Hands-on Labs Without Computers

Shannon Pollard and Jeffrey Forbes
Department of Computer Science
Duke University Durham, NC 27708-0129
{shannon,forbes}@cs.duke.edu

Abstract

Often in teaching an introductory computer science course for non-majors, a primary focus on building programming skills is neither practical nor effective. Many instructors choose a breadth-first approach focusing on building problem solving skills and surveying computer science. This paper argues that conducting hands-on labs where students work together to physically implement algorithms is an effective supplement for programming labs on the computer. We present lab examples and summarize our experiences.

Categories & Subject Descriptors:
K.3.2 *Computer & Information Science Education*: Computer science education

General Terms: Human Factors

Keywords: CS0, Pedagogy, Lab environments, Non-majors

1 Introduction

At Duke University, there has been an ongoing project to teach computer science to a broader audience through the Great Ideas in Computer Science project [3]. Biermann's paper "Computer Science for the Many" proposes that it is possible to have a treatment of a broad selection of topics while maintaining a surprising level of depth and rigor [2]. As the course has evolved, there has been some examination of our goals for our introductory course for non-majors. The goal is to give a survey of computer science areas and the types of research problems that computer scientists encounter. We want them to be exposed to the limitations of computers and have an understanding of what is going on behind the scenes in computer programs that they know, like Napster or game programs. We also want to fulfill the university's "Science, Technology, and Society" course requirement by including a look at the social issues involved with computer science. We have the goal of giving the students knowledge of computer science as a field, taking away the "magic" of how computers work, and giving them practice in critical thinking and logical problem solving.

Having adopted a model of a field survey class like that of introductory biology or economics, we wanted to restructure the labs. Our labs had all been programming assignments, but they were contradictory to our goals for two reasons. First, the labs were not lending to understanding of computer science as a field, but rather to the syntax of one particular programming language. Secondly, it takes so much time to teach students how to program that we were not spending enough time in class to reach our goals.

The new labs would not be focused solely on programming but instead be hands-on labs that encourage a student to think about the concepts in different areas of computer science. The labs would follow the model used by the introductory physics class: the students discover a concept on their own through actual practice. The labs that we developed are new ways to present classic problems. Instead of giving the scenario to the student as a lecture example, we let the students puzzle out the problem in a lab without being distracted by the syntax of the implementation. We propose that the students can learn computer concepts and practice problem solving most effectively by having labs that do not involve programming, and in some cases, do not use a computer at all.

Many argue that instead of programming, problem solving is the core concept and unifying theme for a CS0

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'03, February 19-23, 2003, Reno, Nevada, USA.
Copyright 2003 ACM 1-58113-648-X/03/0002...$5.00

type introductory course [7]. Programming is often the vehicle used for problem solving, but there do exist other viable approaches. Some CS0 courses focus on achieving problem solving skills in the context of a computer literacy course [13] or by using applications such as spreadsheets [8]. Given the evidence that performance in a more programming oriented course is largely dependent on prior programming background [11], a more broad approach will even the playing field in a class with differing backgrounds and may do a better job testing and developing problem-solving ability. Various educators have advocated and implemented breadth-first approaches to introductory computer science [4, 1]. These approaches survey a number of computer science topics instead of focusing on programming, and are particularly well-suited for the introductory course for non-majors.

There do exist those that try to teach computer science concepts without using computers. One such effort is the Computer Science Unplugged project where they have created a set of hands-on exercises designed for elementary school children that explore computer science topics without any use of computers. Similarly, Marks et al. argue that hands-on exploration with implemented systems can be beneficial in teaching students who are not wholly computer science oriented [10]. Staying off the computer, using pseudocode, and working with visual aids allow students to focus on the algorithms, tackle more difficult problems, and have fun. Levitin and Papalaskari investigate the use of puzzles such as Towers of Hanoi and mazes to help teach design and analysis of algorithms [9]. Similar to hands-on labs, puzzles allow students to think about algorithms outside the restrictions of programming languages.

In this paper, we begin by introducing the methodology of Perspectives. We then give three examples of hands-on labs used in our introductory class and discuss the pros and cons of using them in Section 3. For our class, half of the labs used the computer and half were the hands-on labs as described in this paper. The paper concludes with lessons learned and conclusions.

2 Methodology

Integral to the lab experience was the theme of Perspectives. Every problem was looked at from six perspectives: technical, virtual, theoretical, social, ethical, and philosophical. The technical perspective focuses on how something works - making an abstract idea more concrete by showing an algorithm or some behind-the-scenes view. In contrast, the virtual perspective is about how a problem is modeled or how problems are grouped according to shared characteristics. The students were asked to abstract out concepts from seeing a specific problem solved. Students also learned some theoretical analysis skills, used to think about how the problem changes as the input size gets larger or as time passes. They learned tradeoffs between time and space and tradeoffs between programming effort and performance.

We also spent time looking at the social impact that computers have had, including how life has changed for the students given the advantages of the Internet and the increasing reliance of business on computers. We looked at the ethical questions raised by technology and debated the legal issues that follow. Finally, we looked at the philosophical view of a problem by thinking about how humans solve problems and comparing computer algorithms with a guess at how our own minds work. We included round-table discussions of what it means to learn, think, or understand.

The perspective method of teaching was integral to accomplishing our goal of a well-rounded view of the field of computer science, but had the fortunate side effect of making a course that is appealing to many different students. Some students were not as strong in logical thinking or math and were not excited about exercising these skills. Instead of a student simply struggling through a subject that is outside of his major, he is encouraged to bring his interests and areas of expertise into the problems and discussions. We found that students really liked this method because they were able to find a strength in one of the perspectives. The perspectives method let them find a way to use a skill that they were good at and did enjoy to gain an appreciation of computer science.

3 Lab Examples

This section presents three examples of hands-on labs that were used in our introductory class. The first is a lab that teaches students to think about how to do something complex by breaking the problem into components and solving each piece. The students learned how to write algorithms in pseudocode and use loops. The second lab is used to teach recursion using the problem of the Towers of Hanoi. The students use actual posts and disks and simulate the recursive process. Finally, we will give an example of a lab that teaches operating system concepts by having the students act out the dining philosophers problem. Each lab is based on a well-known problem from computer science literature but presented outside of a programming environment.

3.1 Herbie's Laundry

In one lab, students write algorithms to move a simulated cleaning robot analogous to Karel the Robot [12]. Students work in small groups, each with a checkerboard, checkers, and a Pez dispenser. The Pez dis-

penser represents Herbie the robot, the checkerboard is a room, and the checkers are piles of dirty laundry. The commands that Herbie understands are "walk forward", "turn left", "turn right", "go to sleep", and "pick up laundry". There are also predicates for seeing if there is a wall or laundry directly in front of the robot. The assignment is to write an algorithm so that Herbie goes all around the room picking up all the laundry, returns to his starting place, and then goes to sleep.

The lab divides the problem into sub-problems for the student. For example, the first task is to walk to the wall, followed by turning corners, and so on. The students solve the problem using variables, loops, and subroutines.

Students are able to complete this type of lab in the first week. As in all the labs presented, the students use algorithms that they do not yet know how to implement. Sometimes they learn the implementation before the end of the semester, other times only if they go on to become a computer science major. In any case, the students are learning to understand the problem and solution before trying to program.

3.2 Towers of Hanoi

The focus of this lab is to practice writing a recursive algorithm. In class, students have been taught the concept of recursion and seen examples. The lab introduces the Towers of Hanoi problem [6]. Students work on the lab in small groups. Each group has "towers" and "disks" to play with in the form of baby ring toys made by Fisher Price called Rock-a-Stack. The first part of the lab involves writing the disk moves for the three and four disks cases. Next they write an algorithm to move any number of disks from a starting post to an ending post. Questions guide them through finding the base case and understanding other nuances of the program. Finally, they must solve the problem recursively. The solution involves doing two recursive calls, but the students are not told so.

In this lab, we found that students are able to understand how to use recursion for a fairly complex problem. Although some groups will inevitably need help getting to the final solution, hearing each group have its "aha" moment is very encouraging.

3.3 Dining Philosophers

The final example is a lab that is based on Operating System concepts. The students have been taught about processes, shared resources, and multitasking. In lab they are presented with the dining philosophers problem [5] and are asked to use that framework to study resource sharing strategies. For the lab, the students are in groups of five and sit around a table. Instead of rice, there are LEGOs in the middle of the table. Instead of chopsticks, there are kitchen utensils (spatulas, serving spoons, etc...) between each seat. Instead of eating, the students must acquire both utensils and then use them to pick up the items in the middle of the table. The students may grab the utensils as quickly or as slowly as they want, but they can only grab one utensil at a time.

First, the students are given a strategy that deadlocks, then one that does not deadlock, but has the potential of starvation. For each strategy the students must explain how it could end in poor performance for the system. Finally, the group comes up with its own strategy that results in neither deadlock nor starvation. Most student answers are solutions that are very structured (only one process can go at a time, for example) to ensure that there is no deadlock or starvation. The groups were given an extra credit option to tell how their strategies resulted in less efficient use of resources than the previous two.

3.4 Effectiveness of the Labs

This section further describes the questions asked in the labs along with some representative answers from the students.

Laundry: The questions for the laundry lab included examining the process of solving the problem through subproblems, and studying how the problem becomes more complex as the pieces are put together. One question was to tell how the solution would change if the robot could see in all directions. Students were able to say that "it would make the problem more complex but in the end, if programmed right, more efficient."

The students were also asked to think about practical issues that would arise if they were programming a real robot. Students readily pointed out flaws in the lab model of the problem, including the shape of the room, the size of the squares, and the lack of obstacles. "All grounds are not the same. He may reach a snag in the rug and get stuck, unless he is an all-terrain robot." "Another problem a robot would face is the longevity of its battery."

They were asked to think about how household robots change our lives, and they were challenged to define their own trust of technology. Would they let a robot do their housework, drive their car, or babysit their children? The class had to tell the guidelines they would use to tell whether the area of their lives could be run by man or machine. One student wrote, "I just don't think that anything without emotion and feelings should be trusted to deal with people and personal issues." Another said, "I would trust a robot to do the simple tasks that a child could do, but not anything that required a

decision or responsibility." The class discussed what advances would have to be made to strengthen their trust and reliance on technology.

Finally, they were asked to brainstorm ways that a real robot might detect dirty laundry. The students had not been exposed to any visualization or artificial intelligence algorithms, they were simply asked to be creative and use things they supposed that computers could do given their everyday experience with them. The answers to this question were varied and extremely entertaining. Student ideas included having the person dust their dirty clothes with a chemical that the computer could detect, making a computer be able to detect body odor, scanning pictures of your clothes into the computer for comparison, or putting a red ball on the piles before you turn the robot on (the real robot in our department can go to the red ball!). Of course, one student suggested using a hamper.

Recursion: The questions for the Towers of Hanoi lab showed students some differences between writing an algorithm using loops and using recursion. We ask them what happens when there is no base case and how the choice of a base case might change functionality. We also asked them to try to write the algorithm for the Towers of Hanoi without using recursion. This was very effective in teaching the elegance and ease of writing recursive algorithms. In fact, even though they were shown how a recursive algorithm is executed, students thought that recursive algorithms were so easy to write that they were cheating somehow. Questions about recursion revealed these feelings: "All you have to do is identify a base case and then tell the program to solve a smaller problem and then just add on a small part. You don't actually have to go through all the computation. How easy!" "Recursion can greatly cut the complication and length of an algorithm written by a while loop." "With recursion, the code becomes almost magically simple." Later in the course, students would write algorithms recursively, even when it was not required.

The lab also points out the link between recursion and infinite data structures or infinity in general and asks students to look at some of Escher's "recursive" pictures, in which a structure folds in on itself, and tell their reaction to the pictures. Finally, they are challenged to define some everyday relationships recursively. One student said, "I could say that my teacher is my teacher's teacher. For example, my teacher has learned computer science from other teachers, so in some cases, she is teaching me what she was taught by her teachers. She is still teaching me, but through her, her teachers are teaching me too."

Dining Philosophers: The questions for the Dining Philosophers lab focused on relating the experience of playing with utensils and toys back to processes and resources. They were asked to explain why computers running many processes at once sometimes run slower. "The more processes that are competing for resources, the longer it takes for a single process to get that resource." "This system would take longer and longer to perform the more processes there are." The students were introduced to the need for prioritizing processes that might need to get resources right away for real-time computing. They were asked to change their strategy to include giving priorities to processes. One question introduces them (very briefly) to readers and writers and asks them to speculate on how many readers should be allowed concurrent access to a resource, how many writers should be allowed concurrent access, and how many readers and writers together should be given access. This question had a follow-up question that asks what might happen when two algorithms, running concurrently, try to change the same piece of data.

We also included questions that asked the students about their own multi-tasking in their lives. "Multitasking can be seen in everyday life by conversing with friends and family at the dinner table, doing many errands in one trip, shaving in the shower, and doing your lab when the teacher is lecturing (not that I have ever done this)." They were challenged to mirror the advantages and disadvantages of multitasking on the computer to multitasking as a way of life. One student says, "Society encourages to multi-task for sure. There are new products everyday designed for multitasking...picture in picture TV's, books on tape, and of course cell phones. Yet at the same time, society has begun to realize that it is not always the best idea and might even be dangerous." Finally, they brainstormed the ways that a virus program could wreak havoc on a computer by reprogramming the operating system's resource allocation strategy.

As illustrated by these examples, we were able to ask questions about some fairly advanced topics, and received reasonable answers. This leads us to believe that the labs are successful teaching tools and not just playing with props. Many students who have learned to program over a few semesters could not answer these questions about their own programs! We think that the reason for this is not only because the introductory students were exposed to higher level concepts, but also because they were allowed to discover, through their own trial-and-error experience, answers to very interesting questions that computer science majors are usually simply told as examples.

4 Lessons Learned

Although we found the hands-on labs very effective as a whole, there are some drawbacks for using them.

These labs do not always have straight-forwarding grading schemes, since answers are not just right or wrong. Also, the job of a TA in the lab changes significantly. Instead of being trained to find and correct programming errors, the effective TA will need to be able to explain the concepts that the lab is trying to teach, and give hints toward finding solutions without giving away the answer. The class we taught was very small, and it is not clear how the class environment will scale to classes of one hundred or more students. A large class would probably need more well-trained manpower than the traditional class.

Secondly, there are some disadvantages to not teaching programming or computer literacy in the introductory course. Often students are expecting to come out of the class with a marketable skill, and in this environment the student cannot claim to have working knowledge of any programming language or application. The student that takes the introductory class and then goes into a programming class will have an advantage in problem-solving skills only.

5 Conclusion

On the other hand, there were significant advantages for using the hands-on labs. The first semester that we taught using the perspectives methodology, half of the labs were hands-on, and half were on the computer. Attendance to the hands-on labs was definitely better than attendance to computer labs, but this could also be because those were labs were harder for the students to make up later. However, one student told us that the hands-on labs were "more fun, but harder." Another said, "While some [hands-on labs] seemed to be more difficult than the computer labs, they got us thinking rather than just having us follow rote directions off the lab sheets." We feel that this is the best situation for both instructor and student: when labs present a greater challenge, yet students are looking forward to lab time.

We found that the hands-on labs were fun for the students and allowed us to present them with more advanced topics and interesting problems. Due to the Perspectives method of teaching computer science, the students found an appreciation for the problems encountered in computer science and were able to find similar issues in how computers are used in their own favorite subject. They were able to discuss computer issues in society and philosophy intelligently. They were able to bring their own strengths and interests to the class, which enabled them to explore the technicalities of how things work with more excitement. Due to the emphasis on concept understanding rather than implementation details, students were able to understand and apply concepts from more advanced topics in computer science, including electrical circuits, artificial intelligence, operating systems, and analysis of algorithms. The result was an effective survey course of the science of computers.

References

[1] Bagert, D., Marcy, W., and Calloni, B. A successful five-year experiment with a breadth-first introductory course. In *SIGCSE Bulletin* (1995), vol. 27, pp. 116–120.

[2] Biermann, A. W. Computer science for the many. *IEEE Computer 27*, 2 (February 1994), 62–73.

[3] Biermann, A. W., and Ramm, D. *Great Ideas in Computer Science with Java.* MIT Press, 2001.

[4] Denning, P. J., Comer, D., Gries, D., Mulder, M. C., Tucker, A. B., Turner, A. J., and Young, P. R. Computing as a discipline. *Communications of the ACM 32* (1989), 9–23.

[5] Dijkstra, E. W. Hierarchical ordering of sequential processes. *Acta Informatica 1* (1971), 115–138.

[6] Édouard Lucas. *Récréations Mathématiques*, vol. 3. Gauthier-Vallars, 1893.

[7] Joyce, D. The computer as a problem solving tool: a unifying view for a non-majors course. In *SIGCSE Bulletin* (1998), vol. 29, pp. 63–67.

[8] Kolesar, M., and Allan, V. Teaching computer science concepts and problem solving with a spreadsheet. In *SIGCSE Bulletin* (1995), vol. 27, pp. 10–13.

[9] Levitin, A., and Papalaskari, M.-A. Using puzzles in teaching algorithms. In *SIGCSE Bulletin* (2002), vol. 34, ACM Press, pp. 292–296.

[10] Marks, J., Freeman, W., and Leitner, H. Teaching applied computing without programming: a case-based introductory course for general education. In *SIGCSE Bulletin* (2001), vol. 33, pp. 80–84.

[11] Morrison, M., and Newman, T. S. A study of the impact of student background and preparedness on outcomes in cs i. In *SIGCSE Bulletin* (2001), vol. 33, pp. 179–183.

[12] Pattis, R. *Karel the Robot: A Gentle Introduction to the Art of Programming*, second ed. Wiley, 1995.

[13] Townsend, G. C. Turning liabilities into assets in a general education course. In *SIGCSE Bulletin* (1998), vol. 30, pp. 58–62.