Olympiads in Informatics, 2017, Vol. 11, 87–92

DOI: 10.15388/ioi.2017.07

A Competitive Programming Approach to a University Introductory Algorithms Course

Antti LAAKSONEN
Department of Computer Science
University of Helsinki
e-mail: ahslaaks@cs.helsinki.fi

Abstract. This paper is based on our experiences on teaching a university introductory algorithms course using ideas from competitive programming. The problems are solved using a real programming language and automatically tested using a set of test cases. Like in programming contests, there are no hints and well-known problems are not used. The purpose of such problems, compared to traditional problems, is to better improve the problem solving skills of the students.

Keywords: algorithms course, competitive programming.

1. Introduction

This paper summarizes our experiences on teaching the *Data Structures and Algorithms* course at the University of Helsinki using ideas from competitive programming. The course deals with basic data structures and algorithms, such as trees, graphs and sorting. The course is a compulsory course for computer science students, and it is usually taken during the first year of studies. The course book is *Introduction to Algorithms* (Cormen *et al*., 2009), though the course covers only a part of the book.

For some years ago, the course was purely theoretic and algorithm design problems were solved using pen and paper and discussed in exercise sessions. However, it was observed that the learning results were not good and many students had difficulties in designing even very simple algorithms. After this, some ideas from competitive programming have been used on the course to improve the student's problem solving skills.

The *competitive programming approach* is based on the following ideas:

- The algorithm design problems are presented without hints and the solutions cannot be easily found using search engines.
- The solutions are implemented using a real programming language and automatically graded using a set of test cases.
- Solutions are given points only if they work correctly and efficiently, and techniques for testing and debugging are presented.

The above ideas have been used in programming contests for a long time, but we believe that they have potential to be used much more widely also on university algorithm courses. This does not mean that the courses should be *competitive*: only the problem types and the way the solutions are evaluated resemble the practices used in programming contests.

Of course, it is not a new idea to use automatic program evaluation in a university course. For example, García-Mateos *et al*. (2009) describe a programming course where the final exam is replaced with a series of programming contests using the Mooshak system, and Enström *et al*. (2011) present their experiences after using the Kattis system in several algorithm courses.

In this paper, we focus on the features of competitive programming style problems. First, we describe the way the competitive programming approach has been used on our course. After this, we discuss some of the advantages and disadvantages of the approach.

2. Course Organization

The course consists of 12 weeks. Every week there are two problem sets: a set of algorithm design problems and a set of other problems. The algorithm design problems follow the competitive programming approach: they are submitted in an online course system and evaluated automatically. The other problems include simulation problems and mathematical problems.

The competitive programming approach was introduced in the course in 2011. Initially, the DOMjudge system[1] was used to evaluate the solutions. However, this system was not optimal for the course, because it is intended for ICPC style contests and only shows if a solution is correct or not.

Since 2012, the TMC system (Pärtel *et al*., 2013) has been used on the course. This system is also used in the introductory programming courses at the University of Helsinki. TMC allows students to create their Java solutions in the NetBeans IDE and evaluates the solutions using JUnit tests. The tests are written by the course staff and can be downloaded using the TMC plugin.

The algorithm design problems are renewed every now and then so that the students do not get too familiar with them. For example, one of the first problems in the fall 2014 iteration of the course was as follows:

> Given a string, your task is to create a palindrome by removing exactly one letter from the string. For example, the string `ABCBXA` can be turned into a palindrome by removing the letter `X`.
>
> Your task is to implement the following method:
>
> `boolean almostPalindrome(String s)`
>
> The parameter `s` is a string that consists of at most 10^5 letters. The method should return `true`, if it is possible to create a palindrome by removing exactly

[1] `https://www.domjudge.org/`

one letter, and `false` otherwise.

Time limit: 2 seconds.

The above format is used in all algorithm design problems. First there is a short problem statement, then a template for a Java method and an explanation what the method should do. In each problem, there is also a certain time limit for a single test case.

To get points for the problem, the student has to implement a method that corresponds to the problem statement and works efficiently in all test cases. An important detail in the above problem statement is that the string can be quite long (10^5 letters). Thus, the intended solution should work in $O(n)$ or $O(n \log n)$ time.

The benefit in using JUnit tests is that if the algorithm does not work correctly, the tests can give a friendly message to the student. For example, in the above problem, a message could be "The string `AAABAA` can be turned into a palindrome, but your method returned `false`." Depending on the problem configuration, the tests can be either fully or partially public.

3. Discussion

3.1. *Improving Problem Solving Skills*

Algorithm design problems are difficult, and it requires a great deal of work to improve one's problem solving skills. As algorithm design is difficult, it is a common practice to include *hints* in problems. For example, consider the following problem (Cormen *et al*., 2009, page 42):

> Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint:* Modify merge sort.)

It is definitely easier to solve the above problem using the hint. However, at the same time, the hint almost completely spoils the problem, and after reading the hint, there is not much problem solving needed.

Such hints are never seen in competitive programming and there is a good reason for it: it is an important step in problem solving to consider different ideas how to approach the problem before finally finding a way to solve it. Using hints it may be possible to solve problems quickly, but this does not improve one's problem solving skills.

Still, it is clear that the above problem is difficult and without the hint, the problem would be impossible to solve for many students who are beginners in algorithm design. However, we do not think that presenting a hint is a good way to overcome this. If a problem is too difficult, it should be presented later when the students really can solve it.

Thus, the challenge is to find problems that require problem solving but are not too difficult. Fortunately, the easiest problems in many programming contests have those properties. It is better to solve an easy problem using one's own skills than to solve a difficult problem using hints.

3.2. *Originality of Problems*

Another benefit in competitive programming problems is that they are – or should be – *original*. This ensures that it is not too easy to find solutions to problems just by using search engines. For example, consider the following problem (Cormen *et al.*, 2009, page 602):

> The *diameter* of a tree $T = (V, E)$ is defined as $\max_{u,v \in V} \delta(u, v)$, that is, the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

A simple Google search (Fig. 1) can be used to find complete tutorials how to solve the problem. Of course, nobody is forced to use search engines to solve problems. However, in practice, many students do this, and this is very harmful to their problem solving skills. Problems like the tree diameter problem are good *examples* how to design algorithms, but they should not be used as course problems, because they are too well-known problems.

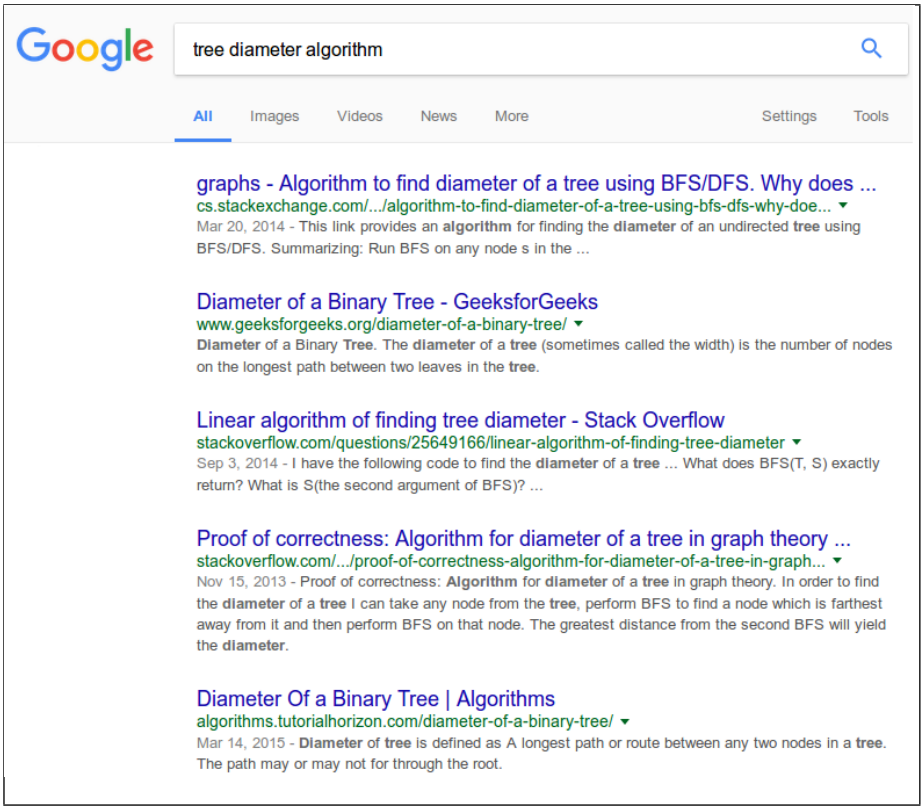

Fig. 1. Solving the tree diameter problem using Google.

3.3. *Focusing on Correct and Efficient Algorithms*

The important question after designing an algorithm is: does the algorithm really work? One way to answer this question is to give a *proof* which shows that the algorithm works. However, it is not realistic to expect such proofs in an introductory algorithms course. Just giving a verbal description or a pseudocode of the algorithm is not satisfactory either, because it is easy to make wrong assumptions or skip important details.

In the competitive programming approach, all algorithms are *implemented* using a real programming language and the implementations are *tested* using a comprehensive set of test cases. This has two important benefits: the students have to precisely describe how their algorithms work, and after that, they will see if their algorithms are correct or not.

When learning to design algorithms, it is not only important to find correct algorithms to problems but also see why certain approaches do not work. In particular, it is easy to sketch intuitive greedy solutions to many problems, but such solutions often do not work in reality.

A side effect of the competitive programming approach is that it also improves programming, debugging and testing skills of the students. In competitive programming, a common way to find a bug in an algorithm is to use *stress testing*, which involves generating a large set of random test cases and checking if a brute force algorithm and an efficient algorithm always agree with each other. Many students are not familiar with this kind of comprehensive testing, even if they have attended courses on software engineering.

Another important factor is the efficiency of algorithms. By implementing and testing algorithms it is possible to see how efficient they are in reality and what is the connection between time complexities and real running times of algorithms. Moreover, it becomes evident which kind of optimizations are important. Typically, during the first weeks, the students try to improve their solutions using micro-optimizations without success, and later understand the importance of time complexities.

The responsibility of the course staff is to create challenging sets of test cases that ensure that accepted solutions are correct and efficient. This is sometimes difficult: for example, optimized brute force solutions can be surprisingly efficient. Still, if an algorithm is worth learning, it should be possible to find a test case where it works better than a brute force algorithm.

3.4. *Limits of Competitive Programming*

It is clear that some features of algorithms cannot be automatically tested using the competitive programming approach. For example, consider the following problem (Cormen *et al.*, 2009, p. 223):

> Describe an $O(n)$-time algorithm that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S.

It is easy to solve the problem in $O(n \log n)$ time: just sort the array and take the k middle elements. Designing an $O(n)$ algorithm is a more difficult task. Unfortunately, it is not possible to automatically test whether the time complexity of an algorithm is $O(n)$ or $O(n \log n)$ by measuring the running time because the difference may be very small. In general, logarithmic factors in time complexities cannot be reliably detected.

In addition, sometimes it may be a problem that it is possible to *guess* a solution to a problem and just test if it works. Of course this also happens in real programming contests. It is difficult to prevent this, but we do not think it is a problem in an introductory course. In fact, even in algorithm research, guessing is a valid way to design an algorithm, though it is required to later prove that the algorithm is correct.

Acknowledgements

The author would like to thank Toni Annala, Matti Luukkainen, Pekka Mikkola and Kristiina Paloheimo for useful discussions.

References

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

Enström, E., Kreitz, G., Niemelä, F., Söderman, P., Kann, V. (2011). Five years with Kattis – using an automated assessment system in teaching. *IEEE Frontiers in Education Conference*.

García-Mateos, G., Fernández-Alemán, J.L. (2009). A course on algorithms and data structures using on-line judging. *ACM SIGCSE Bulletin*, 41(3), 45–49.

Pärtel, M., Luukkainen, M., Vihavainen, A., Vikberg, T. (2013). Test my code. *International Journal of Technology Enhanced Learning*, 5(3–4), 271–283.

A. Laaksonen received his PhD in Computer Science from the University of Helsinki. He is one of the organizers of the Finnish Olympiad in Informatics, and a coach and a team leader of Finnish BOI and IOI teams. He is the author of the book *Competitive Programmer's Handbook* and one of the developers of the CSES contest system.