Incorporating Problem Solving into Programming Classes

Robert Lamey
Computer Technology Department, Purdue University
West Lafayette, IN

Abstract

Problem solving involves far more than the ability to plug numbers into a formula and looking to a calculator to resolve an answer. The real world presents problems, described in words, that require creative applications of the more fundamental principles taught in physics, mathematics, and business classes. The unpopularity of "word problems" and the difficulty in teaching creative thinking have generally led educators to avoid problem solving in favor of equation solving. This paper demonstrates that methods for finding creative solutions to novel problems can be codified and taught within the structure of a programming class.

Keywords: Problem solving; programming; heuristic; word problems

1. THE UNFULFILLED PROMISE

Approximately twenty-five years ago, educators decided to introduce the use of calculators into primary and secondary schools. Calculators were seen as the coming technology and teachers recognized the need for students to learn to use the tools of the electronic era. Calculators brought the promise of taking the drudgery out of arithmetic. Students would certainly be more motivated to learn mathematics once long division, multi-digit multiplication, and square roots were handled by the black box.

But the real benefit would be the timesavings that the calculator would bring. If students no longer spent hours, or days, or weeks learning to do by hand the operations a calculator could perform in milliseconds, there would be more time to learn to solve problems. Problem solving was the goal. Educators argued, correctly, that there was little advantage in learning to divide, to calculate percentages, or to solve equations if the information could not be applied. Passing arithmetic manipulations (and later the solving and graphing of equations) to the calculator would not only eliminate dulling repetition but would create time for the real purpose of studying mathematics. So the calculator came but the problem solving did not.

A significant element that contributes to the avoidance of problem solving is the level of difficulty. Calculating 19% of 2000 is easy. Determining how much money you will owe at the end of the year if you only pay the minimum on your Visa card bill is hard. Finding the sine of 30° is easy. Determining the height of the tree in your backyard is hard. Plugging numbers into a formula then punching those numbers into a calculator is easy. Developing the equation that fits the conditions of a given problem is hard. Unfortunately, real world problems are seldom stated in ready-for-the-calculator form. Real world problems appear in a series of sentences that must be understood and interpreted as a sequence of mathematical operations.

Years after the introduction of the calculator we find a significant number of college students who cannot calculate weighted averages including their own GPA; cannot perform unit conversions; cannot determine the area of a non-rectangular surface; or cannot determine the elapsed time between two events. Many are in a mathematical limbo where problem solving can only begin after someone else has developed the equation ready for their calculator's number crunching. Ironically after someone else has developed the equation, the problem is generally solved by a computer.

We can speculate about the reasons for the current lack of problem solving ability in a large fraction of college students. We might decide that the fifth grade teacher who taught fractions for years was adept at teaching fractions but unable to teach problem solving once calculators took over the job of manipulating fractions. We might decide that learning to use the calculator took just as long as learning to work problems by hand and there was no time saving. We might decide that the algebra teacher found that "word problems" were so unpopular and so difficult for students that it was easier, less troublesome, and provided more job security to have students only learn techniques for solving equations.

Regardless of the root of the problem, the situation exists. The question is whether to confront the issue or to ignore it for another four years when it will turn into an employer's problem instead of an educator's problem.

2 ADDRESSING THE PROBLEM

The student who learns to use the quadratic formula perfectly but cannot construct an equation from the information given in a problem has little more than a fact suitable for forgetting a week after the final exam. Similarly a student who learns to write function calls and looping structures perfectly but cannot use these tools to solve a problem has missed the point of learning to program. Programming classes are not only an excellent venue for teaching problem solving but there is little reason to program at all if not to solve a problem. Writing conditional statements and constructing classes of objects are not objectives in themselves but are tools used to solve problems.

Unfortunately the programming instructor is in the same position as the algebra teacher. It is very easy to teach the syntax for a *while* loop; it is more difficult to teach how the *while* loop is programmed to do something useful. It is simple to demonstrate how to print "Hello World!" on the screen 10 times. It is another matter to create the simulation of a game that is played 1,000,000 times to determine the odds on an outcome. It is simple to teach the syntax of an *if* statement but to use the conditional to implement a successive approximation algorithm is more challeng-ing.

3 BLUEPRINT FOR PROBLEM SOLVING

Once the decision is made that it is necessary for students to be able to direct the knowledge they have acquired toward finding solutions to new, real-world problems, the next question is how can the skill be taught? Traditionally, if the issue of problem solving is approached at all, the effort entails the use of Boolean logic or flowcharting. Each of these has its place but falls short of leading to a solution to a problem. Boolean logic is useful in creating a compound test condition for a *while* loop but if the question is how fast must a satellite travel to stay in orbit, the application of the rules of Boolean algebra is not at all clear.

Flowcharts are useful for describing the solution to a problem so the solution can be converted into programming but if a method for determining the satellite's speed cannot be found, a flowchart cannot be constructed. A flowchart is a description of a solution that has already been discovered. Flowcharts, like pseudo-code, clarify and organize but are of little use in developing creative solutions. Another method for teaching problem solving is needed.

The first rule in problem solving is to be clear in defining the objective. A vague description of the objective will generally foil any project. This can be applied to teaching problem solving as well as any other problem. Teaching problem solving means teaching a directed form of creative thinking. Defined more narrowly, **teaching problem solving means teaching general methods for discovering solutions to problems that the student has never been taught to solve.**

This proposition opens the debate as to whether creativity can be taught or whether it is some undefined and indefinable element of genius that some have and others do not. While it is obvious that some people are more creative than others, the suggestion here is that virtually everyone's creativity can be enhanced. Experience makes this proposition intuitively satisfying. Aspiring writers are told to write. The ability to attack mathematical problems increases with the amount of mathematics studied. Beethoven's early compositions are good but do not compare to his more mature work.

Creating Experience

Isaac Newton is often quoted as saying, "If I have seen farther (than you and Descartes), it is by standing upon the shoulders of Giants."[1] The applicable idea is that the *Philosophiae Naturalis Principia Mathematica* did not suddenly occur to Newton. Instead insight generally comes in small steps that build on previous work, either your own or the work of others. Developing the potential for insight is the goal in teaching problem solving methods.

This concept can be implemented in a programming class by having students build a repertoire of solved problems that begin at an elementary level and grow in complexity. One way to find the area of a trapezoid is to frantically search math books or the Internet for a formula but it is more insightful to realize that a trapezoid is a combination of a rectangle and two triangles. Adding the areas of the familiar figures leads to a solution built on already known principles and without the formula for a trapezoid. From here it is a small step to the realization that the areas of many rectangles can be added to find the area under a curve and integral calculus becomes a tool for use in future problems.

Calculating the time required for light from the sun to reach a planet is daunting unless it is the final problem in a series of programs that first require unit conversions, then the more difficult conversion from a

[1] Letter from Newton to Robert Hooke, Feb. 5, 1675/1676

large number of seconds to hours, minutes, and seconds. Finally, adding the fact that distance is equal to rate multiplied by time, the difficult problem is discovered to be little more than a new arrangement of simpler, familiar ideas. There is no substitute for experience and in problem solving that means a wide background of previously solved problems of increasing variety, intricacy, and difficulty.

However an appeal to experience is less useful to a student with little experience in problem solving. There is a need for a second method for teaching problem solving that can be taught to the novice and is used in parallel with the gradual building of experience. In teaching programming over the past 10 years I have found that the techniques that lead to the solutions of problems can be codified. This is not to imply that creativity can be reduced to a formula but rather that if certain steps are followed gaining insight to the solution of a problem can be enhanced.

The Heuristic

Rule One: The first and possibly most important rule in problem solving is to be clear about the information available for attacking the problem, and to be clear about the objective. Overlooking this starting point either leaves students without the necessary information for solving the problem, or without direction for lack of a well-defined endpoint. Despite the obvious importance of this rule, it is often ignored by students who have developed the habit of searching the problem description only for numbers that can be plugged into a formula. This rule is so critical that I encourage students to rewrite problems in the form of a list of usable information and objectives. I also create this list before I begin to program my in-class demonstrations.

Rule Two: The second rule is to determine if there is any relevant additional information required for solving the problem. This may involve finding a value not given in the problem such as *PI*, the radius of the earth, or Plank's constant. The required value may be information that the user of the program should be prompted to supply.

Rule Three: After information has been gathered and listed in a usable form, relationships are sought that combine the data in ways that lead in the direction of the final goal. Sometimes this will involve experiments where the data is combined in any possible manner and afterward a decision is made whether this produced a situation closer to the stated goal.

Consider applying these three rules to the problem of determining the area of a trapezoid. Looking up a formula would solve the problem but do little for the development of problem solving skills. An alternative approach is to examine the information given in the problem. Here the information is limited: there is a trapezoid and the objective is to find the area. Rule one suggests that clarity about the shape of a trapezoid would help. In this case clarity implies the need for a sketch of a trapezoid, which in turn shows that a trapezoid is the joining of three simpler figures. Rule two indicates that some information about the dimensions of the trapezoid should be supplied by the user.

Once the user has been prompted to supply the values for the height and two bases, this data can be used in more commonly known formulas for rectangles and triangles. It is the discovered relationship between rectangles, triangles, and trapezoids that leads to the solution of the problem.

Rule Four: The example of the trapezoid leads naturally to the fourth rule. Whenever possible a picture should be drawn. While the advantages are obvious in problems involving geometric figures the rule applies far more widely. The process for determining the square root of a number (or third, fourth, or fifth root) using the successive approximation method can be made far clearer by using a number line. If the number is 200, the picture shows that 100 can be guessed, squared, and checked. Since 100 squared is too large the next

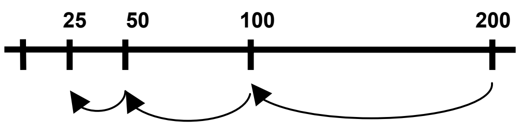

Figure 1

obvious guess is 50. The process can be repeated as long as required until a reasonably close solution if found. (Figure 1)

The problem of finding the speed necessary to keep a satellite in orbit simplifies greatly if a picture of the earth and the orbit is drawn. The picture (Figure 2) shows that all of the values needed for the use of the Pythagorean theorem are available. If the time is kept short, e.g. 30 seconds, the length of the curve of the orbit is approximately the same as the straight-line distance the satellite would travel if there were no gravity. The fall due to gravity for 30 seconds, is easily calculated using $d = .5gt^2$. The altitude of the orbit is a user-supplied number.

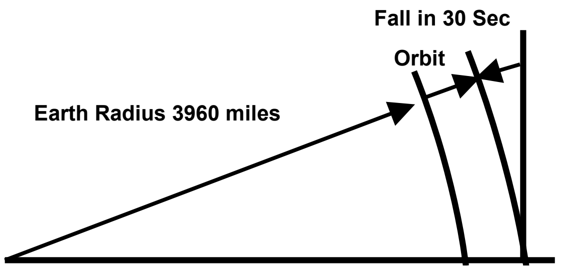

Figure 2

Expanding on the experience of one use of the successive approximation method, students can discover a method for solving unsolvable equations such as fifth order equations. The additional information required is the fundamental theorem of algebra, that is, the number of solutions to an equation is equal to the order of the equation. If we assume for the moment that all the solutions are real, a graph can be drawn of a general fifth order equation. The picture itself (Figure 3) leads to the observation or insight that for increasing values of x, the sign of y will change when we pass a solution to the equation. Evaluating the equation for values around

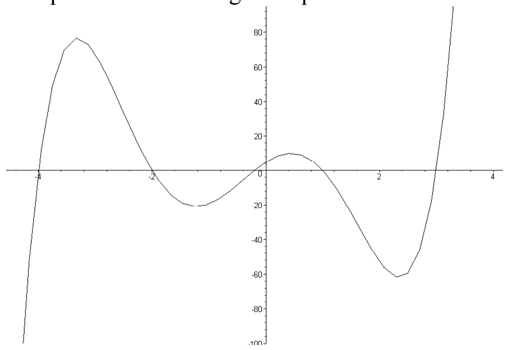

Figure 3

each crossing zeros in on the solution by the process of successive approximation learned while generating square roots.

Even seemingly intractable problems such as finding the minimal path connecting a set of points, commonly called the traveling salesman problem, can be solved after drawing a picture of a representative path. In this case the options available at each point, as the illustration is drawn (Figure 4) give clues to how the looping structures in the program should be built.

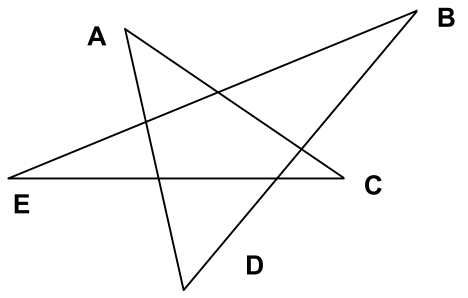

Figure 4

Rule Five: The ability of a modern personal computer to make many millions of calculations per second provides the problem solver with incredible power. This power can be applied in systematic or brute force fashion. The fifth rule is to search the problem for repeatable operations that lead to the objective.

In the case of the square root problem the successive approximation process of guessing at the square root of a number will eventually lead to the square root of the number. By strategically guessing, testing whether the guess is too high or too low, adjusting the guess accordingly, and then repeating the sequence, a solution of can be found. Of course it will not produce the exact value for the square root but the successive approximation method will determine a root of arbitrary accuracy.

Determining the solution to an infinite summation requires mathematical ability beyond that of the ordinary college student. However it is a simple matter to program a computer to repeatedly generate numbers in the series and total those numbers. Only in rare cases will the true summation differ significantly from the sum of the first 1000 terms.

Calculating probabilities can confound even mathematicians. On the other hand programming a computer to simulate the rolling of a pair of dice ten million times and counting the number "sevens" rolled involves a trivial loop and very little time. Again, the result is not mathematically pure but we can produce as much accuracy as we need by adjusting the number of rolls. Building on the experience of programming the odds of winning or loosing a game of craps, my students have found that there is little difficulty in determining odds that would be quite difficult to handle theoretically. Calculating by hand the odds of landing on Indiana Avenue in the first four rolls of a game of Monopoly is intimidating. Using nested loops to simulate the first four rolls of one million games of Monopoly is simple (assuming chance cards that send the player to another square are ignored).

The traveling salesman problem has no mathematical solution. On the other hand a computer can be programmed to try every possible path connecting the points and to keep track of the shortest path. If the salesman has to travel to 10 cities there are 362,880 different paths. These represent more calculations than anyone would like to make by hand but are easily handled by a Pentium III powered computer. The 10^{17} different paths for 20 cities is another matter.

Rule Six: The sixth rule again addresses the need for clarity but in this case the focus is not the initial information or the objective. Instead rule six requires the problem solver to be clear about how many times a given operation is performed. When constructing repeated operations, the problem solver must decide whether the operation should be placed inside or outside of the loop.

In calculating the square root of a number by successive approximation, the initial guess for the square root and the generation of the initial increment for raising or lowering the guess is performed only once. This indicates that the operation is outside of the loop. The testing of the guess to determine whether it is sufficiently close to the correct number, and the generation of new guesses and new increments are handled inside the loop.

This rule takes on added weight when nested loops are involved. Consider a program that will investigate the results of perfectly shuffling a deck of cards several times. The outer loop controls the number of times the deck is to be shuffled. The inner loop performs the shuffling of the cards. Before entering the outer loop it is necessary to order the cards. This ordering takes place only once and is written outside of the nested loops. The outer loop counts the number of times the deck has been shuffled, displays the order of the cards after each shuffle, and asks the user whether to shuffle again. The inner loop performs the perfect shuffle. This involves dividing the deck in half, alternately taking the top card from each half, and generating a new deck from the interleaving of the cards.

Rule Seven: Solutions to complex problems can often be determined by limiting the original problem and searching for a solution to a special case. This approach may lead to a partial solution but a limited solution is usually better than no solution at all. Often the creation of a special case solution generates the insight for a more general solution.

Again the square root problem provides an example. The method for generating the new guess for the square root in each cycle that works for the numbers one and greater may not work for numbers from zero to one and quite likely will not work for negative numbers. However, writing a program that will solve the limited problem for numbers greater than one, can suggest where the routine needs to be adapted for the zero to one case.

This technique is commonly used in physics classes where problems are simplified by limiting the forces allowed to impact an experiment. A program that calculates the distance traveled by a falling object can have many levels of complexity. If only a constant force of gravity is considered the solution is to plug the free-fall time into the equation $d = .5\ gt^2$. This limited solution provides some information about the problem but it ignores wind resistance and is more properly the solution for an object falling through an evacuated tube. It also ignores the fact that the acceleration due to gravity is not constant but varies with altitude.

The seven rules listed above are not ordered by any perceived importance. Instead they are listed in the order that they are usually needed in solving a series of problems of steadily increasing difficulty. Similarly there is no implication that these seven represent a complete list. The list could be expanded to include advice to work with a concrete example before attempting a general solution. Rules for handling probability problems could be added. The rules could include the use of approximations, as in substituting a straight line for the orbital curve in the satellite problem. The intention is to show that likelihood of discovering a solution to a previously unseen problem can be enhanced by a proper approach to the problem.

4. IMPLEMENTING THE SOLUTION

Once a plan for teaching problem solving has been developed the next step is to integrate that plan into a course that has the teaching of a programming language as its primary objective. Fortunately the two objectives mesh well together and can be taught in parallel. The technique that I have used for the past ten years requires a computer for the instructor and some means for allowing the class to view the instructor's monitor. Typically this involves either a projection system or a link system to export the instructor's monitor image to the students' computers.

Each class centers on a single feature of the language and an example that uses the feature in solving a problem. If the topic of the day is construction of a *for* loop, the class begins with an explanation of what a *for* loop does and the syntax for implementing a *for* loop. Then a problem is presented that can be solved by the use of a *for* loop. The question of whether it is better to take $1,000,000 or a penny the first day and each day for a month getting double the money of the previous day would be a typical example.

The lesson continues on the whiteboard using the rules described above. The information given in the problem description and the objective are itemized. Then it is noted that the solution requires a repeated operation, namely the doubling of the number of pennies received on the previous day and the addition of that amount to a running sum. By being clear about which actions are repeated and which actions occur only once, it can be determined that the first day's money must be initialized only once to a single penny and the variable used to hold the running sum must be initialized to zero.

Once the pseudo-code is written the focus changes to the computer and the code is written in real-time to implement the already developed plan. Writing the program in class ensures that every feature of syntax to the smallest punctuation must be included and explained. It also demonstrates the implementation of the pseudo-code in the language being studied. There are further advantages in that previous lessons are reviewed in the writing of the program. In this example the use of variables and mathematical operators is required. Another advantage is the opportunity to examine the "what if" scenario. What if the doubling is continued for three months instead of one? What if the amount is tripled each day instead of doubled? What if the initial penny is replaced by the value of a bank account and doubling is replaced by the daily compounding of interest?

Finally the program becomes part of the repertoire of problems that the student has encountered and solved or has seen solved. These solutions serve as a base for solving similar but more complex problems. For example, determining the odds of winning a game of craps on the first roll become a launching point for the more general problem of calculating the odds of winning if the game is played to conclusion.

5. EVALUATION

If it is accepted that:

1. problem solving skills are valuable and should be part of a student's education,
2. problem solving skills can be enhanced by building experience and applying a set of rules,
3. that a programming class is a reasonable venue for teaching these skills,

the next issue is how to evaluate whether the skills have been learned. Unfortunately the evaluation of problem solving skills does not lend itself well to standard time-based testing. Instructors who include problem solving as part of a lab practical exam often find that students who discover the method for the solution score 100% and those who don't score zero. The problem is solved or it is not. If the problem involves multiple steps the student who cannot discover the solution to the first step has no opportunity to attack later steps. Time-based testing tends to leave instructors with no gradient.

The alternative is to remove the time requirement. Rather than using lab assignments I have developed several sets of problems that are combination homework/lab assignments. The problems are published on the class website and students have unlimited time to work on the programs with the condition that at least one program must be completed each week. Completed programs are demonstrated to the instructor in lab. Experience has shown that this creates an even split between the time the instructor spends evaluating students' work and helping students over problems of syntax and logic. For the student this means that there is no time lag between the time a program is submitted and the evaluation of his work. Students receive instant feedback regarding improvements that could be made or how alternative strategies could be used. This one-on-one interaction also provides an interesting advantage for the instructor. I have found that by the fourth lab session I know all my students by name.

The initial programs are quite simple and usually involve unit conversions or the calculation of percentages. Later programs can involve parsing sentences or determining probabilities. Programs are accepted as complete when they correctly perform the required task. Until that time the program is in progress with no penalty for a program that is demonstrated but does not work correctly. Scoring is based on the percentage of the assignments that are completed in the semester. This allows students a great deal of control over their lab grade. Students who find programming difficult can match students who already have some skill at problem solving or programming by merely dedicating more time to the problems.

6. CONCLUSIONS

If students are to be properly prepared not only for jobs in the information technology sector but for dealing with a wide variety of everyday situations, their ability to creatively apply basic principles to novel problems must be advanced. There would be significant advantages to addressing this issue in both primary and secondary schools. The argument can reasonably be made that there is little advantage to calculating percentages or solving equations if these principles are not applied to everyday problems. Nonetheless "word problems" are avoided because of their difficulty and are unpopular with both students and teachers.

Students have difficulty in dealing with the frustration of finding their calculators ineffective. Even at the collegiate level students are often at a loss as to how to begin to solve a problem if they are not provided an equation. Teachers find that teaching problem solving is no less difficult than learning the skill. Books to guide the process are few and generally rely on flowcharts and Boolean logic that are of little aid in the creative process. On the other hand teaching one more method for solving simultaneous equations is straight forward, mechanical, and produces much less resistance from students.

Similarly, teaching students to write looping structures, function calls, and conditional statements is straight forward, mechanical, easy to learn, and easy to teach. However, if these programming constructs are not applied to solving problems there is little reason for including programming in the curriculum. Without the applications the syntax for a *while* loop is merely a fact without purpose to be forgotten as soon as possible after the final exam.

Fortunately some structure can be brought to the vague area of creative thinking. By applying the heuristic rules described above the chances for discovering a solution to a previously unseen problem can be improved dramatically. By generating a repertoire of previously solved problems, new problems can often be found to be variations on a familiar theme.

7 REFERENCES

Arzarello, F., Bazzini, L., Chiappini, G., 1993, "Cognitive processes in algebraic thinking: towards a theoretical framework", Proc. *PME XVII,* Tsukuba, Japan, I, 138-145.

Baclace, Paul E., 1992, "Competitive Agents for Information Filtering", Commun. ACM 35(12, December), 50.

Chiappini, G., Lemut, E., 1991, "Construction and interpretation of algebraic models", Proc. *PME XV,* Assisi, **I**, 199-206.

Cypher, Allen, 1991, "Eager: Programming Repetitive Tasks by Example", CHI '91 Conference Proceedings. (Eds: Robertson, Scott P; Olson,

Frensch, Peter A, Funke, Joachim, 1995, <u>Complex Problem Solving: The European Perspective,</u> Hillsdale, NJ

Langley, P., Simon, H.A., Bradshaw, G.L., & Zytkow, J.M., 1987, "Scientific Discovery: Computational Explorations of the Creative Processes", Cambridge, MA: The MIT Press.

Larkin, J.H., & Simon, H.A., 1987, "Why a diagram is (sometimes) worth 10,000 words", Cognitive Science, 11, 65-100.

Okada, T., & Simon, H.A., 1997, "Collaborative discovery in a scientific domain", Cognitive Science, 21 (2), 109-146.

Zhu, X., & Simon, H.A., 1987, "Learning mathematics from examples and by doing", Cognition and Instruction, 4, 137-166.