Translation from Problem to Code in Seven Steps

Andrew D. Hilton
Duke University
Durham, NC, United States
adhilton@ee.duke.edu

Genevieve M. Lipp
Duke University
Durham, NC, United States
genevieve.lipp@duke.edu

Susan H. Rodger
Duke University
Durham, NC, United States
rodger@cs.duke.edu

ABSTRACT

Students in introductory programming courses struggle with how to turn a problem statement into code. We introduce a teaching technique, "The Seven Steps," that provides structure and guidance on how to approach a problem. The first four steps focus on devising an algorithm in English, then the remaining steps are to translate that algorithm to code, test the algorithm, and debug failed test cases. This approach not only gives students a way to solve problems, but also ideas for what to do if they get stuck during the process. Furthermore, it provides a way for instructors to work examples in class that focus on the process of devising the code—instructors can show *how to come up with the code*, rather than just showing an example. We describe our experience with this technique in several introductory programming courses—both in the classroom and online.

CCS CONCEPTS

• **Social and professional topics → CS1**.

KEYWORDS

CS1, introductory programming, from problem to code, metacognition, computational thinking

ACM Reference Format:
Andrew D. Hilton, Genevieve M. Lipp, and Susan H. Rodger. 2019. Translation from Problem to Code in Seven Steps. In *ACM Global Computing Education Conference 2019 (CompEd '19), May 17–19, 2019, Chengdu,Sichuan, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3300115.3309508

1 INTRODUCTION

Novice programmers often struggle to take a problem statement and turn it into working code. This struggle arises from needing to both devise an algorithm to solve a class of problems, as well as to turn that algorithm into working code. This struggle is seen in a multi-national ITiCSE working group study [11] that showed students do not know how to program at the conclusion of an introductory programming course. Their study collected data on over 200 students from four institutions in which the average student score was 21%. Many students struggled to write any code at all. They were stuck on the algorithm design. Another multi-national ITiCSE working group study [9] showed that some students in introductory courses may have a fragile grasp of the skills needed to problem solve. In this study they tested students on predicting the outcome of short code segments and on completing near-complete code from a small set of possibilities. Many students were weak at these tasks, especially the latter.

Some approaches to teaching programming instruct students to first devise an algorithm, and then write code. They may have several steps in their approach with the algorithm as only one of those steps [16, 19]. However, devising an appropriate algorithm is the more difficult part of the programming process. Students should not only be told *to* devise an algorithm, but also be instructed in *how to* devise an algorithm. In this paper, we present *The Seven Steps*—the approach we teach our students to devise and implement an algorithm. This process, which we describe in Section 3, not only describes how to develop an algorithm and turn it into code, but also gives students strategies for when they are stuck on a particular step. We use this approach with small problems in introductory programming courses. For larger problems, the problem could be broken down into components and our approach applied to each component.

The Seven Steps forms not only a way for students to work problems, but also a way for instructors to present material. Rather than showing students example code (which students are often prone to try to memorize), an instructor teaching with The Seven Steps can present an example of *how to come up with* a piece of code. The instructor can work through The Seven Steps, starting from a problem statement, and follow the steps to produce working code. In such an approach, students can understand the logical thinking that led to the algorithm and ultimately code to solve the example problem. Such an approach takes more instructional time than simply showing students working code but in our experience has far better results.

In the remainder of this experience report, we describe related work (Section 2), then describe The Seven Steps (Section 3), including some examples of its use. In Section 4 we discuss its usage in our in-person courses at Duke University, as well as in two online Coursera specializations. In Section 5 we evaluate The Seven Steps based on student anecdotes, as well as an end-of-semester survey. Finally, we conclude in Section 6.

2 BACKGROUND AND RELATED WORK

There are many practices to help novice programmers succeed in learning to program. Some practices provide support to learners, such as Peer Instruction [3] and Pair Programming [12, 20]. Some practices focus on the subject matter of the material as a motivating factor, such as Media Computation [5]. For example, Porter and Simon [15] showed that combining Peer Instruction, Pair Programming, and Media Computation resulted in a 31% improvement in

CompEd '19, May 17–19, 2019, Chengdu,Sichuan, China

ACM ISBN 978-1-4503-6259-7/19/05.
https://doi.org/10.1145/3300115.3309508

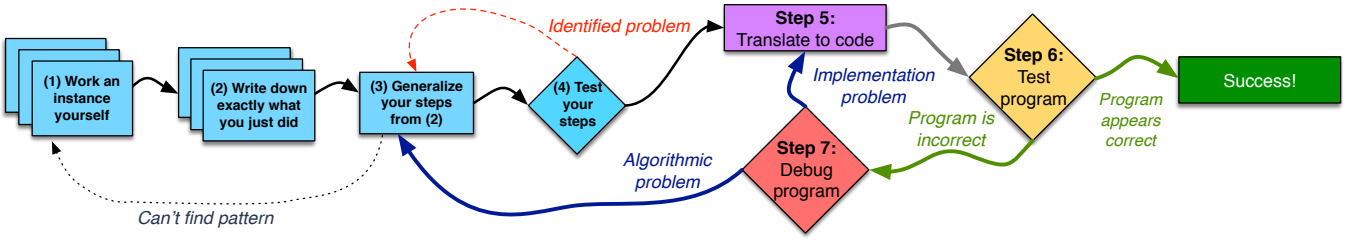


Figure 1: Diagram of The Seven Steps

retention of students in their CS1 course. Some practices provide a programming environment that aids the programmer in creating the code, such as the drag-and-drop interface in Alice [18] and Scratch [14]. Some practices focus on providing an environment for debugging the code, such as the program visualization tool Python Tutor [4]. All of these practices help the novice programmer in some way, but not in understanding how to develop an algorithm from a problem statement.

Problem solving in the context of introductory programming can be defined in many ways, and we list four such ways. First, computational thinking [21] includes the core concepts of abstraction, algorithmic thinking, decomposition, and generalization and pattern recognition—different meanings of it were discussed in two workshops [7, 8]. Second, Problem-Based Learning (PBL) was adapted to computer programming [13] with group work PBL sessions that involved seven steps: 1) examination of the case, 2) identification of problem, 3) brainstorming, 4) sketching explanatory model, 5) establishing learning goals, 6) independent studying, and 7) discussion. Third, the ITiCSE Working group [11] defined problem solving as the learning objectives for their assessment in a five-step process: 1) abstract the problem from its description, 2) generate sub-problems, 3) transform sub-problems into sub-solutions, 4) re-compose, and 5) evaluate and iterate. These three methods are very abstract. We were interested in defining concrete steps that students could more easily follow.

A fourth approach to problem solving in introductory programming is metacognition, being aware of and understanding one's own thought processes. In [10] the authors give explicit instructions on problem solving in stages that involve reinterpreting the problem, searching for similar solutions, evaluating those solutions and then implementing a solution. Our approach is different in that four of our seven steps focus on working out an algorithm with a sequence of small problems solvable by hand.

3 THE SEVEN STEPS

Because many students do not have much experience thinking about how they solve problems (they just do), we introduce The Seven Steps, shown in Figure 1 to give them a framework for thinking about solving a programming problem. Here is a summary of the explanation we give students:

Step 1: Work an example yourself. The first step is to work an example yourself by hand. If you cannot yet do that, it means either that you need to further specify the problem or that you need domain knowledge.

Step 2: Write down exactly what you just did. The next step is to write down the details of how you worked the problem by hand in Step 1. If you get stuck on this step because you "just knew it," you should try a more complex example that you cannot just solve by inspection.

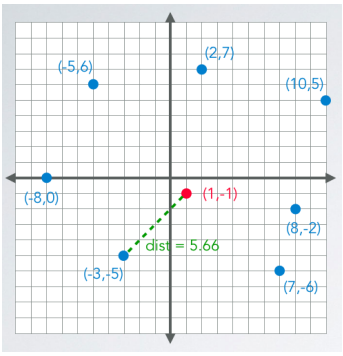
Step 3: Generalize. Once you have worked at least one example by hand and written down the process, you can begin to generalize. Why did you do something a certain number of times? Where did you start? When did you stop? It is often necessary to have worked several examples by hand to generalize well. In fact, if you have trouble with this step, you should repeat Steps 1 and 2 on different instances of the problem.

Step 4: Test your algorithm. Now that you have a draft of a generalized algorithm, apply it to a new instance of the problem, and see if you get the correct answer. This is essential for catching generalization mistakes. Finding mistakes before translating to code avoids wasting time. You can go back to Step 3 if you identify a generalization mistake.

Step 5: Translate to code. Steps 1–4 can be done with pencil and paper and are independent of language. For Step 5, you need to know the syntax of a language and how to translate parts of the algorithm, such as counting or decision-making constructs. It is often helpful to start with your algorithm steps as comments before you write any code. If any of the lines in your algorithm do not immediately translate into one or two lines of code, they should be abstracted out into their own function: make a good name for the function, call it here, and make yourself a note about what it does. After you finish translating this algorithm, go implement that function. That function is itself a programming problem (for which you wrote the problem statement), so use The Seven Steps.

Step 6: Test. Another round of testing is important because you could have a correct algorithm and still have made mistakes in the implementation in code. You also can have the computer run many more test cases more quickly than you can do them by hand. Testing is the act of finding bugs, and you want to find as many as you can by writing a robust set of test cases. While it is not possible to know for certain that your program is correct, you can become more and more certain with good testing.

Step 7: Debug. Debugging is the act of fixing bugs you identified in the testing stage. Once you've identified a problem, you need to figure out if the issue is with the algorithm or code implementation and go back to Step 3 or Step 5, respectively. We teach debugging by the scientific method, rather than ad hoc, as many students would do otherwise.



(a) Step 1: Working this example by hand

Write down exactly how you solved the problem:

(1) Computed $\sqrt{1^2 + 8^2} = 8.06$
(2) Started with best choice of $(2, 7)$
(3) Computed $\sqrt{9^2 + 6^2} = 10.82$
(4) Compared 10.82 to 8.06—8.06 is smaller
(5) Computed $\sqrt{7^2 + (-1)^2} = 7.07$
(6) Compared 7.07 to 8.06—7.07 is smaller
(7) Updated best choice to $(8, -2)$
(8) Computed $\sqrt{6^2 + (-5)^2} = 7.81$
(9) Compared 7.81 to 7.07—7.07 is smaller
(10) Computed $\sqrt{(-4)^2 + (-4)^2} = 5.66$
(11) Compared 5.66 to 7.07—5.66 is smaller
(12) Updated best choice to $(-3, -5)$
(13) Computed $\sqrt{(-9)^2 + 1^2} = 9.06$
(14) Compared 9.06 to 5.66—5.66 is smaller
(15) Computed $\sqrt{(-6)^2 + 7^2} = 9.22$
(16) Compared 9.22 to 5.66—5.66 is smaller
(17) Gave an answer of $(-3, -5)$

(b) Step 2: Write down exactly what you did

Figure 2: Steps 1 and 2 of the closest point example

We give examples on how we teach with this technique from two courses at Duke University. ECE 551 is an intensive introductory programming course in C and C++, for graduate students in Engineering. CompSci 101 is an introductory programming course in Python, for undergraduates.

3.1 Example 1 from ECE 551

From ECE 551 we give an example of executing The Seven Steps for the closest point problem—finding the closest point in a set of points to a given point in the plane. We use this problem, as it not only illustrates many algorithm concepts but is useful in a wide variety of applications: from maps to tagging data based on similarity.

Step 1. We pick a set of points S and single point P and find which point in S is closest to P

$$S = \{(2, 7), (10, 5), (8, -2), (7, -6), (-3, -5), (-8, 0), (-5, 6)\} \quad (1)$$
$$P = (1, -1) \quad (2)$$

We strongly encourage drawing a diagram. We also identify the domain knowledge needed to solve this problem, namely the Pythagorean theorem, rearranged into the distance formula: $d = \sqrt{\Delta x^2 + \Delta y^2}$. By hand, we calculate the distance between each point in S and P, finding that $(-3, -5)$ has a distance of 5.66, smaller than each of the other distances calculated, and shown in Figure 2a.
Step 2. We write down exactly what we did in Step 1, with a focus on being precise and complete. See Figure 2b.
Step 3. We generalize the results from Step 2, recognizing some key patterns: the "compute" and "compare" lines come in pairs; updating the best choice only happens sometimes, and some lines only happen at the start and end.
The resulting algorithm is

```
Compute the distance from S_0 to P--call it bestDist
Start with best choice of S_0
Count from 1 to the number of points in S exclusive,
  call each number i
    Compute the distance from S_i to P
    If currDist is smaller than bestDist,
        Then update bestChoice to S_i and
            update bestDist to currDist
Give an answer of bestChoice
```

Here is more explanation. Each compute uses Δx and Δy from point P to each point in S, so we generalize this to "Computed $\sqrt{(S_i\text{'s } x - P\text{'s } x)^2 + (S_i\text{'s } y - P\text{'s } y)^2}$, (call it `currDist`)." Each comparison always compares `currDist` to something. We realize that we are implicitly keeping track of the `bestDist`, and the update only happens when `currDist` is smaller than `bestDist`.
Step 4. We tested this code for several inputs, including the corner case where there are zero points in S. This reveals a problem with the algorithm, so we add the line

```
If S is empty, give an answer of "no answer exists"
```

Steps 5–7. This example was used in the first part of ECE 551, before students learned to write code, so we only practiced Steps 1–4. Later in the course, when students are learning about pointers and arrays, we revisit this problem. The students work through the translation to code, building their understanding of these concepts and how they relate to the algorithm that was designed earlier.

3.2 Example 2 from CompSci 101

In CompSci 101, we use The Seven Steps to solve the following problem (from the TopCoder website [17]). Given a word, translate it into a cryptic text message using the following rules. 1) If the word is only vowels, it is not changed. 2) If the word has at least one consonant, then write only the consonants that do not have another consonant immediately following them, and do not include any vowels. 3) Vowels are 'a', 'e', 'i', 'o', and 'u'.

Step 1. Work an example. The word is "please". We step through the word on paper, letter by letter, and end up with the result "ps".
Step 2. Write down exactly what you did. 1) Word is "please". Create empty answer. 2) First letter is "p", consonant, no consonant before, include it, answer is "p". 3) Next letter is "l", consonant with consonant before it, do not include it. 4) Next letter is "e", vowel, do not include. 5) Next letter is "a", vowel, do not include. 6) Next letter is "s", consonant, vowel before, include, answer is "ps". 7) Next letter is "e", vowel, do not include. Our answer is "ps".

Step 3. Generalize. In generalizing, we discuss how we find repetitive behavior and conditional behavior, as is typical of many algorithms. For this problem, we see how making the lines in the algorithm repetitive is done most easily by "pretending" that there is a vowel before the word—doing so removes special cases and makes the algorithm uniform. Generalization results in an algorithm that looks like this:

```
start with before being 'a'
ans is empty
for each letter in word
    If letter is a consonant, and before is a vowel,
        then add letter to ans.
    set before to the current letter
give back ans as your answer
```

Step 4. Test your algorithm. We follow the algorithm with the word "message" and get "msg". It works!

Step 5. Translate to Python code. Translation to code proceeds in a straightforward fashion: each line in the algorithm corresponds directly to one line of code. Checking if a letter is a vowel (or consonant) are logical choices to abstract out into another function. Students are familiar with the function `isVowel()`, so they can simply use it without solving another problem. The resulting code is:

```
before = 'a'
ans = ''
for ch in word:
    if not(isVowel(ch)) and isVowel(before):
        ans += ch
    before = ch
return ans
```

Step 6. Test. Testing shows that this algorithm works on many cases, but not on the string "a" (where the result should be "a", but is "").

Step 7. Debug. Debugging shows us that the problem was with our algorithm, so we return to Step 3. We note that repairing this algorithm likely requires working more examples, as none of the examples previously worked behave like the problematic test case. Once the algorithm is repaired, it can be translated back into code (repeating Step 5), and re-tested to find that it is now correct.

4 USAGE OF THE SEVEN STEPS IN COURSES

4.1 ECE 551

ECE 551 is an intensive programming class for Masters students in Electrical and Computer Engineering. This course builds fundamentals from the ground up, to take students from whatever their prior programming background might be (including nothing) to solid programmers ready to take graduate-level computer engineering courses the next semester. This course has pervasive use of The Seven Steps—we discuss it (and its importance) on day 1 of the course, and use it as the primary way to teach examples.

This course uses a flipped classroom, so the examples are not done in lecture. Instead, the students watch them in videos, which are part of the textbook [6] that we wrote for this course (which has embedded videos). This book includes 19 video examples where we use The Seven Steps to devise an algorithm and turn it into code. Three of the early chapters are entirely devoted to detailed discussion of The Seven Steps (one focuses on Steps 1–4, one focuses on Step 5, and one focuses on Steps 6 and 7). This course also makes use of the Lego Lab, described in Section 4.3.

Furthermore, when students seek help, the instructors and TAs give assistance within the framework of The Seven Steps. For example, we respond to students saying "I'm stuck on this problem" with "Which step are you having problems with?" When students say "I cannot find the pattern in these steps," then we respond that it is quite good that they can precisely identify the problem they are having and work with them on how to find the pattern in the steps they wrote.

4.2 CompSci 101

CompSci 101 is an introductory Python programming course for undergraduates. This course is the first computer science course for the major, and typically 80% of the students have never programmed before. We integrate examples of The Seven Steps into the course and illustrate with those examples how this technique helps to write code. We introduce the technique in the fifth lecture and show them all of The Seven Steps for solving a particular problem. We work the first four steps with pencil and paper and then work Steps 5–7 on the computer. We ask them for input as we work through all seven steps. We spend a lot of time on this first example so students can see how we would solve each step. Writing an algorithm by hand and going through all seven steps in such detail takes a long time. For later examples, we built the problem solving steps into our lecture slides so that we could cover them more quickly and have time to go over more examples in detail. We used this technique in eight of the 26 lectures (30%), spread throughout the semester, and used it in the Lego Lab, described in Section 4.3. In the majority of the examples we focused on the first four steps, and in a few examples we worked through all seven steps.

In two consecutive lectures, we revisited a problem to show a different translation to code. In the first lecture for this problem we went through all seven steps to solve the problem, with the code resulting in a loop that iterates through all of the elements. In the next lecture we learned about indexing with a loop. Then we revisited the same problem from the previous lecture, but started with step 4, the same algorithm we had previously developed. This time we translated each line from that algorithm into different code that used indexing, resulting in a cleaner solution for the problem.

In another lecture, we were able to show the value in putting time into developing an algorithm before coding. In this lecture we worked through the first four steps for a problem. Then we translated each line of the algorithm with live coding with suggestions from students. We ran the program on several sets of test data and it worked correctly the first time! Students were very impressed. This really showed how investing a lot of time in the first four steps meant that little to no time was spent on debugging.

4.3 Lab with The Seven Steps

Both courses do a lab which focuses on Steps 1–4 of The Seven Steps, in which students determine and write down the algorithm used to place Legos on boards in a sequence of boards that defines a particular pattern. Patterns typically start with one lego on a

board, then two legos on a board, then three legos on a board, etc. Students are suppose to identify an algorithm so that given a particular number, say 8, they could put down 8 legos on a board that following the pattern. The goal of this lab is to completely decouple Steps 1–4 from anything dealing with code (it can be done before students have even learned any syntax of any programming language).

An example Lego pattern is shown in Figure 3. This pattern would be appropriate when students have learned about repetition, conditionals, and modulus. This example uses two types of legos, square brown legos and orange rectangular legos that are twice the size of the square legos. The pattern is the following. The first board labeled 0 has one lego, a square lego in the bottom left corner. The second board labeled 1 uses two legos, a rectangular lego in the bottom left corner and a square lego that starts four columns over and one row up. Each consecutive board has one more lego. Students will need to make observations to help them in writing an algorithm that describes the pattern. They may notice the first lego on each board alternates from square to rectangle to rectangle, and then repeats. Meaning the seventh board would start with a square lego in the bottom left corner. They may notice that consecutive legos on a board are always four columns over and one row up. It can be a challenge for students to write an algorithm for the Nth board that describes the placement of the N legos on that board.

Prior to the lab, the instructors prepare two different patterns: the "A" and the "B" pattern. The algorithm one writes to explain a pattern is parameterized over one integer (N). The complexity of the algorithm should be appropriate to the amount of practice/skill the students have. The concepts involved should be appropriate to the material covered recently. Instructors (or TAs) execute the algorithm for $N = 0, 1, 2, 3, 4$ (more values of N if needed for a complex algorithm). Each parameter value is done on a separate Lego board. Boards are labeled with the value of N with a post-it note.

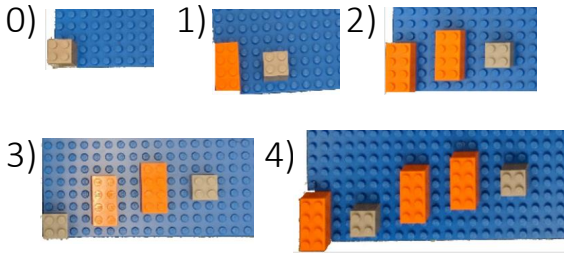


Figure 3: Example Lego boards for an algorithm with N=0–4.

For the actual lab session, students are divided into groups (approximately 4 students per group). The groups are then divided into "A" and "B" groups. Each "A" group is given a set of boards on which the "A" pattern is shown for *e.g.*, $N = 0$ to 4 legos, and each "B" group is given a set of boards representing the "B" pattern. "A" groups should not be able to see "B" boards, and vice-versa. The students are then instructed to devise and write down an algorithm which would produce the pattern of Legos for any value of N. Each group has some spare Legos and boards so that they can work examples themselves.

After the students have had sufficient time to devise their algorithms, they are instructed to execute their own algorithm for the next value of N (*e.g.*, if they were given $N = 0$–4, they are instructed to execute $N = 5$). When they complete this task, they swap algorithms (but not boards) with a group who had a different pattern ("A" swaps with "B").

The students are then instructed to execute the algorithm they *received* for a value of N, say $N = 5$ (or whatever value was just used). This means that an "A" group is executing a "B" group's algorithm, without having seen the "B" pattern or any of the "B" boards—the only information they have to go on is what is written in the algorithm.

When the groups finish executing the algorithm they received, they are told to compare results with the group whose algorithm they executed. The students are instructed to discuss the discrepancies in their results with each other, as well as any difficulties they had following the instructions.

This lab provides students with a lot of insight into Steps 2 and 3 of The Seven Steps (with Steps 1 and 4 being quite helpful in being successful). Often, students overlook important details (*e.g.*, color, size, or orientation of their Legos) when writing their algorithms—this is the key skill of Step 2 and is critical to programming. Of course, finding the patterns in the Lego placement is key to writing a general algorithm, so students practice Step 3 as well. Sometimes students who struggle to find a pattern try to give instructions like "etc." "and then keep doing it similarly," or even "you know what I mean"—other groups struggle to follow such ambiguous steps and report their frustrations back to the group who wrote the poor algorithm. The fact that the algorithm is completely separated from writing code is crucial—the students can see that it is not a problem of remembering the syntactic details of a particular language, but rather that they are not being clear, and/or not finding patterns accurately.

4.4 Other uses

Beyond our use of The Seven Steps in our in-person courses, we have two different online introductory programming specializations in Java [1] and C [2], which make heavy use of The Seven Steps. In these courses, major examples are worked through with The Seven Steps, to go from a problem statement to working code. As with many in-class examples, we typically focus on Steps 1–5.

We are also seeing other instructors who are aware of The Seven Steps put it to use in their own courses. We are aware of a class at another University which makes use of the textbook [6] that we wrote for ECE 551. As previously mentioned, this textbook heavily emphasizes The Seven Steps.

5 EVALUATION

5.1 Student Anecdotes

One of the clearest indicators of the success of The Seven Steps in our classes is the stories that students tell us of their programming success. For example, one student in CompSci 101 wrote the following e-mail to Rodger towards the end of the semester:

> I just want to tell you that I tried the seven step method, and I worked on all of my code for one or two hours before I even looked at the computer. AND

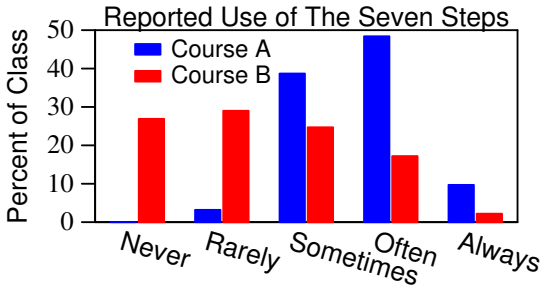


Figure 4: Students' use of The Seven Steps, by course.

> IT WORKED! I got all my code right on the first try! For the first time ever, I don't have to go to the help lab for hours on end. I just wanted to tell you how satisfied I am. Yay! Have a good day and thank you for re-teaching me the strategy.

We find this story particularly telling, as the student clearly has been struggling to write code on her own. However, when she follows The Seven Steps—and uses them to carefully plan before writing any code—she is able to write correct code on the first try. She is excited about the success and empowered by the ability to write working code without help.

A student in one of our online courses wrote the following:

> I have been programming for a couple of years. Learned from so many resources but none said how to write the algorithm, they just say you should write your algorithm first. The steps illustrated here are beautiful and definitely help to understand how to decompose a problem.

Even though this student has some experience programming, we find this story important—the student contrasts The Seven Steps with the approaches he has tried to use before. While he does not say what the prior approaches were or in what context they were used, he notes that "none said *how* to write the algorithm."

5.2 Survey Results

At the end of the semester, we conducted an anonymous, voluntary survey of both courses. We asked the students to report how often they used The Seven Steps, how useful they found various steps in the process, how important they thought various skills were (at start and end of course), and how confident they were in their programming ability. In total we received 132 responses from a total of 463 students (122 from ECE 551 and 341 from CompSci 101). Of these responses, 70 students reported their gender as "male," 61 reported their gender as "female," and 1 student did not report.

Figure 4 shows the percentage of students in each class who reported using The Seven Steps never, rarely, sometimes, often, or always. ECE 551 shows a remarkably higher rate of use of The Seven Steps than CompSci 101. One logical explanation is that The Seven Steps was emphasized much more strongly in ECE 551, as described in Section 4. However, other factors, such as the age and maturity of the students, may also contribute to these differences.

One set of these questions asked students to rate their perceived importance of four skills at the end and the start of the semester.

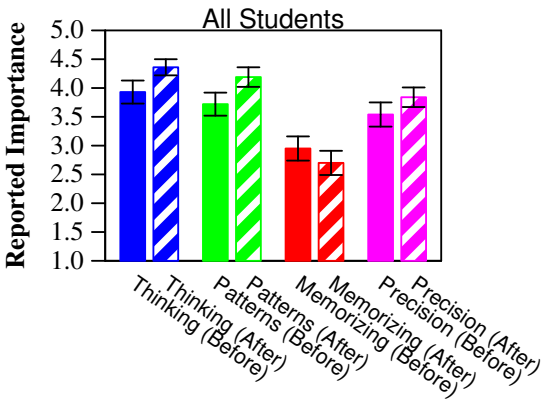


Figure 5: Mean student perceptions of skill importance

Three of these skills—Thinking before starting to write code, Finding patterns in a set of directions, and Being able to carefully and precisely articulate how you solved a problem—are important to programming, and we would hope students both learn them, and learn that they are important. The other skill—Memorizing example code that you have seen—is *not* important for programming, even though many students *think* that it is, based on prior academic success from memorization.

Figure 5 shows the average rating, on a scale of 1 ("not at all important"), 2 ("slightly important"), 3 ("neutral"), 4 ("very important"), and 5 ("extremely important"), for each of these four skills before the course (shown in solid bars) and after the course (shown in striped bars). We plot error bars for a 95% confidence interval around the mean on each bar.

The first observation is that the students' perception of what is important moves in the correct direction on all four skills: it increases for the three that are actually useful for programming and decreases on memorizing code. This change is statistically significant for the first two skills. Unfortunately, there is no control group available, so we cannot draw conclusions about whether these changes come from The Seven Steps in particular, or programming in general. However, The Seven Steps explicitly places emphasis on these skills. We also divided this data by gender, and found similar results and trends for both male and female students.

We also examined the relationship between how often students reported using The Seven Steps, and the level of confidence that they reported in their programming ability at the end of the course. While results were not statistically significant, those with more use of The Seven Steps reported higher confidence on average.

6 CONCLUSIONS

The Seven Steps provides students a step-by-step approach to work through programming problems, from problem statement to working code. It is important for instructors to integrate this technique into solving problems throughout the semester. We have had great success using this technique in a variety of courses, both in the classroom and online—and other faculty are beginning to adopt this technique. We highly recommend this approach to other CS instructors for use in their own classrooms.

REFERENCES

[1] Owen Astrachan, Robert Duvall, Andrew Hilton, and Susan Rodger. 2018. Java Programming and Software Engineering Fundamentals Specialization. (Dec. 2018). Retrieved December 12, 2018 from https://www.coursera.org/specializations/java-programming

[2] Anne Bracy, Andrew Hilton, Genevieve Lipp, and Liz Wendland. 2018. Introduction to Programming in C Specialization. (Dec. 2018). Retrieved December 12, 2018 from https://www.coursera.org/specializations/java-programming

[3] Catherine H. Crouch and Eric Mazur. 2001. Peer Instruction: Ten years of experience and results. *American Journal of Physics* 69, 9 (Sept. 2001), 970–977.

[4] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. *SIGCSE 2013* (2013), 579–584.

[5] Mark Guzdial and Barbara Ericson. 2007. *Introduction to computing and programming with Java: A multimedia approach.* Pearson Prentice Hall, Upper Saddle River, NJ.

[6] Andrew Hilton and Anne Bracy. 2015. *All of Programming.* https://play.google.com/store/books/details/All_of_Programming?id=-zViCgAAQBAJ&hl=en, online book.

[7] Marcia Linn, Alfred Aho, M. Brian Blake, Robert Constable, Yasmin B. Kafal, Janet L. Kolodner, Lawrence Snyder, and Uri Wilensky. 2010. *Report of a Workshop on the Scope and Nature of Computational Thinking.* The National Academies Press, Washington, D.C.

[8] Marcia Linn, Alfred Aho, M. Brian Blake, Robert Constable, Yasmin B. Kafal, Janet L. Kolodner, Lawrence Snyder, and Uri Wilensky. 2011. *Report of a Workshop on the Pedagogical Aspects of Computational Thinking.* The National Academies Press, Washington, D.C.

[9] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A Multi-national Study of Reading and Tracing Skills in Novice Programmers, An ITiCSE 2004 Working Group Report. *SIGCSE Bulletin* 36, 4 (Dec. 2004), 119–150.

[10] Dastyni Loksa, Andrew J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret Burdett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. *CHI 2016* (2016), 1449–1461.

[11] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, , and Tadeusz Wilusz. 2001. A Multi-national, multi-institutional study of assessment of programming skills of first-year CS students, An ITiCSE 2001 Working Group Report. *SIGCSE Bulletin* 33, 4 (Dec. 2001), 125–140.

[12] Charlie McDowell, Linda Werner, Heather E. Bullock, and Julian Fernald. 2006. Pair Programming improves student retention, confidence and program quality. *Commun. ACM* 49, 8 (Aug. 2006), 90–95.

[13] Esko Nuutila, Seppo Törmä, and Lauri Malmi. 2005. PBL and Computer Programming — The Seven Steps Method with Adaptations. *Computer Science Education* 15, 2 (2005), 123–142.

[14] Massachusetts Institute of Technology. 2018. Scratch website. (Dec. 2018). Retrieved December 12, 2018 from http://scratch.mit.edu

[15] Leo Porter and Beth Simon. 2013. Retaining Nearly One-Third more Majors with a Trio of Instructional Best Practices in CS1. *SIGCSE 2013* (2013), 165–170.

[16] Stephen Prata. 2013. *C Primer Plus* (6th ed.). Addison-Wesley Professional, United States.

[17] Topcoder. 2018. Topcoder Competitive Programming. (Dec. 2018). Retrieved December 12, 2018 from https://www.topcoder.com/community/competitive-programming/

[18] Carnegie Mellon University. 2018. Alice website. (Dec. 2018). Retrieved December 12, 2018 from http://www.alice.org

[19] Wikibooks. 2018. Five Steps of Programming. (Dec. 2018). Retrieved December 12, 2018 from https://en.wikibooks.org/wiki/The_Computer_Revolution/Programming/Five_Steps_of_Programming

[20] Laurie Williams and Robert Kessler. 2003. *Pair Programming Illuminated.* Pearson Education, Boston, Ma.

[21] Jeannette M. Wing. 2006. Computational Thinking. *Commun. ACM* 49, 3 (March 2006), 33–35.