Problem Solving and Algorithm Education in Computer Science

Caitlyn Pickens

September 23, 2012

Abstract

Computer scientists are expected to be problem solvers, yet the current structure of algorithm education does not facilitate learning the necessary skills to do so. Learning objectives for algorithm education are defined and Bloom and SOLO taxonomy are employed to evaluate their complexity. Current algorithm curriculum, like that of Michigan State University, is examined and evaluated for effectiveness.

1 Introduction

Computer scientists and programmers are problem solvers. When presented with a specific context and dilemma, they are expected to be able to construct a computing solution to a real-world problem. It follows, then, that problem solving is one of the “big ideas” that should be taught in computer science education. Currently, such coursework is not generally called “problem solving,” but instead “algorithms” – a term that originates from a treatise by widely-read Arabic mathematician al-Khwrizm. The treatise title was Latinized as “Algoritmi on the Numbers of the Indians.” Over time his name evolved into the word “algorithm,” which is now taken to mean the process or set of rules to be followed for problem-solving operations.

This paper will discuss the importance of algorithms, or problem solving, as a big idea in computer science education. The learning objectives for algorithms courses are defined, and the skills successful students should exhibit are discussed. Then, this paper defines the difficulties which arise from algorithm education and discusses the educational theory and statistical evidence that support the definitions. The existing structure for algorithm education at Michigan State University (MSU) is outlined, and the possible shortcomings of the current system are discussed.

2 Defining Algorithm Education: What are the learning objectives?

The Accreditation Board for Engineering and Technology (ABET) requires that by graduation computer science students must attain “an ability to analyze a problem, and identify and define the computing requirements appropriate to its solution” and “an ability to design, implement and evaluate a computer-based system, process, component, or program to meet desired needs” [4]. More specifically, the curriculum requirements indicate that students must have at least one and one-third years of coursework in “fundamentals of algorithms, data structures, software design, concepts of programming languages and computer organization and architecture” [4].

From their emphasis on fundamentals of algorithms, data structures, and software design, it is clear that ABET stresses the importance of problem solving education for CS students.

2.1 Learning Objectives

Taking a closer look at the ABET accreditation requirements helps dissect the components of learning how to solve a problem using a computing solution. Upon graduation from a computer science degree-granting program, students should be able to:

1. analyze a problem
2. identify the computing requirements appropriate to its solution
3. define the computing requirements appropriate to its solution
4. design a program to meet desired needs
5. implement a program to meet desired needs
6. evaluate a program that meets desired needs

All of these learning objectives can be targeted by a simple programming task. For example, CSE 231: Introduction to Programming I at MSU requires students to start writing problem solving programs within the first week of the curriculum [1]. Although these problems and their accompanying programmatic solutions are relatively straightforward and simple, the task of engineering a solution to a given problem still meets all of the learning objectives defined above. Consider the first programming assignment from Fall 2012; note that this project is the first ever exposure to programming (in any form) for the majority of these students in CSE 231.

It is clear then that even the most simple of programming tasks targets the learning objectives outlined by ABET. Starting as early as the very first introductory programming class, students are expected to be developing and employing problem solving skills. As they proceed with their education and reach a course like MSU’s CSE 331: Algorithms and Data Structures (which focuses specifically on problem solving techniques), students are expected to demonstrate these skills at an even more advanced level. The rest of this paper will discuss what algorithms education looks like currently, why it is a cognitively complex subject, and a survey of the problem solving skills of today’s computing students.

3 Algorithms Curriculum: What are algorithms classes teaching?

3.1 Famous Algorithms as Examples

Generally, the second and/or third programming classes in a degree program are dedicated to the study of algorithms and data structures. These courses often teach classic problems faced in computer science and their tried-and-true solutions: the sorting problem and Quicksort, the shortest path problem and Dijkstra’s algorithm, finding a minimum spanning tree and Kruskal’s algorithm, maximum network flow and the Ford-Fulkerson algorithm.

At MSU, the concept of an algorithm is formally defined in CSE 232: Introduction to Programming II. The course objectives of state:

In this course, students will increase their capability to solve larger problems by learning fundamental algorithms and data structures and methods to create reliable programs using them...Students will understand the characteristics of basic methods for update, searching and sorting. [2]

Algorithm (and therefore problem solving) education is further underscored in CSE 331: Algorithms and Data Structures. This class takes a more traditional approach to algorithm education than CSE 232, and teaches the
classic problems and their algorithmic solutions - like insertion sort, quicksort, merge sort, topological sort, shortest path algorithms, depth-first and breadth-first search, and greedy algorithms versus dynamic programming. It also spends two weeks of the course at the end of the semester discussing algorithm design techniques. The learning objectives are defined below:

In this course, students will survey fundamental data structures and many associated algorithms. Study of classical abstract data types (ADT) will be fairly comprehensive. Emphasis will be placed on matching the appropriate data structures and algorithms to application problems. Analysis of algorithms is crucial to making proper selections, so analysis is important in the course. This course assumes that students are already familiar with advanced programming techniques including the definition of classes, and use of dynamic memory and linked data structures, including lists and trees. Even though the treatment of algorithms and data structures is mostly conceptual, students are expected to be able to transform these algorithms and data structures into programs with proper approaches of software module development. [3]

3.2 Complexity Analysis

As alluded to in the course objectives for CSE 331, many algorithms classes also analyze the efficiency of specific algorithms. Time efficiency is referred to as an algorithm’s complexity, and is denoted formally as $\Theta(n)$. An algorithm’s complexity is determined using a pseudocode version of the algorithm - a simple, concise, and general representation that is not dependent on a programming language.

4 Evaluating Algorithm Education: How difficult is algorithm education?

4.1 Bloom’s Taxonomy

Recent research in the past decade has applied the traditional Bloom’s taxonomy to computing curriculum, to evaluate both its effectiveness and its difficulty. Bloom’s taxonomy identifies six major stages of understanding that learning objectives can fall into: (1) Knowledge, (2) Comprehension, (3) Application, (4) Analysis, (5) Synthesis, and (6) Evaluation. Although researchers disagree on where tasks such as code–tracing fall in the spectrum of Bloom’s hierarchy, they all agree that the act of writing a computer program to solve a specific, previously unseen problem falls in (at least) the Synthesis level – the second from the most cognitively complex [10].

More specifically, Anderson et al suggest a revision to Bloom’s taxonomy, which clarifies the stages originally proposed by Bloom [6]:

<table>
<thead>
<tr>
<th>Categories</th>
<th>Cognitive processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remember</td>
<td>Recognizing, Recalling</td>
</tr>
<tr>
<td>Understand</td>
<td>Interpreting, Examining, Classifying, Summarizing, Inferring, Comparing, Explaining</td>
</tr>
<tr>
<td>Apply</td>
<td>Executing, Implementing</td>
</tr>
<tr>
<td>Analyze</td>
<td>Differentiating, Organizing, Attributing</td>
</tr>
<tr>
<td>Evaluate</td>
<td>Checking, Critiquing</td>
</tr>
<tr>
<td>Create</td>
<td>Generating, Planning, Producing</td>
</tr>
</tbody>
</table>

Anderson et al also propose a second dimension to Bloom’s taxonomy of evaluating coursework: defining the type of knowledge elements that exist at each stage of the hierarchy. These are (a) factual knowledge, (b) conceptual knowledge, (c) procedural knowledge, and (d) metacognitive knowledge. Learning objectives (like those from Section 2.1) can then be mapped onto a matrix:
4.2 SOLO Taxonomy

Another popular taxonomy in recent literature is Structure of the Observed Learning Outcome (SOLO). This taxonomy diverges from Bloom’s in it measures the structural complexity of the outcomes of a learning objective, rather than the expected complexity of the cognitive processing necessary to attain a learning objective. The levels of SOLO taxonomy define possible learning outcomes as one of the following:

1. Prestructural: not on-task, are disjoint, or missed the point
2. Unistructural: simple in meaning, focus on one issue in a complex case
3. Multistructural: disorganized collection of items, like a shopping list
4. Relational: understanding, concept integrates data, applying a concept to a familiar context
5. Extended Abstract: relate unseen problems to existing principles, going beyond existing principles [6]

When applied directly to a programming task, such as writing a scheduling program to add students from a waitlist to a course until it reaches capacity, the learning outcomes would look like:

1. Prestructural: inability to write correct code
2. Unistructural: ability to write a small section of correct code; e.g. writing a selection statement to check if a specific course enrollment is at capacity or not
3. Multistructural: ability to complete a few statements of code based on detailed specification of the problem (or provided pseudocode); e.g. writing a method to check if any input course is at capacity, which returns a boolean value
4. Relational: ability to write code to solve a problem that is not presented as pseudocode; e.g. writing a "scheduler" or "course" class
5. Extended Abstract: ability to write code to solve a problem that has loose specifications; e.g. writing a program that adds students from a waitlist into a course until it is at capacity [?]?

A well-developed, complete, and functional program that a student produces to solve a problem in an unfamiliar context, then, falls somewhere in the highest levels of SOLO taxonomy, between the Relational and Extended Abstract levels.

It is clear from studying both Bloom’s and SOLO taxonomy that learning the big idea in computing of problem solving is both cognitively complex and requires structurally complex learning outcomes. These conclusions lead educators to suggest that algorithm education, then, is a difficult concept for students to grasp [11]

4.3 A Glance at the Statistics

The educational theory behind rating the cognitive complexity of algorithms curriculum provides a foundation for claiming that the big idea of problem-solving is hard. A brief survey of introductory programming courses which require students to employ problem-solving skills reveals that students are, in fact, not learning the desired skills.

A well-known study by Soloway, employed "The Rainfall Problem" as a benchmark for students. The Rainfall Problem asks students to write a program that reads in positive integers until it reads the integer 99999. After seeing 99999, the program should print the average of all of the integers. In a study of Yale’s novice computing
students, only 14% of students could produce a correct solution to The Rainfall Problem. A follow–up study by Guzdial yielded similarly low percentages. [7]

A multi–institutional, multi–national survey headed by McCracken gave a test consisting of a set of program–writing/problem–solving tasks of varying difficulty to students completing Computer Science I and II courses. The average score of these students of different backgrounds, nationalities, and educations, was only 23 out of 110 points, a score of 21% [9]. In fact, many of the students in the study never got past design questions to actually start writing code. The significant result from this study was “that the students did much more poorly than we expected” [10], indicating that many students do not learn necessary problem–solving skills in introductory programming and algorithms classes.

Another multi–national study, this time of graduating students, found that over 20% of students produced no program when confronted with a design project. 60% of students showed no significant progress toward a functioning design. These figures allowed the researchers of this study to conclude that “the majority of graduating students cannot design a software system.” [5]

Yet a third multi–national, multi–university study conducted by Lahtinen et al. surveyed some 559 students and 34 professors to identify the most difficult issues in programming. The students responded by defining "understanding how to design and program to solve a certain task” and "dividing functionality into procedures" as two of the three highest ranked tasks in difficulty. Professors responded similarly, also emphasizing the difficulty of "understanding programming structures.” [8]

These surveys and tests reveal that computer science students and professors alike identify that problem solving skills are difficult to learn. Further, they show that the current state of algorithm education is not teaching problem solving skills sufficiently.

5 Conclusions

It is well–known in the computer science community that problem solving is an essential task for programmers to be able to accomplish. Students in computer science undergraduate programs, then, should be learning the necessary skills for problem solving. Introductory programming courses and algorithms courses both target this "big idea" in computer science. Specifically, the curriculum in these courses focuses on (a) presenting existing tried–and–true algorithms to classic computing problems and (b) analyzing the complexity (efficiency) of these algorithms. Unfortunately, this curriculum has not been sufficient thus far in creating problem solvers. Students across the board are failing at simple design and implementation programming tasks which ABET deems to be essential to programming education. Analysis and evaluation of algorithm curriculum with the Bloom’s and SOLO taxonomies reveal that the cognitive complexity and learning outcomes of algorithm education are placed highly in the taxonomy hierarchies. This placement indicates that learning to problem solve is difficult for students.

It is necessary then that educators strive to create course curriculum and materials that account for the difficulty inherent in studying algorithms. Problem solving skills should be developed and encouraged, and then assessed within the algorithm curriculum. In doing so, educators should see an improvement in student responses in surveys evaluating ability to write a simple program to solve a given problem. The next two components of this project will (1) develop assessments which analyze a student’s progress in learning and developing problem solving skills and (2) propose a curriculum module which targets the learning objectives discussed in this paper.

References

