Developing Applied Problem-Solving Skills in Computer Engineering Curricula

Dave Williams
Western Nevada Community College

Abstract

First-year computer engineering students are faced with the difficult challenge of developing a number of skills simultaneously. In addition to acquiring a mastery of the basic principles covered in each course, students must also exercise their problem solving muscles in order to successfully apply that material in the proper engineering context. Further, their solution must often be achieved through the use of a programming language or technical software package with a steep initial learning curve. A student working hard to retain subject matter may be severely handicapped by underdeveloped problem-solving skills and the need to learn a seemingly arcane programming language at the same time.

A successful instructional technique is presented which introduces complex topics through the use of a series of comprehensible examples of escalating sophistication that are easily implemented using standard computer engineering tools and techniques. As one example, public key cryptography (essential to the fields of computer engineering and security) is highly dependent on the generation of very large prime numbers. The rich topic of primality testing, immediately digestible at the simplest level by every student, is explored using a variety of algorithms in C++. Beginning with the simplest brute-force method, students are led through the process of progressive algorithm development. The speed and efficiency of their algorithms increase along with their understanding of the techniques as well as their command of the programming language. From the easily-implemented first approach, the module culminates with programs several orders of magnitude more efficient than students’ initial solutions. The progressive nature of this module provides students with the satisfaction of accomplishment while permitting them, working primarily in small groups of students with comparable abilities, to proceed at an appropriate pace as their proficiency develops.

Index Terms

Computer Programming, C++, Problem Solving, Algorithms

Introduction and Background

Introductory computer programming is a very difficult course for most students. By itself, algorithm fundamentals can be difficult to master, especially for students lacking problem-solving skills. But when the algorithms must be translated into source code, the difficulty increases exponentially. Add to that the need for class-wide comprehension of the math or
engineering principles that form the basis of the programming problem to be solved, and the sum total of the required material can easily become unmanageable. This situation is not unlike that of students studying a foreign language. For obvious reasons, new students are first exposed to simple vocabulary and sentence structure, the complexity of which increases along with their command of the language. No reasonable language course would begin with by studying an advanced scientific publication. Yet, many students struggle to grasp the basic nature of what their first computer programs are required to do long before they muster enough courage to even attempt translating their solution into code.

One key to successful instruction in introductory computer programming is to simplify the nature of the programming tasks until students have developed some ability in the language. However, many simple problems lack sufficient interest to capture students’ attention. Traditionally, beginning programming exercises start with some variety of the ubiquitous “Hello, world!” program and proceed to tasks such as temperature conversion from Fahrenheit to Celsius. While these are certainly valid exercises, they fail to draw many students attention to the true challenges of programming. A number of excellent instructional modules are available for these courses¹, but they require student mastery of fundamental material prior to their introduction. Preparing students for those and similar topics was the primary focus of this development effort.

In the author’s experience, beginning students generally prefer working in groups to working alone. “Far better to be lost as a group than all alone”, they agree. While assessing student performance usually requires some individual performance, much of the learning process is best accomplished in groups. The value of team-based learning in computer science is well established and documented². Students benefit from pooling their limited understanding and learning how to help each other. Team skills, important to their later success in industry, begin to develop.

Too many students try to write computer programs in one big chunk and then are overwhelmed when trying to debug them. A large program that doesn’t compile, or which fails to execute properly, often presents too many possible sources of errors to be useful in an instructional environment. A much better approach is to encourage students to complete the entire assignment by taking each small, functional step at a time. The instructor equates program development with the task of eating an elephant; both are best accomplished one bite at a time. By starting with a simple program that might only accept input data and display it back to the user, a series of progressive modifications and additions to that code ensures that students will be working with a program that functions and that any failure to compile or run is attributable only to their most recent modifications. Development and debugging is therefore learned and practiced in a methodical fashion.

Many courses rely on a series of progressive but unrelated programming exercises. This has the advantage of requiring a fresh approach to each individual problem. However, this pedagogy requires that students spend more time re-creating basic program overhead that might be better utilized perfecting algorithms. In lieu of unrelated or disconnected exercises, the author has developed a sequence of progressive exercises within a single-topic instructional module. This approach offers several advantages:

¹ "Proceedings of the 2004 American Society for Engineering Education Annual Conference & Exposition
Copyright © 2004, American Society for Engineering Education"
1) True beginners may start their programming effort at the most basic level, improving and augmenting their code as they progress to improve its performance and to add features;

2) More advanced students can begin at an appropriate level above that, retaining their interest by presenting a challenge better suited for their abilities;

3) By making a series of modifications to their earlier functional program, students are spared the necessity of starting each program from scratch, thereby allowing more time to think and work on algorithms, and also learn the proper technique for progressive program development, and;

4) Teams of students within the class may proceed at their own pace for better absorption of the material without hindering the progress of others.

A Prime Example

Upper-division programming students have acquired application-specific material in math, science, and engineering courses which may be used as the basis for interesting and challenging programming assignments. However, lower-division (and particularly first-semester) classes are composed of students with varying educational backgrounds. Programming topics which have sufficient complexity to provide rich algorithm possibilities but which do not require significant background in another technical area are highly desirable for these introductory courses. Further, if the basis for these topics can be linked to a branch of computer science or engineering, the students will express greater interest than if the topic has little or no relevance to their other studies.

Many techniques in modern cryptography rely on the use of very large prime numbers (typically, 512, 1024, or even 2048 bits or larger). Strong prime numbers form the basis for techniques essential to computer science, including public-key encryption and digital signatures. With only a cursory non-mathematical overview of these cryptographic functions, students will quickly understand the practical application of the module topic they are about to study.

The concept of primality is exceedingly simple and requires little math to understand. Integers less than 100 can usually be identified as prime or non-prime almost intuitively by most students. However, primality verification of larger integers is hardly intuitive. Handling cryptographically-useful primes is well beyond the range of introductory students, but the concepts are the same. In C++, use of the unsigned long long int data type allows consideration of positive integers less than $2^{64} = 18,446,744,073,709,551,616$ (more than eighteen billion billion). Although the smallest cryptographically-useful primes begin with this number squared, and squared again, and then squared a third time, $1.8447E19$ is sufficiently large to impress students with the scope of the problem and to fully utilize common computing resources with the simple algorithms in this topic.

For the purpose of algorithm comparisons in lecture, the number 10,007 (prime) is used and is referred to the “subject” (denoted as S). Its proximity to the round value of 10,000 simplifies
rapid comparison of algorithm efficiency. The concept of modular arithmetic and its applicability to prime number testing is presented in lecture, and its implementation in C++ is illustrated using a very simple program. Students soon understand that a result of 0 for the modulus operation $S \% n$ indicates that $n$ is a factor of $S$ and that $S$ is not prime.

As Simple as It Gets

Students are asked to consider the simple definition of a prime number (a number with only two unique factors, namely that number and 1) and propose the most simplistic, reliable, foolproof way to determine whether this integer (or any other integer supplied by the program user) is prime. Students have never disappointed the instructor and always propose the desired “brute-force” algorithm:

1) Test all integers $n$ such that $1 < n < S$

For the suspect value of 10,007, students understand that they will need to perform approximately 10,000 separate modulus operations to complete this task. They agree that testing every possible factor is very comprehensive and quite simplistic.

Better by Half

The discussion continues by considering the factors of a small non-prime number such as 14. By illustrating that factors come in pairs and that one such pair consists of the largest and smallest factors (here, 2 and 7). If 2 is pre-ordained as the smallest possible integer factor of interest (since 1 is excluded as a factor of every integer), then the largest integer factor of any non-prime $S$ can never be larger than exactly half of its value ($S/2$). As the paired factor of any integer between $S/2$ and $S$ will lie between 2 and 1, and since there are no such integers, testing any integer greater than $S/2$ is pointless. This recognition leads to a significant improvement in the previous algorithm:

2) Test all integers $n$ such that $1 < n < S/2$

It is evident that only about 5,000 separate modulus operations are required to evaluate 10,007 using this method (half required by the previous algorithm), and that the method is every bit as comprehensive as its predecessor.

The Functionality of Symmetry

Students at this point are reminded that the process of algorithm development is as much of an art as a science, as evidenced by the title and content of Knuth’s seminal work. Elegant algorithms are born from creative thought and the application of novel techniques. While a strong man may be able to open a locked door using sheer force, the feeble but crafty man may be able to open the same door by discovering a weakness in the locking mechanism. Each approach has its merits and drawbacks which should be considered in light of the problem.
All of the integer factor pairs of $S = 36$ are then placed on the board for the class to consider:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36</td>
<td>9 x 4</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>12 x 3</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>18 x 2</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>36 x 1</td>
</tr>
</tbody>
</table>

Figure 1. Factor Pairs of $S = 36$

It becomes obvious that the first term increases while the second term decreases, and that the column values mirror each other but “cross” in the middle. Students are asked to study this example and propose the next improvement to the algorithm. Someone will soon realize that testing from 2 to $S/2$ (18 in this case) involves considerable duplication, and that the testing of repetitive factors can be eliminated in this case by only testing from 2 to 6 (the square root of 36). The fact that pairs equally spaced from this “crossing point” consist of the same values can be used to further reduce the testing procedure and yields the next refinement:

3) Test all integers $n$ such that $1 < n < \sqrt{S}$

For the suspect number, this leaves just 100 modulus operations to perform. Students are reminded that so far, the problem space has been easily reduced from about 10,000 operations to 100 (a reduction of 99%) with more improvements to follow.

Better by Half Again

The focus is now directed to the actual operations necessary to the algorithm using a sampling of small suspect values starting with the first possible factor of 2. Students will soon recognize the futility of testing multiples of possible factors already considered. In particular, if 2 is not a factor of $S$, then no other multiple of 2 (commonly known as the set of even numbers) can be a factor of $S$. The algorithm has now evolved to the form:

4) Test integers $n = 2, n = 3$, then $n = n + 2$ for $1 < n < \sqrt{S}$

By eliminating half of all remaining numbers, only about 50 values of the original 10,000 are left to test without compromising the integrity of the process. The algorithm is presented this way to assist the subsequent coding efforts. Since 2 and 3 are the only consecutive primes, isolating the test for $n = 2$ allows students to test every other number beginning at $n = 3$. 
More Effort for Less Results

At this point, further development of this algorithm begins to require more force than elegance. Most of the improvement has already been achieved with minimal programming skill and students are now facing a problem that is but a tiny fraction of its original magnitude. However, the remaining improvements will be less significant and will require far more effort.

If eliminating every other even number reduced the problem space by 50%, someone usually suggests that subsequently removing every multiple of 3 will eliminate one-third of the remaining possible factors. Confirmation of this suggestion is left to the class as a whole until someone realizes that alternating (even) multiples of 3 have already been eliminated when multiples of 2 were stricken, so the actual improvement is only one-half of one-third of the remaining values. This removes approximately 17% of the 50 remaining values, or another 8 possible factors. The algorithm now becomes significantly more complex if every multiple of 2 and 3 are eliminated:

5) Test integers \( n = 2, n = 3, n = 5 \), then \( n = n + 2 \), then \( n = n + 4 \),
   Then \( n = n + 2 \), then \( n = n + 4 \), etc. for \( 1 < n < \sqrt{5} \)

Beginning at 5, the remaining factor candidates are separated by alternating steps of 2 and 4 (2 – 4 - 2 - 4 - 2 - 4 - 2 -…). The steps of 4 skip over integers that are multiples of both 2 and 3. Only 42 factors now remain to be tested for the suspect value of 10,007, but at the cost of a more complex algorithm.

Maxing Out

This process can be repeated for other small factors as well. For example, eliminating multiples of 5 (the next smallest factor not already eliminated) will further reduce the problem space. However, students observe how rapidly the complexity of the algorithm is increases while the benefits obtained from this complexity are becoming smaller and smaller. For example, eliminating the remaining multiples of 5 for the suspect prime 10,007 will only remove the numbers 25, 35, 55, 65, 85, and 95 from evaluation. With this improvement, only 36 modulus operation tests will remain. However, to accomplish this process, the pattern of consecutive possible factors follows the sequence 2 - 4 - 2 - 4 - 6 - 2 - 6 - 4 starting at 11. The values of 2, 3, 5, and 7 must be tested separately because no repeating pattern exists which includes them.

Coding of these two special-sequence examples is not trivial. During algorithm discussions, no specific coding examples or instructions are provided. Students are expected to devise the code within their teams (see below). The instructor’s C++ solution for these beginning programming students utilizes a standard “unrolled” for loop with the index value \( n \) (for the operation \( S \% n \)) incremented inside the loop as well. For the final algorithm discussed above, the outer loop is incremented by 30 (the sum of all steps within each repeating pattern) while \( n \) is further incremented by the steps in the pattern inside each iteration of the loop.

For the simpler algorithms, essentially all of the computational overhead was consumed by the modulus operations. However, once the algorithms begin to incorporate the special-sequence
method, a significant amount of CPU time is required for loop management, and the resources devoted to this function increase as the sequences become more sophisticated. When the number of operations saved by eliminating factor candidates is exceeded by the increased overhead of the loop functions, program speed decreases. Smarter is not always faster.

Over the Top

The process of factor elimination can be continued as far as the programmer desires, until no remaining values less than the square root of the suspect value remain. However, the algorithm becomes unwieldy with any further “improvements”. Students have always been offered the opportunity to earn extra credit if they could extend the algorithm to exclude multiples of the next factor possible (7). After many sections of this exercise, not one student has ever attempted to do so. The similar sequential pattern would be significantly longer than in the last illustrated case, greatly increasing the possibility of an algorithm or programming error which would compromise its accuracy and reliability.

Of course, in the limit, this approach will end up testing the suspect prime with the entire set of primes less than the square root of that value. Students are asked to devise an alternate algorithm that first generated the set of all primes less than $\sqrt{S}$ and then tested each against $S$ (pre-calculated look-up tables are not permitted). The well-documented Sieve of Eratosthenes (quite similar to the special-sequence algorithm used here) is presented as an option. Most students soon realize that generating all of these primes first and then testing each against $S$ would take longer (and therefore run slower) than directly evaluating each possible factor in order, even if some unnecessary tests are performed in the process. Since the presence of a single factor denies primality, testing should stop as soon as the first factor is discovered. The complete list of possible factors will only be required for prime numbers, as non-primes will very likely be uncovered long before the list is exhausted. Students should be encouraged to consider why the probability that a test value will be a factor of $S$ decreases as the test value increases; this affirms that smaller numbers should always be tested before larger numbers.
The progressive nature of this algorithm development exercise provides a first-hand illustration of several critical relationships in algorithm development which are depicted graphically in Figure 2. When programs become too complex, valuable CPU time is consumed by overhead (such as calculating possible factors that are never tested) in the same manner that simple “brute-force” programs often perform unnecessary calculations. At some point, the optimal balance of sophistication and execution speed is achieved, and adding further to the algorithm detracts more from the program than contributes. The other potential liabilities of overly-sophisticated programs should also be explained. Complex code is harder to debug and is more likely to have a computational error that may creep in when some small aspect of the proper algorithm is overlooked. What’s left out is sometimes more significant than what’s added; “Any intelligent fool can make things bigger and more complex. It takes a touch of genius, and a lot of courage, to move in the opposite direction.” (Albert Einstein) In engineering parlance, this is commonly known as the KISS (“keep it simple, stupid!”) principle. Additionally, complex code requires more development time, and spending hours optimizing code to run several seconds faster is rarely beneficial unless that code lies at the heart of a much-repeated function or operation.

Making it Happen

Once all six levels of algorithm sophistication have been discussed in lecture, teams of 2 or 3 students are formed to begin writing code during lab sessions. The instructor allows students to form their own teams, with an occasional student electing to work alone. Based on their expressed preference and the instructor’s assessment of their abilities, the groups are assigned both a starting point and a minimum expected objective (usually, at least through algorithm 5). Several consecutive lab periods are devoted to the exercise, and students are never discouraged from working on their code at other times between labs.

Students are shown how to incorporate timing functions in their programs to evaluate comparative run-times for larger suspected primes. This provides a solid basis for meaningful comparison between subsequent versions of their programs and those produced by other groups (which are shared with the class after all team and personal identification have been removed). Affording students the opportunity to review the work of other groups encourages the best effort of all teams and exposes students to a variety of coding styles. When they experience difficulty following those programs, they learn to appreciate the importance of proper stylistic coding. Most students will be amazed at how long their programs will run for some suspected primes, even on ultra-fast machines, and the speed enhancement attributable to algorithm improvements will be even more pronounced.

Once the module has concluded, the instructor has been known to incorporate a lightly-disguised version of these same algorithm fundamentals on a subsequent programming exam. Students are asked to write and debug a short program during an in-class exam to calculate the greatest factor (other than itself) of an input integer. This is simply the primality algorithm in with a twist. Solutions which begin by testing the largest possible factors less than S (which is perceived by some students as the appropriate place to start testing) and work down from there generally exhibit longer run times for most values of S. Sharp students have learned that half of the possible input values will be even and therefore divisible by 2, and that the factor paired with 2 will be the largest factor (see Figure 1). Starting at n = S/2 will produce the same result for even
values of S just as quickly, but decrementing n by 1 from that point (also a common practice) will duplicate the fundamental shortcoming of the simple algorithm (searching the problem space for numbers whose paired factor is not an integer). The next most likely factors are 3 and its pair, and many integers between S/2 and S/3 will needlessly be tested if the process begins at S/2. Starting low is still the best approach, and those students who realize this have learned the material well.

Summary and Conclusions

Both the similarities and differences between the primality project and this “greatest factor” programming exam question make them a synergistic package. The academic value of this module may be measured quantitatively through exam scores; high scores will indicate solid mastery of the subject material. Meaningful assessment would require a comparison between students exposed to this approach and those who were not. As there was no parallel course taught by the same instructor with another pedagogy as the sole difference, that data is not available. However, the author is willing to offer an anecdotal, qualitative assessment of skill mastery obtained by contrasting the performance of both groups in subsequent programming courses. Assuming all other things having been equal in their academic preparation, students who have worked through the progressive prime number algorithm development module generally seem more cognizant of efficient algorithm construction than are student who did not. However, it must be noted that all other things are almost never equal.

From another perspective, student feedback indicates that the module described in this paper is well-received. The motivation and challenge provided by an interesting project of progressive intensity in an appropriately-paced group-learning environment is reflected in positive student comments. If only for that reason, the use of this module is believed to be of significant instructional value. Retaining students in the program is directly linked to retaining their interest, and by their own evaluations, this module accomplishes that goal. Similar progressive modules in other engineering and computer science-related topics are currently being developed and evaluated for future dissemination.

Bibliography


Biographical Information

DAVE WILLIAMS is the Engineering Instructor at Western Nevada Community College in Carson City, Nevada and a collaborator in Computer Vision research and development projects at other institutions. He is a Registered Professional Engineer and has practiced as a consulting engineer for 30 years in the field of antenna design and analysis, computer simulation and modeling, and electromagnetic compatibility. Holds the Ph.D. degree in Electrical Engineering from the University of Nevada, Reno.