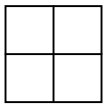
Preface

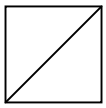
The Greek philosopher Plato relates the following story[1]. A master is sitting with a student, and draws a square in the sand.

"Can you", the master asks the student, "draw another square which will have an area *exactly* half that of the square I have drawn?" The student, realizing that the sides of such a square would not be any fractional part of the side of the original, says no.

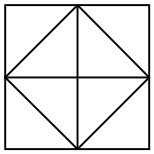
"Can you", the master asks the student, "divide my square into four smaller equal sized squares?" To which the student responds: "Easily!" And draws the following picture



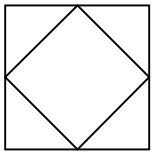
The master then draws another square the same size as the first. "Can you", he asks, "draw a line that divides this new square into two equal sized triangles?" The student once again answers in the affirmative, and draws the following:



At this point the student suddenly realizes the key to the problem. Returning to the original drawing, she divides each of the four smaller squares in half:



Erasing the inner lines, she leaves the following in the sand.



"Now," she says triumphantly, "the inner square has exactly half the area of the original".

[1] The story appears in the book *Meno*. In the original the master is Socrates, and the student is a male slave. I should state here that I agree with Plato only in part. In any technical field there is new information that the student cannot possibly have seen previously. But the key feature I take from this story is that a teacher must help a student feel confident that they understand the material, and more importantly the student must be an active participant in the process, and not simply a passive observer.

Plato uses this story to argue that the student already possessed the knowledge of the answer, but that she was not aware that she knew the answer. The role of the master is not only to present new information, but also to lead the student to understand what it is they already know.

A few years ago I had occasion to remember this story. I had written several data structures textbooks that had received modestly favorable comment. Visiting an institution where my book was being used, an instructor offered this frank assessment: "You have many clever programs, but my students could never produce code that looked like your code. You *show* the students too much, and expect them to *do* too little".

I returned home a little bit stung by this criticism. I had spent many hours in writing the code contained in my books, and with no small amount of *hubris* I was somewhat proud of the result. But the more I thought about it, the more I realized the instructor was wrong in the first part, correct in the second, and that the second was key to the first.

Like many others, I had long worried about a persistent and unfortunate secret that has plagued our profession. Our students (and not just at my institution, but all over) could graduate from a year-long sequence of courses, and still not know how to program. I looked at my own teaching style, and asked myself "what was I doing wrong"?

Like most instructors, I had adopted the format of hour-long lectures, illustrated with nicely developed powerpoint presentations, covering all the essential material, and often much more as well. This would be reinforced with a good amount of homework, programming assignments, laboratories, and occasional examinations.

But in thinking about the process of teaching a skill such as programming, I realized that programming has much more in common with crafts such as creative writing or learning a foreign language than we often would like to believe. And we certainly don't teach creative writing in the style I was using to teach programming. It would be a bit like having the students read Faulkner, and then expecting them to be able to write a novel in a southern dialect. Owen Astrachan uses a different analogy, he says it is like showing students how to use a saw and hammer, and then expecting them to be master cabinet builders.[2] But the only way to learn to be a writer is to write. The only way to learn how to create furniture is to do it. Similarly, the only way to learn how to be a programmer is to program.

It was thinking along these lines that lead me to the approach I now use, and that I advocate in this book. In pondering my hour-long powerpoint presentations, I realized that more often than not the new key idea would be contained in the first ten minutes. The remainder of the time would then be spent illustrating this idea, going through examples, looking at code, and often a good bit of minutia. And while I also think that being able to read code is an important skill (and students need to also do more of this), my students were missing the first important step, which was learning to write.

---

[2] *Why I care about Programming and How to Teach it*, Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, Norfolk, VA, March 2004.

Students would suffer through the presentations, and confuse understanding the words with understanding the concepts. In short, like the student in Plato's story, they often did not know what it was they knew. Of course, I augmented the lectures with traditional homework and programming assignments designed to reinforce the ideas contained in the lectures. More often than I would have liked it was not until the students reached the point of doing this homework, in the solitude of their own room, that they realized their confusion. (Or worse, they would realize their lack of knowledge only when faced with a blank examination sheet).

Remembering Plato, I recognized that if I could help the student to discover what they needed to know, help them to take an active part in the learning process, rather than being a passive observer, that they would have greater confidence in what they did know, and greater appreciation for what they did not know. This movement towards allowing students to become more involved in the educational process is now known as *active learning*.

The style I developed works as follows. Most days I will present a fifteen to thirty minute lecture, in more or less the traditional form. This will then be followed by a daily worksheet. I call these worksheets an "anti-quiz", because unlike a traditional quiz the students are encouraged to work cooperatively, ask questions, use the book, and in general do whatever is necessary to ensure that everybody has the correct answer by the end of the hour. I wander around and help the students, giving advice and direction freely. I collect the anti-quiz worksheets in order to get a feel for the how the students are progressing, but the answers are not "graded" in the traditional sense. (Once in a while I will have the students take the anti-quiz home to work on, and will collect and discuss them the next day).

These class sessions are not "laboratories" in the traditional sense, because they are not creating a complete application and are not working at a keyboard. The fact that they are not addressed sitting in front of a keyboard is important. It means the students are spending time thinking about the logic of the problem at hand, and not dealing with syntax errors and compiler issues. But they are lab-like. They are designed to help the student to learn how to *think* like a programmer.

If you look at the discursive material that precedes each daily worksheet in the material that follows, you will see that I *present* far less code than is typically found in a data structure textbook, in the sense that I don't show completely developed classes. Rather than looking at the code I have written, students *create* their own abstractions, thereby become a participant in development, rather than a passive observer. As my critic indirectly requested, I am presenting less and asking more.

I have found a number of benefits of this approach:

- Students spend more time programming. As I indicated earlier, I believe the more experience you have with programming, the more quickly you will become a programmer.
- Students experience a more methodical approach to software development. Rather than programming by responding to cryptic error messages from a compiler, students experience that the first tool in algorithic development is logical thinking.
- Students gain more confidence in their own skills.
- By gathering and examining the worksheets on a daily basis, the instructor has immediate feedback on how well the students are understanding the material.
- Since examination questions are often very similar to the daily worksheets, students know what they will be expected to know and be able to do on the exam.

Schedule and Format

Almost all of the worksheets in this book can be completed in a typical one-hour class, including the initial introductory lecture. There are only a few topics (skip lists, AVL trees) that might require more than one class session. I have provided more topics than will fit into a typical ten week quarter or fifteen week semester. This allows the instructor to pick and choose. Some topics (self-organizing lists, for example) are of less importance and can easily be omitted in the interest of time. It is most important that the students learn about various sorting algorithms, the concept of big-Oh, the difference between linear and binary search, the dynamic array data structure, the implementation of the various ADT's using a dynamic array, a variety of Linked List data abstractions, the use of random chance in the implementation of the Skip List, trees of various types, efficient trees (AVL or red-black), heaps and hash tables. Other topics can be included or omitted at the instructors discretion.

Some of the worksheets include additional work that typically can not be expected to be completed in the one-hour time period. If desired, these can be assigned as homework assignments. I have found that worksheets that are not used in lectures can be a good source of examination problems.

Many people have provided comments or suggestions as I developed this approach, and their assistance has been most welcome. I continue to solicit comments on this approach, which I truly believe is most beneficial to the student.

Happy learning – Tim Budd