

Incorporating Problem-solving Patterns in CS1

David Reed

Department of Mathematics and Computer Science
Dickinson College
Carlisle, PA 17013

reedd@dickinson.edu

1 INTRODUCTION

In [Wall96], Wallingford describes an approach to introductory courses that is based on programming patterns, i.e., algorithms or problem-solving approaches that can be applied to various applications. By focusing on patterns such as "Input-Process-Test" or "Process all items in a collection", students reason at a higher-level of abstraction when solving problems. In addition, code schema can be provided which apply to certain patterns, and these schema then serve as frameworks for program development. (See also [Rist89], [Coad92], and [GHJV95].)

Closely related to the patterns approach is the use of themes in a programming course. Selecting a particular idea (such as self-reference [Astr94]), methodology (such as formal specifications [MH96]), or application domain (such as databases [AR95]) provides a framework for learning new techniques and concepts. Once a concept has been studied in one context, new applications which similarly utilize that concept can be understood more easily.

This paper describes the use of a particular problem-solving pattern, binary reduction, as a recurring theme in the CS1 course. Other problem-solving approaches, such as divide-and-conquer or generate-and-test, could similarly be used. By introducing problem-solving patterns early in the course and then revisiting them in different contexts, students learn to look for common characteristics in problems, and to use an existing solution as a framework for solving related problems. Perhaps more importantly,

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.

SIGSCE 98 Atlanta GA USA

Copyright 1998 0-89791-994-7/98/ 2...\$5.00

understanding the behavior of one problem solution can simplify the analysis of other problem solutions based on the same pattern.

2 THE BINARY REDUCTION PATTERN

The binary reduction pattern, in which a problem is solved by repeatedly reducing it to subproblems half its size, appears in many different contexts. For example, binary search repeatedly halves the list to be searched until the desired item is found (or else the list is exhausted). Merge sort breaks the list in half, recursively sorts each half, and then merges the sorted halves together again.

Algorithms such as binary search and merge sort are commonly taught in CS1, but not without difficulty on the part of the students. There is a tendency to focus on details such as the syntax of list manipulation or the behavior of recursion, thus failing to see the similarities between algorithms. As such, lessons learned implementing one algorithm are not carried over to another. The complexity of the implementations also tends to make analysis of the algorithms difficult.

By stressing the common binary reduction pattern, algorithms such as binary search and merge sort can be more easily understood and appreciated. The effect of repeatedly doubling or halving a quantity can first be understood by solving and analyzing exponential growth puzzles. Building from this knowledge, a guessing game which incorporates the binary reduction pattern can then be introduced. The implementation of this game and its analysis serve as foundations for implementing and analyzing binary search, since it similarly uses the binary reduction pattern. More complex binary reduction applications such as recursive exponentiation and recursive sorting can then follow, with the lessons learned from

earlier problem solutions making each successive problem easier to tackle.

2.1 Exponential Growth Puzzles

The power of binary reduction as a problem-solving approach can be foreshadowed early in the CS1 course through the use of simple, exponential growth puzzles. For example,

Folding a piece of paper in half results in a folded sheet twice as thick as the original sheet. Similarly, folding this doubled sheet results in a folded sheet four times as thick as the original sheet, and so on. If this process is continued, how many folds would be necessary to obtain a folded sheet whose thickness stretched from the earth to the sun?

I often introduce this puzzle to demonstrate while loops, since a solution can be written in only a few lines of code using a while loop and an assignment. This simple example also demonstrates the explosive effect of doubling, i.e., exponential growth. In solving this puzzle, the students are surprised to discover that only 52 folds are necessary, assuming that a single sheet of paper is .002 inches thick.

The exponential growth obtained through repeated doubling can conversely be described in terms of halving. Repeatedly halving a quantity reduces its size very quickly, reaching unit size in only a logarithmic number of steps. In the folding puzzle, each unfolding of a folded sheet reduces its thickness by one half. Starting with a folded sheet one astronomical unit thick, only 52 unfoldings are necessary to reduce it to a single unfolded sheet.

The following are several variations on this puzzle which similarly demonstrate the power of doubling/halving:

Note that every time a piece of paper is folded, its writing surface is halved. An interesting follow-up question to the paper folding puzzle is to ask how big the sheet would have to be initially in order for 52 folds to leave an 8.5 x 11 inch writing surface.

Consider a chain letter. Suppose the originator of the letter emails a copy to one other person on January 1, and on each subsequent day, everyone who already has the letter emails a copy to someone who doesn't. On what day does everyone in the United States have a copy?

Because floating-point numbers have a limited representation size, there is a point at which floating-point values become so small that they are rounded down to zero. This transition point can be found by repeatedly halving a value. Starting with a value such as 1, repeated halving will reduce its size very quickly until it eventually becomes zero.

Puzzles such as these demonstrate in a concrete way the concept of exponential growth (or, conversely, logarithmic reduction). Students easily learn that repeatedly doubling a value makes it grow very quickly, and, conversely, halving a value reduces it very quickly. This intuitive understanding then sets the stage for a better comprehension of binary reduction in other contexts.

2.2 Guessing Game

Assuming only a familiarity with loops and conditionals, a variation of binary search can be introduced in the context of a guessing game:

Player 1 thinks of a number in the range 1 to 100. Player 2 then attempts to determine that number by making repeated guesses, with Player 1 identifying each guess as being either correct, too high, or too low. The goal is for Player 2 to identify the number in as few guesses as possible.

As Player 2 in this game, students usually start by proposing random numbers in the range, hoping to get lucky. Eventually, some student will propose a systematic approach, picking the middle number in the range, and then adjusting the potential range accordingly.

Essentially, this midpoint strategy implements binary search, where the range can be thought of as an implicit list being searched. Unlike an explicit binary search, however, the game context makes the guessing strategy easier to understand and analyze. For example, the students are able to identify the best-case and worst-case behaviors of this strategy. In the best-case, Player 1 selects 50, the midpoint of the range, and so Player 2 will find that number in only one guess. A worst-case occurs when Player 1 selects 100, requiring seven guesses. The students are even able to sketch out an argument as to why this midpoint strategy is best, i.e., why no other strategy can guarantee finding the number in fewer guesses.

Understanding the efficiency of the guessing strategy is aided by the earlier analysis of the doubling/halving puzzles. One way of looking at the strategy is to recognize that each guess halves the potential range of

numbers. Starting with a range of 100 possible numbers (1 to 100), the first guess halves the range to at most 50 (either 1 to 49 or 51 to 100), the next guess halves that range to 25, and so on. As demonstrated by the earlier puzzles, halving reduces a value very quickly. From this intuition, students understand how this strategy is able to hone in on the correct number quickly, even if the initial range is expanded considerably. (More formally, a range of N numbers requires at most $\lfloor \log_2 N \rfloor + 1$ guesses.)

Conversely, the efficiency of the binary search strategy can be understood in terms of doubling. We have already seen that the maximum number of guesses required to find a number in the range 1 to 100 is seven. Now consider doubling the initial range. The maximum number of guesses required to find a number in the range 1 to 200 is eight, since the first guess reduces the range to at most 100 (and we know that that range can be searched in at most seven guesses). In general, doubling the range only requires one more guess in the worst-case. Since the earlier puzzles demonstrated how doubling a value increases its size very quickly, it follows that large ranges of numbers can be handled in very few guesses. (More formally, N guesses are sufficient for a range of 2^{N-1} numbers.)

Coding this game is a good exercise in combining loops and conditionals. More importantly, however, the guessing game provides a simple and intuitive context for studying the basic binary search algorithm.

2.3 Binary Search

Once lists have been covered, the explicit binary search algorithm can be introduced. After analyzing and implementing binary reduction in the context of the guessing game, binary search is an almost trivial extension. The basic algorithm is the same as in the guessing game, the only difference is that there is now an explicit list to be searched (as opposed to an implicit range of numbers). Discussions about best-case and worst-case behavior of the guessing strategy apply equally well to binary search. In particular, the logarithmic efficiency of binary search is intuitively understood as a function of its halving the list on each inspection.

On occasions where I have introduced binary search without having first implemented the guessing game, I found that students had difficulties coding binary search. For example, confusion would arise between list elements and list indices when searching a list of numbers. By first implementing the guessing game with its implicit range, and then using that code as a framework for

implementing binary search, much of this confusion is avoided.

2.4 Recursive Exponentiation

Recursion is a difficult subject for beginning programmers to comprehend. In many CS1 texts, the subject is omitted completely or only introduced through trivial examples. Unfortunately, introducing recursion through examples such as fibonacci and factorial does recursion a great disservice. Since problems such as these can more naturally (and more efficiently) be encoded without recursion, students fail to see the power behind this technique.

If recursion is to be introduced at the introductory level, it needs to be motivated by problems for which an iterative solution is either less intuitive or less efficient. One such example is exponentiation. A naive approach to computing the value x^y would be to multiply x by itself y times. However, the following recursive relation suggests a method for computing x^y in roughly $(\log_2 y)$ multiplications.

$$x^y = \begin{cases} 1 & \text{if } y = 0 \\ x^{y/2} * x^{y/2} & \text{if } y \text{ is even} \\ x * x^{y/2} * x^{y/2} & \text{if } y \text{ is odd} \end{cases}$$

In order to compute x^y , you recursively compute $x^{y/2}$ and then square that quantity (and multiply again by x if y was odd). The base case for the recursion is when $y = 0$, since x^0 is always 1.

Getting beginning programmers to understand recursion, let alone appreciate its efficiency, can be very difficult. However, I have found that my students immediately recognize the logarithmic nature of this function due to their previous exposure to binary reduction. While the coding of this function appears dissimilar from binary search and the doubling/halving puzzles, the binary reduction pattern is apparent. Each recursive computation involves an exponent which is half as large. From earlier analysis of this pattern, it is known that a quantity can only be halved a logarithmic number of times before reaching 1. And since this recursion terminates when the exponent is 1, only a logarithmic number of recursive computations can be made before reaching the base case. Each recursive computation is involved in at most two multiplications (when the exponent is odd), so the entire computation requires at most $(2 * \log_2 y)$ multiplications.

2.5 Recursive Sorting

Sorting is often the last topic covered in an introductory course. While $O(N^2)$ algorithms such as selection sort and insertion sort are relatively easy to code and analyze, more efficient $O(N \log N)$ algorithms such as merge sort and quick sort are often found to be too challenging. In particular, the recursive nature of these algorithms makes them difficult for first year students to analyze.

Once again, the binary reduction pattern provides a framework for understanding and analyzing these algorithms. For example, the merge sort algorithm breaks a list into halves, recursively sorts each half, and then merges the sorted halves together. Since this is yet another instance of the binary reduction pattern, the students already have a context for understanding the algorithm. Each recursive call operates on a sublist that is half as long, so the recursion must terminate (i.e., reach a list of size one or zero) after only a logarithmic number of calls. Intuitively, this logarithmic limit on the length of the recursive sequence can be seen as the source of the logarithmic term in the complexity of merge sort.

A more careful analysis can be made based on this intuition. Consider the number of calls at each level of the recursive calling sequence. At the topmost level, the initial call to merge sort operates on the original list of size N . This routine makes two recursive calls, each with half of the original list (size $N/2$). Each of these makes two recursive calls, for a total of four calls, each with list size $N/4$. The next level of recursion results in eight calls, each with list size $N/8$, and so on down to the final ($\log_2 N$) level of the recursion which has N calls, each with list size 1. (For simplicity, we will assume that N is a power of two.) In general, at level i of the recursion there are i calls, each operating on a list of size N/i . Since each recursive call does an amount of work proportional to the length of its list, the overall cost is:

$$\begin{aligned} & \sum_{i=1}^{\log_2 N} \text{cost of calls at level } i \text{ of the recursion} \\ &= \sum_{i=1}^{\log_2 N} i \times (\text{cost of call with list size } N/i) \\ &= \sum_{i=1}^{\log_2 N} i \times (c \times N/i) \quad \text{for some constant factor } c \\ &= \sum_{i=1}^{\log_2 N} c \times N \\ &= c \times N \times \log_2 N \quad \Rightarrow O(N \log N) \end{aligned}$$

These same insights can be used to study quick sort. Quick sort partitions a list into all items less than and

greater than a chosen pivot element, and then recursively sorts each of those partitions. In the ideal case, the pivot chosen at each step is in the median of the list, and so the partitions are the same length. If this is the case, quick sort also follows the binary reduction pattern, since each recursive call operates on a partition that is half as long. If, however, the pivot is repeatedly chosen from one extreme or the other, then the partitions will not be similar in length and the halving effect will be lost. In this case, quick sort can degrade to $O(N^2)$.

3 CONCLUSION

As my experiences demonstrate, the use of problem-solving patterns as a recurring theme has numerous advantages. Pattern recognition allows students to adapt simpler strategies to new and more complex problems. Similarly, understanding the solution to a problem can greatly simplify the analysis of more complex problems which follow the same pattern. Building from the analysis of simple examples, complex problems such as recursive exponentiation and merge sort, which might otherwise have been beyond the grasp of the students, are easily understood. In fact, one might hypothesize that the time spent understanding successive problem solutions would decrease logarithmically.

4 REFERENCES

- [Astr94] Astrachan, Owen (1994). "Self-Reference is an Illustrative Essential." *SIGCSE Bulletin* 26(1): 238-242.
- [AR95] Astrachan, Owen and David Reed (1995). "AAA and CS 1: The Applied Apprenticeship Approach to CS 1." *SIGCSE Bulletin* 27(1): 1-5.
- [Coad92] Coad, Peter (1992). "Object-Oriented Patterns." *Communications of the ACM* 35(9): 152-159.
- [GHJV95] Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides (1995). *Design Patterns*, Addison-Wesley.
- [MH96] McLoughlin, Henry and Kevin Hely (1996). "Teaching Formal Programming to First Year Computer Students." *SIGCSE Bulletin* 28(1): 155-159.
- [Rist89] Rist, Robert S. (1989). "Schema Creation in Programming." *Cognitive Science* 13: 389-414.
- [Wall96] Wallingford, Eugene (1996). "Toward a First Course Based on Object-Oriented Patterns." *SIGCSE Bulletin* 28(1): 27-31.