Analysis of Algorithms: Programming To Problem Solving

Karina V. Assiter
Computer Science and Systems Department, Wentworth Institute of Technology
Boston, MA 02115 assiterk@wit.edu

Abstract - In Introduction to Analysis of Algorithms students’ traditionally apply a combination of computer science theory and mathematics to paper-based problem solving, analysis of pre-developed algorithms and proofs of algorithmic run-times. In this paper, we suggest that that a major factor that determines success in Analysis is the discrepancy between the programming styles of CS1 and CS2, with immediate feedback of student solutions, a dynamic working environment and non-sequential implementation, and the learning requirements of a theory-based course, with delayed feedback of student solutions, a static working environment and sequential problem solving. Since they are not (as yet) supported by empirical evidence, these discussions do not lead to definitive claims about analysis and student performance; on the other hand, they do generate theories for research that could enhance the undergraduate experience in theory based courses.

Index Terms – Algorithms, Computer Science, Learning Styles, Programming, Theory.

INTRODUCTION

In the introductory weeks of an Analysis of Algorithms course students learn to translate algorithmic components into mathematical equations (loops to summations and recursion to recurrence relations) and solve the equations so that the original algorithms are assigned to efficiency classes. Analysis problems in algorithms’ texts [1, 2, 3] generally start with a complete algorithm or program; writing the program or algorithm is not part of the solution.

In CS1 and CS2 students had been learning to program and then in Analysis they must solve mathematical equations. We would like to determine the factors that make analysis challenging for undergraduate computer science students as they make this transition from programming to theory.

We first present assessment results that suggest there is a decline in student performance in the first weeks of the Introduction to Analysis of Algorithms course. We then compare the teaching methods and learning requirements of Analysis with the environment and requirements of CS1 and CS2. Once we identify the dimensions in which there are differences, we propose one tool that could supplement teaching methods in Analysis. In conclusions and future work we summarize our subjective and objective experiences and discuss next steps for research.

MOTIVATION

This research was initiated after there was a high occurrence of late homework submissions, incomplete and/or incorrect problem sets and low exam scores in the first half of an Introduction to Analysis of Algorithms course. Most significantly, students who claimed to be strong in CS1 and CS2 were as likely to exhibit these indications of failure as weaker students.

We consider here the factors that might predict success in the Algorithms course: success in CS1 and CS2, student initiative, strength of instructor, clarity of the textbook, adequate course pre-requisites, and differences between learning style and course/content presentation style. Since we observed failure across the board (exhibited by both strong and weaker students) we ruled out student initiative as a significant factor. Also, student evaluations did not indicate a consensus of opinion on either instructor performance or textbook clarity. In terms of the adequacy of pre-requisites as a factor, students in Algorithms would have had those recommended by Curriculum 2001[4]. Since other factors do not seem to significantly influence student performance, we focus on how differences between learning styles and course/content presentation style affect student performance.

In the next section, we compare a theory of the programmer style that may have developed in CS1 and CS2 with the predominant teaching methods in the Introduction to Analysis of Algorithms course.

PROGRAMMING STYLE

Learning styles are “characteristic cognitive, affective, and psychological behaviors that serve as relatively stable indicators of how learners perceive, interact with, and respond to the learning environment [5]. The Felder-Silverman model [6] classifies students as having preferences in each of the following dimensions [7]:

- Perception Preferences:
  - Sensory: concrete, practical, facts and procedures
  - Intuitive: Conceptual, innovative, theories and meanings.

- Sensory input preferences:
  - Visual: Prefer pictures, diagrams, flow-charts.
  - Verbal: Prefer written or spoken explanations

0-7803-9077-6/05/$20.00 © 2005 IEEE.
35th ASEE/IEEE Frontiers in Education Conference
October 19-22, 2005, Indianapolis, IN
Session S1H
S1H-7
There are other types of learning style models [8, 9, 10, 11] that we could include, but Felder and Silverman point out that “each of the stated dimensions has parallels in other learning style models”.

Learning Styles in Engineering Education

As Felder points out in [12], “most engineering education has been heavily biased toward intuitive, verbal, deductive, reflective and sequential learners” while “relatively few engineering students fall into all of these five categories”. The objective, then, of any curriculum should be to teach to the “full spectrum of learning styles”. We shall consider this as we look at performance more specifically as it applies to computer science education.

Performance in the Introductory Programming Sequence

Research on performance in computer science education has focused, primarily, on the early programming courses of CS1 and CS2. Thomas et al. [13] found that the combination of reflective/sensing/verbal/sequential learners outperformed other combinations; probably, as they conclude, because software engineering education “favors students with certain kinds of learning style preferences”. If we make a parallel to studies in engineering education, we can theorize that few software engineering students fall into all four of these categories.

Instead of determining correlations between learning style and performance in CS2, Howard et al [14] argue that by combining techniques, tools, methods and models (i.e., the Minute Paper tool and the Constructive Learning model) we can both reach all levels of Bloom’s Taxonomy of Learning and accommodate all learner types.

Wilson and Shrock [15] looked at twelve factors that might contribute to success in CS1, including work style preference, previous programming experience, math background and encouragement, and found that comfort level and math preparation were the best predictors of success in CS1.

Goold and Rimmer [16] observe that different learning styles excel for different tasks within a discipline or even within a single course. This implies that success in computer science is not limited to students with specific combinations of learning styles.

Chamlillard & Karolick [17] had students and faculty in an introductory course use learning style results to improve study habits and instruction techniques, respectively. This is in agreement with Felder and Spurlin’s [7] suggestion that learning style results be used to: 1) help instructors design instruction that addresses the learning needs of all students, and 2) give students insight into their learning strengths and weaknesses. In other words, learning style results should not be used to prevent students with non-dominant learning styles from entering the discipline.

At Wentworth there is a lab component for both CS1 and CS2. In these labs, students are guided through exercises that put into practice concepts learned in lecture. This suggests that students with a preference for active learning would fare better than they might have without a lab requirement in CS1 and CS2.

Performance in Introduction to Analysis of Algorithms

Our research focuses on student performance in Introduction to Analysis of Algorithms, after students have completed the CS1 and CS2 sequence. We incorporate the Felder-Silverman learning styles into a discussion of students’ experience in Analysis. We have added dimensions, Interaction in the problem solving domain, Mathematical Thinking and Timing of Feedback, that reflect learning methods of CS1 and CS2 (or that are work style preferences of students interested in programming), and substituted the term programming style for learning style. The following sections compare programmer style with the instruction methods applied in Analysis.

Type of Information Preferentially Perceived

In Software Engineering, complex real-world problems are translated to models (UML or algorithmic) which are then translated into programming language constructs. In the case of novice programmers in CS1 and CS2, the step of developing a model is simplified into personal notations and code/algorithm combinations. This method of modeling is not ideal but it works for student problems with low complexity.

Research suggests that procedural programming, where real-world objects cannot be directly translated into program constructs, favors intuitive learners, while object oriented programming, where “objects are simulated view of real world objects”, favors sensing learners [18]. Thus, students in CS1 and CS2 oriented toward concrete learning, preferring facts and procedures over conceptual abstractions may still be successful in CS1 and CS2.

In analysis, problem solving begins one step removed from a real-world problem with a pre-written algorithm. The algorithm might even be an abstract component (selection sort) used to solve many types of real-world problems (i.e., sorting a list of names). The student must translate programming language constructs into mathematical constructs that represent run-time components. These mathematical constructs are combined into expressions of efficiency which are either solved or simplified to an efficiency class.
Analysis will develop students’ ability to think in conceptual abstractions even if they prefer to learn through concrete facts and procedures. However, we could supplement traditional instruction in Analysis with methods that address the needs of the sensing learner.

One way to connect abstract concepts with concrete problems in analysis is to have student teams implement variations of well-known algorithms (same behavior but attempting to improve efficiency). Results could include the implementation and an analysis of the algorithms’ run-time. Teams would be judged on correctness of analysis and efficiency of solution.

Another approach would be to integrate algorithm visualization into the both the lecture and lab components of the course. In their paper, Naps et al [19] suggest methods for: 1) integrating visualization techniques into classroom instruction, and 2) evaluating learning outcomes when the techniques are applied. As for other authors, their goal in developing instructional techniques is to have students “extend their skills of adopting and processing information within all four” learning style continuums.

Type of Sensory Information Effectively Perceived

Felder in [20] suggested that visualizations alone are more helpful to visual learners than non-visual learners. In their study on algorithm visualization (AV), Grissom et al [21] found that learning increased as level of interaction (with a visualization system) increased. For example, students who responded to questions integrated into an AV showed more improvement than students who either simply viewed a visualization or who were not shown a visualization at all. This implies that visualizations can assist visual learners in understanding verbal material. The authors also point out that they developed a substantial amount of material (i.e., lectures notes and printed study guides) to accompany their visualizations.

As many authors have concluded, these results suggest that presenting instruction in all of Felder’s Learning Styles improves the effectiveness of teaching [14].

Student Preferences for Processing Information

There appear to be a balance of active and reflective learners in computer science courses at Wentworth. For example, when given the option, half of all students in a class select to work in pairs on programming assignments (or on in class exercise).

Programming can be an active or a reflective process depending on the preferred working style of the student. Active learners might develop applications through repeated cycles of planning, implementation and testing. Extreme Programming [22] is an example of teaching geared toward the active learner. In contrast, the reflective learner might design and model a complete solution on paper before they implement it on the computer.

In terms of research results, the authors in [17] found that reflective learners were not as successful in CS1 as active learners, and Grant [18] found that the combination of active and abstract learners had the highest performance in CS1.

If students work together on problem sets then their similar solutions could be taken as proof of copying/cheating. Though it is the preferred working style of the reflective learner working individually on hand written solutions may be a source of frustration for the active learner. Just as students with non-dominant learning styles in CS1 and CS2 may leave the computer science program, students in analysis who prefer active learning may believe that they are weak in theoretical computer science, and may reconsider plans for graduate school.

Mathematical Thinking

Numerous papers have considered the importance of mathematical preparation for computer science student [23 - 29]. This is evident in analysis where problem sets resemble problems sets in mathematics textbooks, where students develop hand written solutions to numerically based mathematical equations. Students in analysis are applying mathematical thinking to problem solving according to the definition from the Mathematical Thinking working group [23] of, “applying mathematical techniques, concepts and processes, either explicitly or implicitly, in the solution of a problem”. An observation from these authors is that students who use mathematics in the form of formal methods (symbols and notation, logical precise reasoning, using patterns, problem analysis, modeling, abstraction, generalization, understanding software) develop better software than those who do not. This implies we should improve the development of mathematical thinking in CS1 and CS2; a side effect of which would be that students would be better prepared for mathematical thinking in analysis.

Student Preferences for Progressing Toward Understanding

Software development is not a strictly sequential process; problem solutions can be developed by defining functionality at a high level (testing using stubs) and filling in details of behavior as components are created (top-down), or by defining building block components (testing using drivers) and combining the components for higher levels (bottom-up). In all cases, individual components can be developed and tested in any order before they are combined into a final solution. This suggests that both sequential and global learners can excel in software development.

In contrast, analytic problem solving starts with the problem description and progresses one step at a time toward a solution. The drawback with a sequential process is that when errors are introduced in one step they exist in all subsequent steps. Additionally, computer science students who are global thinkers may have trouble with the sequential constraint of analysis.
Timing of Feedback

When students in CS1 and CS2 implement their solutions in a programming language, they receive immediate feedback about code correctness from the language compiler or translator. Students, therefore, have a good idea of their programs’ correctness before they submit their work to the instructor for formal evaluation. Since assignments are not graded on intermediate (possibly incorrect) solutions, students can experiment and, possibly, enjoy the process of programming/problem solving (i.e., have fun).

When the format of an Analysis course is similar to that of a math class (as at Wentworth), once student solutions are submitted to the instructor feedback is not available until at least the next class period. If an answer is incorrect it can’t be fixed by the student until after the work is graded.

When we graded on effort rather than correctness (and distributed solution sets with returned work) there was an increase in the number of assignments submitted on time. Unfortunately, students who submitted late assignments could have copied from the answer set even to receive an effort grade. We could have, instead, relied on exam grades to assess student performance. Unfortunately, this would have unfairly punished students who do not perform well on timed exams, but who submit excellent homework solutions.

As an alternative, we plan to supplement traditional instruction with a tool that would automate the steps of analysis so that solutions can be verified before they are submitted for evaluation. An initial design of this environment is presented in Interactive Online Environment for Analysis.

Level of Interaction in Problem Solving Domain

Because it is relatively easy for students to make changes to their code, we describe programming as a dynamic process. For example, when a student changes a section of their program, they will not, generally, have to update all subsequent lines. On the other hand, we consider problem solving in analysis a static process; students write solutions on paper by hand or use a word processor (where mathematical symbol insertions are not well supported). When an analysis solution step is updated, all dependent steps have to also be changed.

To make analysis a dynamic process, we could supplement the traditional static methods with an online, interactive environment. The environment could include an equation and efficiency class verifier as well as a smart equation builder. Student solutions could be entered and verified a step at a time so that errors do not trickle down to dependent steps. All interaction between student and environment could be logged and printed at the end of the session; either in verbose mode or in final solution mode. Thus, the emphasis of Analysis will be on problem solving and student understanding and not just on the final solution.

Significance of Comparisons

Our research goal is to improve student performance in Analysis. We have highlighted areas in which there may be differences between the programming style preferences of students entering from CS2 and the learning requirements in Analysis.

We do not imply that computer science students in Analysis should not have to think abstractly, solve sequential problems or be prepared for delayed feedback. On the other hand, we are initiating a formal study to determine whether our observations can be validated across various institutions/programs and over several years. Once the study is complete, we will develop appropriate supplemental materials to ease the transition from programming to theoretical computer science.

Unfortunately, methods that have been implemented (grading homework on effort, having open book/open notes exams, etc.) have addressed the symptoms and not the causes for computer science students’ difficulty in analysis.

We now plan to supplement traditional instruction in Analysis with techniques that provide immediate feedback to students about solution correctness and that allow dynamic non-sequential problem solving.

In the next section we summarize a proposed interactive environment for interpreting and verifying analysis solutions.

Interactive Environment for Analysis

In this section, we propose an interactive environment (FIGURE 1) that could supplement traditional instruction for the student learning analysis. The environment should provide immediate feedback to students about solution correctness and allow independent, non-sequential solution development.

Since analysis is a two step process with 1) an algorithm translated to a mathematical expression and 2) expression solved to an efficiency class, the environment should have separate components for equation verification and efficiency class verification.

Equation Verification

The equation verifier could take an algorithm as input and compile it to generate run-time translations. Then a verifying interpreter could request the student’s run-time contribution entry for each algorithmic component (i.e., nested loops, additions, multiplications, assignments, etc.) and compare it to the correct run-time. The output for each entry could include correctness and reason(s) for invalid run-time contributions. The objective would be to provide feedback to students without revealing the solutions; (otherwise students could use the equation verifier as an equation generator).

Efficiency Class Verification

The efficiency class verifier would also be an interpreter. It would take as input the equation from the equation verifier process. At each step, students could select from among a set of translations that could be legally applied as mathematical
transformations for the equation at that step. Steps could not be skipped, but students could replace a set of repeated steps with a macro (frequently performed transformations could be combined, named and executed as a group).

**Design Summary**

The interactive environment will provide immediate feedback about analysis solutions. Since they are conceptually distinct, we divide behavior into the algorithm to equation phase and the equation to efficiency class phase. Because algorithm component interpretation is dependent on scope (i.e., addition inside of a loop) the algorithm is should be pre-compiled before translations are verified. Algorithm compilation requires that students use a common algorithmic language.

**CONCLUSIONS AND FUTURE WORK**

We suggest that students entering from CS2 are familiar with a non-sequential, dynamic problem solving style (where they receive immediate feedback about solution correctness), and this conflicts with the learning requirements of Analysis. We could address this by supplementing traditional instruction in analysis with tools that make the process similar to programming.

**REFERENCES**