point. This context switch is not clear to all students, and every year some students complain about “lost” time when they were trying to reduce in the wrong direction. As teachers, we believe that the experience of first trying to reduce the wrong problem, and then realizing this and what should really be done, is most educating, and irreplaceable by instructions.

7.3. Complexity

In order to connect all three parts of reductions that students find especially hard, coming up with an idea, prove correctness and understand the implications of a reduction, we believe that we should both teach each item separately, and show how the parts relate to each other and to other parts of the students’ knowledge, for instance by showing the connections to algorithm construction. If a student has the preconception that everything in the course will be about algorithms, an exhaustive search would seem just as good as any other algorithm for any purpose. On the other hand, if a student clearly likes designing algorithms, it ought to be relatively easy to learn what specific requirements are always posed for reductions in the context of proving NP-completeness. By learning that there are such requirements and by learning them, in a context free from details about the involved problems, the student could later feel more comfortable in designing reductions.

In this study we have mainly addressed the students' motivation and the implications of reductions in NP-completeness proofs. The assignment type in Figure 2, which was not considered especially difficult on the exam, shows that this was maybe not that hard after all. The fact that had been at the core of many difficulties was not difficult in itself, after getting proper attention in teaching. The disposition to reduce abstraction mentioned by [Armoni 2008; Armoni et al. 2006] also could not affect the students' thinking here, since nothing is known about the problems involved. It is still unknown if this specific difficulty also disappears in more complicated tasks.

We found that the students liked the new activities and that most of them thought that the activities helped them to learn computational complexity. It is hard to prove that more students now will pass the exam, but the statistics indicate that the students who attended most of the activities improved their grade by doing this.

Finally, we would like to emphasize that our cooperative course development model, where we exchanged activities and discussed problems and solutions over university and country borders, was very successful and satisfying.

7.4. Dynamic programming

The students in 2012 did not express that the recursion-related parts of dynamic programming were particularly difficult, as suspected based on prior work on younger students. It is likely that third year students have moved on to another level of understanding of recursion, and therefore do not find it difficult as is. We have, on the other hand, a suspicion that the step of a problem solving exercise where students need to come up with a recurrence relation, still can be difficult. Without seeing that the problem has this structure, there can be no dynamic programming. It remains clear that when dealing with multi-dimensional dynprog problems, students often forget about some dimension and produce incorrect solutions (which they, on the other hand, often can modify so that they work during oral presentations of homework.)

7.5. Proof and pseudo code

These concepts have not been explicit course content. Instead, they are interweaved in everything else in the course. Students seemed to feel a tiny bit more confident in these areas in 2013, when we dedicated more attention to them, than in 2012, but only marginally so.

Looking at the initial values on proofs, we see that students in 2013 increased their reported self-efficacy quite a lot, from a low level. The ADC students are continuously exposed to the proof practice of TCS – as presented in teaching situations – and hopefully gradually grow into the discipline.

As for pseudo code, the increase is very moderate, but regarding the tasks of presenting an algorithm with pseudo code, the increase is higher. This could be a consequence of the two information texts we produced and the more frequent mentioning of pseudo code during lectures, but also it could be related to the fact that the self-efficacy surveys have made the students reflect on pseudo code, or to pure chance.

Topics like these are probably best tackled at a educational program level, combining the efforts of various teachers into a coherent picture of what is required.

8. CONCLUSIONS

It has been illustrated here, that students can be very positive towards changes in the "experienced curriculum", and that they can be very happy that new methods are tried. Non-anonymous versions of clicker systems, like the colored response cards we have used, are valuable for both teachers and students to assess the progress of a lecture, and this was acknowledged by students in their responses to the final survey.

When starting to use an automated assessment system, if the previous method was to have a TA testing code, students might have a harder time passing the exercises than before. On the other hand, when introducing automated assessment to a course that previously did not have programming assignments, the completion rate can increase. In any case, the use of an automated tool signals some type of context to students, and they will interpret their tasks accordingly. If experimental tasks, like the complexity lab assignment, are introduced, they might need extra explaining, since they are in disguise.

The particular difficulties we have spotted in the ADC course concerns computational complexity, dynamic programming, and proof and pseudocode production (maybe the very presentation of all other tasks in the course). Computational complexity is a mathematical topic, and it appearing side by side with “ordinary” problem solving tasks could confuse students. The mathematical language and context need explaining.

When looking at computational complexity, choosing a problem to reduce, and prove correctness are difficult tasks. Also performing reductions between problem types are not perceived as equally easy as other parts of the content. The same difficulty arises with dynamic programming exercises, together with the difficulty of imagining all dimensions that can vary and all parameters that are needed, in the case where a 2-dimensional matrix is not sufficient to hold all calculated values. When it comes to proving, the necessity of proving sometimes seems to be questioned by students. It is likely the explaining aspect of the proof that they are not acquainted with, other than for teaching purposes. The convincing aspect of a proof is maybe closer to everyday experience, but on the other hand some students are convincing themselves of the truth of a statement (or, more often, of the correctness of their algorithm) based on experimentation and heuristics.

Given that these seem to be the most difficult aspects of the ADC course, it would be interesting to interview students about what a correct program is, and how you know that, and what a correct algorithm is with the corresponding methods for establishing truth. The CS branches occupied with each of these tasks use different methods and assumptions, and the students meet with both these perspectives during studies. They may need some help to structure these impressions.

More information on how students experience the purpose and the quality criteria of typical ADC tasks could also help shed light on what *is*, really, difficult among the included topics.

ACKNOWLEDGMENTS

We would like to thank Monika Lundell and Anna Björklund who kindly have provided us with student documentation and admission data.

References

ACM/IEEE-CS Joint Task Force on Computing Curricula. 2013. *Computer Science Curricula 2013*. Technical Report. ACM Press and IEEE Computer Society Press. DOI:http://dx.doi.org/10.1145/2534860

Michal Armoni. 2008. Reductive thinking in a quantitative perspective: the case of the algorithm course. In *ITiCSE ’08: Proceedings of the 13th annual conference on Innovation and technology in computer science education*. ACM, New York, NY, USA, 53–57. DOI:http://dx.doi.org/10.1145/1384271.1384288

Michal Armoni, Judith Gal-Ezer, and Orit Hazzan. 2006. Reductive thinking in undergraduate CS courses. In *ITICSE ’06: Proc. 11th ann. SIGCSE conf. on Innovation and technology in Comp. Sci. education*. ACM, New York, NY, USA, 133–137. DOI:http://dx.doi.org/10.1145/1140124.1140161

Petek Askar and David Davenport. 2009. An Investigation of Factors Related to Self-Efficacy for Java Programming Among Engineering Students. *Turkish Online Journal of Educational Technology* 8 (2009), 26–32.

Nicolas Balacheff. 1988. Aspects of proof in pupils' practice of school mathematics. *Mathematics, teachers and children* (1988), 216–235.

Albert Bandura. 1986. *Social foundations of thought and action: A social cognitive theory.* Prentice-Hall, Englewood Cliffs, New Jersey.

Albert Bandura. 2006. *Self-Efficacy Beliefs of Adolescents*. Information Age Publishing, Chapter 14: Guide for constructing self-efficacy scales, 307–337.

Benjamin S. Bloom, Max D. Engelhart, Edward J. Furst, Walker H. Hill, and David R. Kratwohl. 1956. *Taxonomy of educational objectives Handbook 1: cognitive domain*. Longman Group Ltd., London.

Markus Andreas Brändle. 2006. *GraphBench: Exploring the Limits of Complexity with Educational Software*. Ph.D. Dissertation. Swiss Federal Institute of Technology.

Jane E Caldwell. 2007. Clickers in the large classroom: current research and best-practice tips. *CBE Life Sci Educ* 6, 1 (2007), 9–20. DOI:http://dx.doi.org/10.1187/cbe.06-12-0205

Daniel Chazan. 1993. High School Geometry Students' Justification for Their Views of Empirical Evidence and Mathematical Proof. *Educational Studies in Mathematics* 24, 4 (1993), pp. 359–387. http://www.jstor.org/stable/3482650

Pierluigi Crescenzi. 2010. Using AVs to Explain NP-completeness. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '10)*. ACM, New York, NY, USA, 299–299. DOI:http://dx.doi.org/10.1145/1822090.1822175

Pierluigi Crescenzi, Emma Enström, and Viggo Kann. 2013. From Theory to Practice: NP-completeness for Every CS Student. In *ITiCSE '13: Proceedings of the eighteenth annual conference on Innovation and technology in computer science education*. ACM, New York, NY, USA, 16–21. http://doi.acm.org/10.1145/2462476.246558

Douglas Duncan. 2006. Clickers: A new teaching aid with exceptional promise. *Astronomy Education Review* 5, 1 (2006), 70–88.

Emma Enström. 2013. Dynamic programming – structure, difficulties and teaching. In *2013 Frontiers in Education Conference (FIE 2013)*. Oklahoma City, USA.

Emma Enström and Viggo Kann. 2010. Computer lab work on theory. In *ITiCSE '10: Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*. ACM, New York, NY, USA, 93–97. http://doi.acm.org/10.1145/1822090.1822118

Emma Enström, Gunnar Kreitz, Fredrik Niemelä, Pehr Söderman, and Viggo Kann. 2011. Five Years with Kattis – Using an Automated Assessment System in Teaching. In *2011 Frontiers in Education Conference (FIE 2013)*. Rapid City, USA.

Efraim Fischbein. 1982. Intuition and proof. *For the learning of mathematics* (1982), 9–24.

Gary Ford. 1982. A framework for teaching recursion. *SIGCSE Bull.* 14, 2 (June 1982), 32–39. DOI:http://dx.doi.org/10.1145/989314.989320

David Ginat and Eyal Shifroni. 1999. Teaching recursion in a procedural environment–how much should we emphasize the computing model?. In *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education (SIGCSE '99)*. ACM, New York, NY, USA, 127–131. DOI:http://dx.doi.org/10.1145/299649.299718

Oded Goldreich. 2006. On Teaching the Basics of Complexity Theory. In *Theoretical Computer Science*, Oded Goldreich, ArnoldL. Rosenberg, and AlanL. Selman (Eds.). Lecture Notes in Computer Science, Vol. 3895. Springer Berlin Heidelberg, 348–374. DOI:http://dx.doi.org/10.1007/11685654_15

Gustav Grundin. 2011. Personal communication. (2011).

Bruria Haberman and Haim Averbuch. 2002. The case of base cases: why are they so difficult to recognize? Student difficulties with recursion. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education (ITiCSE '02)*. ACM, New York, NY, USA, 84–88. DOI:http://dx.doi.org/10.1145/544414.544441

Bruria Haberman and Orna Muller. 2008. Teaching abstraction to novices: Pattern-based and ADT-based problem-solving processes. In *2008 Frontiers in Education Conference (FIE 2008)*. DOI:http://dx.doi.org/10.1109/FIE.2008.4720415

Paola Iannone and Matthew Inglis. 2010. Self efficacy and mathematical proof: are undergraduate students good at assessing their own proof production ability?. In *Proceedings of the 13th Conference on Research in Undergraduate Mathematics Education*. https://ueaeprints.uea.ac.uk/16104/ February 2010.

Keith Jones. 2000. The student experience of mathematical proof at university level. *International Journal of Mathematical Education in Science and Technology* 31, 1 (2000), 53–60. DOI:http://dx.doi.org/10.1080/002073900287381

Hank Kahney. 1983. What do novice programmers know about recursion. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI ’83)*. ACM, New York, NY, USA, 235–239. DOI:http://dx.doi.org/10.1145/800045.801618

Hank Kahney and Marc Eisenstadt. 1982. Programmers’ mental models of their programming tasks: The interaction of real-world knowledge and programming knowledge. In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, Vol. 4. 143–145.

Jon Kleinberg and Éva Tardos. 2006. *Algorithm Design*. Pearson Education. http://books.google.se/books?id=25p3mHu3ij8C

Yi-Yin Ko and Eric Knuth. 2009. Undergraduate mathematics majors’ writing performance producing proofs and counterexamples about continuous functions. *The Journal of Mathematical Behavior* 28, 1 (2009), 68 – 77. DOI:http://dx.doi.org/10.1016/j.jmathb.2009.04.005

G. Light, R. Cox, and S. Calkins. 2009. *Learning and Teaching in Higher Education: The Reflective Professional*. SAGE Publications. http://books.google.se/books?id=N9uqqXGujJsC

Andrea F. Lobo and Ganesh R. Baliga. 2006. NP-completeness for All Computer Science Undergraduates: A Novel Project-based Curriculum. *J. Comput. Sci. Coll.* 21, 6 (June 2006), 53–63. http://dl.acm.org/citation.cfm?id=1127442.1127450

Dino Mandrioli. 1982. On Teaching Theoretical Foundations of Computer Science. *SIGACT News* 14, 3 (July 1982), 36–53. DOI:http://dx.doi.org/10.1145/990511.990516

Ference Marton and Shirley Booth. 1997. *Learning and Awareness (Educational Psychology Series)*. Routledge.

Orna Muller. 2005. Pattern oriented instruction and the enhancement of analogical reasoning. In *Proceedings of the first international workshop on Computing education research (ICER ’05)*. ACM, New York, NY, USA, 57–67. DOI:http://dx.doi.org/10.1145/1089786.1089792

Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-oriented instruction and its influence on problem decomposition and solution construction. *SIGCSE Bull.* 39, 3 (June 2007), 151–155. DOI:http://dx.doi.org/10.1145/1269900.1268830

Orna Muller and Bruria Haberman. 2008. Supporting abstraction processes in problem solving through pattern-oriented instruction. *Computer Science Education* 18, 3 (2008), 187–212. DOI:http://dx.doi.org/10.1080/08993400802332548

Orna Muller and Amir Rubinstein. 2011. Work in progress; Courses dedicated to the development of logical and algorithmic thinking. In *2011 Frontiers in Education Conference (FIE 2011)*. DOI:http://dx.doi.org/10.1109/FIE.2011.6142846

Frank Pajares and M. David Miller. 1994. Role of Self-Efficacy and Self-Concept Beliefs in Mathematical Problem Solving: A Path Analysis. *Journal of Educational Psychology* 86, 2 (1994), 193–203. http://www.eric.ed.gov/ERICWebPortal/detail?accno=EJ490260

Christos H Papadimitriou. 1997. NP-completeness: A retrospective. In *Automata, languages and programming*. Springer, 2–6.

Christian Pape. 1998. Using Interactive Visualization for Teaching the Theory of NP-completeness. In *Proc. ED-MEDIA/ED-TELECOM*. 1070–1075.

Vennila Ramalingan and Susan Wiedenbeck. 1998. Development and Validation of Scores on a Computer Programming Self-Efficacy Scale and Group Analyses of Novice Programmer Self-Efficacy. *Journal of Educational Computing Research* 19, 4 (1998), 367–381.

Justus J. Randolph. 2007. Meta-Analysis of the Research on Response Cards: Effects on Test Achievement, Quiz Achievement, Participation, and Off-Task Behavior. *Journal of Positive Behavior Interventions* 9, 2 (April 2007), 113–128. DOI:http://dx.doi.org/doi:10.1177/10983007070090020201

Tamarisk Lurlyn Scholtz and Ian Sanders. 2010. Mental models of recursion: investigating students’ understanding of recursion. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education (ITiCSE ’10)*. ACM, New York, NY, USA, 103–107. DOI:http://dx.doi.org/10.1145/1822090.1822120

Michael Sipser. 2012. *Introduction to the Theory of Computation*. Thomson.

Andreas J. Stylianides and Thabit Al-Murani. 2010. Can a proof and a counterexample coexist? Students’ conceptions about the relationship between proof and refutation. *Research in Mathematics Education* 12, 1 (2010), 21–36. DOI:http://dx.doi.org/10.1080/14794800903569774

Franklyn Turbak, Constance Royden, Jennifer Stephan, and Jean Herbst. 1999. Teaching Recursion Before Loops In Cs1. (1999).

Thomas Varghese. 2011. Possible Student Justification of Proofs. *School Science and Mathematics* 111, 8 (2011), 409–415. DOI:http://dx.doi.org/10.1111/j.1949-8594.2011.00106.x

Jessica M. Zerr and Ryan J. Zerr. 2011. Learning from Their Mistakes: Using Students' Incorrect Proofs as a Pedagogical Tool. *PRIMUS* 21, 6 (2011), 530–544. DOI:http://dx.doi.org/10.1080/10511970903386915

Received October 2014; revised ; accepted