Using Puzzles in Teaching Algorithms

Anany Levitin Mary-Angela Papalaskari
Department of Computing Sciences
Villanova University
Villanova, PA 19085
anany.levitin@villanova.edu

Abstract

This paper advocates a wider use of puzzles and puzzle-like problems in teaching design and analysis of algorithms. It discusses a variety of puzzles and classifies them according to the general algorithm design techniques. Pedagogic issues are explored.

1 Introduction

Traditionally, most important concepts related to design and analysis of algorithms have been illustrated by problems of mathematical nature. This has been true even for textbooks organized around algorithm design techniques [3, 7, 12]. We believe that it would be very desirable to illustrate the principal ideas of algorithmics on other material as well. In particular, we believe that puzzles and puzzle-like problems remain an untapped source of such enrichment.

The idea of using puzzles to illustrate or reinforce key concepts is not new for computer science education. There has been no attempt, however, to collect and systematically catalogue puzzles illustrating each algorithm design technique.

What are advantages of using puzzles and puzzle-like problems for teaching design and analysis of algorithms? In our view, there are several:

- Puzzles force students to think about algorithms on a more abstract level, divorced from programming and computer language minutiae.
- Puzzles show that algorithm design strategies can be looked upon as general problem-solving tools that might be useful in areas far removed from computer science.
- Solving puzzles helps in developing creativity and problem solving skills — the qualities any future CS professional should strive to acquire.
- Puzzles usually attract more interest on the part of students, making them work harder on the problems assigned to them.
- Puzzles can be used in a variety of ways in an algorithms course:
  a) to introduce a new approach to algorithm design;
  b) as exercises to hone students' skills in a particular approach to algorithmic thinking;
  c) as a theme for a programming project.

As mentioned earlier, there are a few puzzles that are a standard feature of algorithm textbooks but the number of them is very small. Of these, the most widely used puzzle is arguably the Tower-of-Hanoi problem. It provides a natural and convenient vehicle for illustrating the idea of a recursive algorithm, for showing how a recursive algorithm can be analyzed by setting up and solving a recurrence relation, and even for proving an algorithm's optimality. Other standard examples include the Königsberg bridge puzzle (to introduce Euler circuits), mazes (in conjunction with depth-first search), and the n-queens problem (for illustration of backtracking).

Why is this list so short? In addition to the unavoidable inertia of textbook writing, there is, of course, a deeper reason for the paucity of puzzles in algorithm textbooks. Many puzzles are based on exploiting idiosyncrasies of problems, whereas algorithms typically seek to solve problems with a very large if not infinite variety of inputs.

Despite this principal obstacle, we believe that more puzzles useful for teaching algorithms can be found. In support of this view, we quote below several puzzles that illustrate general algorithm design techniques. Many of these puzzles are included in the forthcoming textbook by one of this paper's authors [11].

SIGCSE'02, February 27- March 3, 2002, Covington, Kentucky, USA.

2 Puzzles for illustrating design techniques

We will use the hierarchical taxonomy of algorithm design techniques proposed by Levitin [10]. It divides the strategies into two groups. The group of most general paradigms includes four strategies: brute force, divide-and-conquer, decrease-and-conquer, and transform-and-conquer. The group of less general ones consists of the greedy approach, dynamic programming, backtracking, and branch-and-bound. We will start the discussion with the most general strategies. For each of these four strategies and their varieties, we will provide examples of puzzles whose algorithmic solutions illustrate the strategy in question.

Brute Force This strategy is defined as a straightforward approach to solving a problem, usually directly based on the problem's statement and definitions of the concepts involved. Its important special case is *exhaustive search*: a brute-force approach to solving problems that involve — explicitly or implicitly — combinatorial objects such as permutations, combinations, and subsets of a given set. It suggests generating all the elements of the problem's domain and then finding a desired element (e.g., the one that optimizes a given objective function).

For obvious reasons, interesting puzzles don't usually lend themselves to solutions by brute force in general and by exhaustive search in particular. In fact, many puzzles can provide good examples of problems that either cannot be solved by brute force at all, or for which this strategy yields a very clumsy and unsatisfactory solution. To make this point, one can assign, for example, the famous puzzle of crossing nine dots in the plane by four straight lines without lifting a pen off the paper (e.g., [8]) – the quintessential "thinking outside the box" puzzle.

As for puzzles that can be solved by brute force, one can suggest, for example, getting the 3-by-3 magic square by exhaustive search. It provides a good illustration of the limitations of exhaustive search and the usefulness of knowing an algorithm's efficiency class. Thus, students can be asked to run an experiment to determine the largest magic square that the exhaustive search algorithm can find in one minute of computer time and then estimate the order of the largest magic square that can be found in one year or in one hundred years.

Divide-and-conquer Divide-and-conquer is based on partitioning a problem into several smaller subproblems (usually of the same kind and ideally of about the same size), solving each of them recursively, and then combining their solutions to get a solution to the original problem.

Surprisingly, there seem to be few puzzles solvable by the divide-and-conquer approach. Here are two examples that are rather well known. The first one is the triomino puzzle (e.g., [13, #49]). The object is to cover any 2^n-by-2^n chessboard with one missing square with triominos, which are "L"-shaped tiles formed by three adjacent squares of a chessboard. (A missing square can be any of the board squares. Triominos should cover all the squares except the missing one, with no overlaps.)

The other problem is the *nuts-and-bolts problem* [16]: Given a collection of n bolts of different widths and n corresponding nuts, match each bolt to its nut. One is allowed to try a nut and bolt together, from which one can determine whether the nut is larger than the bolt, smaller than the bolt, or matches the bolt exactly. There is no way, however, to compare two nuts together or two bolts together.

Another divide-and-conquer algorithm for solving a puzzle is an algorithm for the knight's tour problem suggested by Parberry [15].

Decrease-and-conquer Solving a problem by reducing a given instance to a smaller one, solving the latter recursively, and then extending the obtained solution to get a solution to the original instance is, of course, a well-known algorithm design approach. Though this approach is considered by some to be a special case of divide-and-conquer, it is better to consider them as distinct design strategies. The crucial difference between the two lies in the number of smaller subproblems that need to be solved: several (usually, two) in divide-and-conquer algorithms and just one in decrease-and-conquer algorithms. It is further useful, both from the design and the analysis perspectives, to distinguish three varieties of this strategy.

The *decrease-by-a-constant* variety suggests decreasing a problem's size by a constant, typically one. Here is an old and very simple puzzle illustrating this strategy. A detachment of n soldiers must cross a wide and deep river with no bridge in sight. They notice two boys playing in a rowboat by the shore. The boat is so tiny, however, that it can only hold two boys or one soldier. How can the soldiers get across the river and leave the boys in joint possession of the boat? How many times does the boat need to pass from shore to shore? This puzzle is very good for our purposes. First, it is at the right level of difficulty because the answer is not immediately obvious, but, after some thought, students can fairly reliably arrive at the answer. Second, it is not a very widely known puzzle, giving students the opportunity to actually be engaged by it and to successfully solve it (the "aha" experience). Third, it can also be used as springboard for a discussion of the relationship between the recursive nature of some strategies and their ultimate implementations in specific situations.

Another puzzle illustrating the decrease-by-a-constant strategy is a version of the nuts-and-bolts problem in which the goal is just to find the smallest bolt and the matching nut [13, #612].

Decrease-by-a-constant-factor, the second variety of the decrease-and-conquer strategy, suggests decreasing a problem's size by a constant factor, most often two. Since the principal example of the decrease-by-half technique is binary search, any puzzle or game (like 20 questions) based on this algorithm's idea can be used to illustrate the strategy.

A more interesting and useful puzzle is determining a fake coin with a balance scale. An easier version of the problem assumes that it is known that the fake coin is, say, lighter than the genuine one. The attractiveness of this puzzle lies in the fact that it can be solved either by dividing coins into two groups of coins or into three groups of coins. The first approach is most obvious, and its analysis is almost identical to that of binary search. The division-into-three is less obvious, provides a rather rare example of dividing a problem into three subproblems, and is, in fact, more efficient (asymptotically, by a constant factor) than the divide-into-two algorithm. A comparative study of these two algorithms can be particularly instructive for students.

A much more challenging version of this problem does not assume any additional information about relative weights of the fake and genuine coins. It is famous among puzzle lovers but would be difficult for most students. Its solution can be found, for example, in [17]. Brassard and Bratley [3] use this puzzle to demonstrate the interplay between algorithm design and decision trees.

As one more nontraditional decrease-by-half example — though it can hardly be called a puzzle — we can mention the Josephus Problem. A good analysis of it can be found in [6].

Variable-size decrease algorithms is the last variety of the decrease-and-conquer strategy. In such algorithms, the size reduction varies irregularly from iteration to iteration. Euclid's algorithm, the partition-based algorithm for the selection problem, and binary tree search are mathematical examples of *variable-size decrease* algorithms. Some versions of the game of Nim can be used to illustrate this strategy. Here is a typical example. There is a pile of n matches on a table. Two players take turns removing 1,2, 3, or 4 matches. The winner is the player who removes the last match. Design a winning strategy, if it exists, for the player making the first move.

Transform-and-conquer The last most general technique is based on the idea of transformation. Its first variety — called *instance simplification* — solves a problem by first transforming an instance given to another instance of the same problem (and of the same size) with some special property which makes the problem easier to solve. A good example of this technique is presorting (e.g., for finding equal elements in a list).

The same idea can also be illustrated by an easier version of the nuts-and-bolts problem in which one can not only compare any bolt with any nut but also can also compare nut sizes with each other, and bolt sizes with each other. The goal is to match each bolt to its nut.

The second variety of the transform-and-conquer strategy — called *representation change* — is based on a transformation of a problem's input to a different representation, which is more conducive to an efficient algorithmic solution. Examples from traditional computer science include search trees, hashing, Horner's rule, FFT, and heaps. As to unorthodox problems, one can point out an impressive algorithm for finding all sets of anagrams (e.g., ``eat,'' ``ate,'' and ``tea'' belong to the same set) in a dictionary of English words [2]. The algorithm first assigns each word a signature obtained by sorting its letters (it is a representation change) and then sorts the dictionary in alphabetical order of the signatures (it is a presorting idea) to put anagrams next to each other.

As very exotic (but still making the point) examples of representation change, we can mention two items from computer science folklore (see, e.g., [4]). The first one is the so-called "spaghetti sort." Numbers to be sorted are represented by lengths of uncooked spaghetti rods that are placed vertically on the table to immediately identify the longest one in the bunch. Note that this provides even a better illustration of an efficient implementation of the priority queue. The second item is solving the shortest path problem with a string model of the graph in question.

The third variety of the transformation strategy is *problem reduction*, in which an instance of a given problem is transformed into an instance of a different problem altogether. To illustrate, several puzzles can be reduced to questions about graphs. The Königsberg bridge puzzle is one such example. There are also several well-known ferrying puzzles such as the problem of how a peasant can ferry himself, a wolf, a goat, and a cabbage head over a river (e.g., [8]). The problem can be represented by a graph whose vertices specify possible states of the problem and edges correspond to legitimate river crossings. This reduces the problem to the question about a path (say, with a minimum number of edges) from the vertex representing the initial situation to the vertex representing the final one.

Less general strategies One should not expect a rich bounty from a hunt for puzzles solvable by the *greedy approach*. The reason is the same as for the brute-force strategy: good puzzles are usually too "tricky" to be solvable in a straightforward fashion. We found one notable exception in Huffman trees, which are usually introduced for constructing optimal prefix-free codes and whose construction is universally considered to be a greedy algorithm. Less known are their applications to decision trees mentioned by Knuth [9, p. 402]. Gardner [5, p. 30] provides a useful puzzle — a version of the twenty

questions with unequal probabilities of item selection — that takes advantage of this idea. Also, Parberry [14] has suggested an algorithm for the sliding tiles puzzle, which is based on a combination of the greedy and divide-and-conquer strategies.

Algorithm textbooks contain a rather standard set of examples illustrating *dynamic programming*, which do not include puzzles. The book by Richard Bellman, the method's inventor, and Stuart Dreyfus [1], does contain two such applications. The first one deals with the so-called "missionaries and cannibals" puzzle (pp. 97-99); the second one deals with a very difficult problem of detecting two fake coins with a balance scale (pp. 168-178). These examples can be useful for discussing limitations of dynamic programming.

We have already mentioned above the n-queens problem algorithm based on the *backtracking* technique. This and several other similar puzzles are standard topics in Artificial Intelligence (AI). From the AI perspective, backtracking is just one of several approaches to state-space search which can also be adapted to solve some optimization problems. The "missionaries and cannibals" problem and the sliding tiles puzzle are in fact very commonly used as illustrations in this connection (see, for example, [18]). State-space search can also be adapted to the analysis of two-person games; the minimax algorithm with α-β pruning is the premier approach to adversary search. This, in turn, can be used as an illustration of the *branch-and-bound* strategy.

In this discussion we have intentionally expanded our puzzle domain to include some games. From our perspective, the use of games offers all the pedagogical advantages mentioned in the introduction.

3 Pedagogical Comments

What courses would benefit from a more extensive usage of puzzles? We believe that any course dealing with algorithms would. This includes introductory programming courses and courses on computer problem solving for non-majors. But it is the course on design and analysis of algorithms that, in our view, should benefit from such an introduction most. We are in agreement with a widely shared opinion that such a course should be organized around algorithm design techniques rather than specific problem types. Such organization of the course makes puzzles indispensable for showcasing fundamental algorithm design strategies as general problem solving paradigms. In our experience, many students come to a course on design and analysis of algorithms prone to thinking in terms of a specific programming language taught in their introductory programming sequence. This makes it harder for them to think on the more abstract level necessary for a successful mastery of algorithm design and analysis techniques. Puzzles provide an indispensable tool for loosing this unfortunate grip of computing minutiae.

As to specific ways to introduce puzzles, we have used most of them through a systematic inclusion in homeworks, as examples of applying specific algorithm design strategies. A few of them can be used in a lecture to introduce a particular design technique. In addition to the canonical introduction of backtracking via the n-queens problem, we can mention again the fake-coin identification problem. One may argue that this puzzle provides the best way to introduce the decrease-by-a constant factor strategy because of the natural opportunity to discuss the size reduction by both a factor of two and a factor of three.

Some puzzles — such as the magic square construction already mentioned above — can be used as projects involving an algorithm timing and empirical investigation of relative efficiency of several algorithms for the same problem. Assigning puzzles as projects has an additional advantage of leading often to less straightforward data structures and objects. Games, of course, can provide even better vehicles for the latter purpose.

Though our principal goal was to find puzzles for illustrating algorithm design strategies, they can also be used for teaching algorithm analysis techniques. For example, the same fake-coin identification problem can serve this purpose as well as the often-used Tower-of-Hanoi puzzle.

4 Conclusion

Puzzles can be very helpful for teaching different aspects of algorithmics. In this paper, we quoted several puzzles and puzzle-like problems that can be used for illustrating most general algorithm design techniques: brute force, divide-and-conquer, decrease-and-conquer, and transform-and-conquer and less general techniques: greedy, dynamic programming, backtracking, and branch-and-bound. We hope that more puzzles suitable for teaching design and analysis of algorithms will be found in existing collections or specifically designed for this worthy purpose. Some promising sources of such puzzles are references [19-22], which primarily deal with problems related to computer science. There are also a few Web sites that provide links to puzzle-related repositories (see, for example, links at http://home.sunrise.ch/pglaus/puzzlinks.htm).

References

[1] Bellman, R.E. and Dreyfus, S.E. *Applied Dynamic Programming*. Princeton University Press, 1962.

[2] Bentley, J. *Programming Pearls*. 2nd ed., Addison-Wesley, 2000.

[3] Brassard, G. and Bratley, P. *Fundamentals of Algorithmics*. Prentice-Hall, 1996.

[4] Dewdney, A.K. *The (New) Turing Omnibus: 66 Excursions in Computer Science*. Computer Science Press, 1992.

[5] Gardner, M. *My Best Mathematical and Logical Puzzles*. Dover, 1994.

[6] Graham, R.L., Knuth, D.E., and Potashnik, O. *Concrete Mathematics*. Addison-Wesley, 1988.

[7] Horowitz, E., Sahni, S., and Rajasekaran, S. *Computer Algorithms*. Computer Science Press, New York, 1996.

[8] Kordemsky, B. *The Moscow Puzzles: 359 Mathematical Recreations*. Dover, 1992.

[9] Knuth, D.E. *The Art of Computer Programming, Volume I: Fundamental Algorithms,* 3rd ed., Addison-Wesley, 1997.

[10] Levitin, A. Do we teach the right algorithm design techniques? in *Proceedings of SIGCSE '99* (March 1999), 179-183.

[11] Levitin, A. *Introduction to Design and Analysis of Algorithms*. Addison-Wesley, 2002 (to appear).

[12] Neapolitan, R.E. and Naimipour, K. *Foundations of Algorithms*. Jones and Bartlett Publishers, 1996.

[13] Parberry, I. *Problems on Algorithms*. Prentice-Hall, 1995.

[14] Parberry, I. A real-time algorithm for the $(n^2 - 1)$-puzzle. *Information Processing Letters 56* (1995), 23-28.

[15] Parberry, I. An efficient algorithm for the knight's tour problem. *Discrete Applied Mathematics 73* (1997), 251-260.

[16] Rawlins, G.J.E. *Compared to What? An Introduction to the Analysis of Algorithms*. Computer Science Press, 1991.

[17] Reingold, E.M., Nievergelt, J., and Deo, N. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.

[18] Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach*. 2nd ed., Prentice-Hall, 2001.

[19] Shasha, D.E. *Codes, Puzzles, and Conspiracy*. W.H. Freeman and Co., 1992.

[20] Shasha, D.E. *The Puzzling Adventures of Doctor Ecco*. Dover, 1998.

[21] Shasha, D.E. "Dr. Ecco's Omniheurist Corner," the column in the *Dr. Dobb's Journal*.

[22] Shasha, D.E. "Puzzling Adventures," the column in the *Scientific American*.