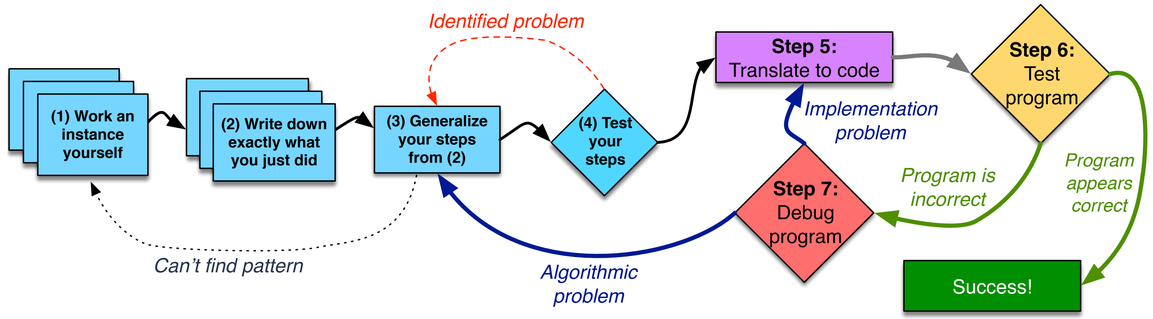
The Seven Steps

A Technique for Translation from Problem to Code

Andrew Hilton, Genevieve Lipp, and Susan Rodger
Duke University, ECE and CS Departments
adhilton@ee.duke.edu, genevieve.lipp@duke.edu, rodger@cs.duke.edu



Problem and Goal

Novice programmers often struggle to take a problem statement and turn it into working code [1,2]. This struggle arises from the need to both devise an algorithm to solve a class of problems, as well as to turn that algorithm into working code.

Several factors contribute to these difficulties:

- Students focus on memorization, not problem solving.
- Students lack a problem solving methodology.
- Finished examples do not show the process.

Our goal is to provide students with a methodology they can follow from problem statement to working code. In particular, such a methodology should:

- Recognize that students may struggle and provide direction for how to proceed in such a case.
- Work on any programming problem.
- Focus on problem solving and algorithm development.
- Include testing and debugging.

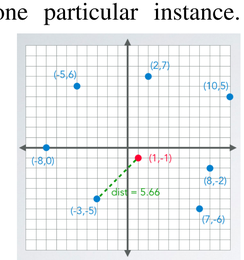
The Seven Steps

We call our methodology **The Seven Steps**, as it is a seven-step process for students to follow. The diagram in the upper-right corner shows these steps at a high level.

Step 1: Work an Example

The first step is to work *one* instance of the problem by hand. The student should not try to generalize to all possible parameter values at this point—they should simply solve one particular instance.

For example, if the problem statement is "Write a program that, given a set of points S and a point P, finds the point in S that is closet to P," then a student should select a particular set S (*e.g.*, $\{(2,7), (10,5), (8,-2), (7,-6), (-3,-5), (-8,0), (-5,6)\}$) and a particular point P (*e.g.* $(1,-1)$).



If a student cannot do this step, then either (a) they lack *domain knowledge* or (b) the problem is not stated clearly. Remedying (a) requires consulting a source of domain knowledge, while remedying (b) requires obtaining clarification on the problem.

Step 2: Write Down What You Did

The second step is to write down *precisely* what was done in Step 1. The most common way for novice programmers to get stuck at this step is to say "I just did it—I don't know how." In such a situation, they should work a slightly more complex example where the solution is not immediately apparent.

Another common problem in this step is that students omit instructions and/or write ambiguous directions. Such mistakes do not cause a student to get stuck here, but they cause problems that show up in Step 4. These problems are what instructors teach about with the common "Peanut Butter and Jelly" activity.

Step 3: Generalize

In Step 3, the student should generalize from the *specific* values they used in Step 1 to *any* values of the parameters. This step involves several components, such as generalizing specific values, naming values, identifying repetition, and making "almost repetitive" steps into uniform steps.

Students who get stuck here should repeat Steps 1 and 2 with different parameters to help see the pattern. We also encourage students to split their generalization into specific questions, and to make tables to identify patterns.

We note that this step is the hardest and that we explicitly tell students this is the hardest part when we introduce the process.

Step 4: Test by Hand

Step 4 is to test your algorithm by hand. Students should pick *different* values for the parameters that were not used in Steps 1–3 and execute their algorithm step by step with pencil and paper. If the answer is incorrect, they should revisit their generalization from Step 3.

Step 5: Translate to Code

Now students are ready to translate their algorithm into code. Any step that is too complex to translate into one or two statements should become its own function: students repeat the Seven Steps for that function and call it in this algorithm.

Step 6: Test

Step 6 is to test the program, by running actual test cases on the code. If a test case fails, students proceed to Step 7. If no test cases fail (and the student has sufficient test cases to convince themself that the code is correct), they declare success and are done.

Step 7: Debug

Step 7 is to debug failed test cases. We teach debugging as an application of the scientific method. Debugging will either identify a problem with the underlying algorithm (in which case students should return to Step 3), or with the translation to code (in which case students should return to Step 5).

Uses and Results

We use The Seven Steps extensively in ECE 551, as well as two Coursera specializations. CS 101 has begun to adopt the approach based on the successes observed in these courses. One CS 101 student sent the following e-mail about The Seven Steps:

> 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 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.

A learner in one of our Coursera specializations wrote:

> I have been programming for 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.

References

[1] 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. A multi-national study of reading and tracing skills in novice programmers, an iticse 2004 working group report. *SIGCSE Bulletin*, 36(4):119–150, December 2004.

[2] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, , and Tadeusz Wilusz. 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):125–140, December 2001.