Puzzles as Algorithmic Problems

This article is a brief advocacy for the use of puzzles as algorithmic problems for learning purposes as an alternative to the current style of Competitive Programming(CP) or Mathematics problems.

The current trend in "how to learn algorithms" is mostly based on big tech companies and their interview process. This process is interleaved with the ICPC(International Collegiate Programming Contest) style of competitive programming, where a problem is usually a combination or adaptation of several known algorithms or mathematical concepts. I see four problems with this approach:

1. **The problems are not real-world problems**
2. **The problems have pre-defined and determined solutions**
3. **The problems are very hard to solve if you don't already know the solution**
4. **The problems are not fun**

Let me elaborate on these points by walking through the process of solving a LeetCode problem. I picked a medium problem with 34.5% solve rate, 3Sum. The problem statement is as follows:

> Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0. Notice that the solution set must not contain duplicate triplets.

The naive solution would be to simply brute force the entire array, which would be O(n^3). This is obviously not the intended solution, so we are led to think where we can optimize. A classic approach for array questions is to first sort the array in the hope that it will allow for some optimization in the algorithm. After sorting the array, we can use the two-pointer technique to find the triplets.

As you might've realized when reading the above paragraph, I didn't invent anything when solving the question. In fact, the author already had an optimal solution, as well as a naive one, when they were writing the question. This is a common theme in competitive programming problems. The author has some solution in mind, usually by combining a few techniques or algorithms in a clever way, and the problem solver has to figure out that specific solution.

This is in stark contrast to real-world problems. In a real-world problem, there is no "existing" solution that you have to discover. Of course the tricks and techniques you learn from competitive programming can be applied to real-world problems, but the process of solving a real-world problem is much more open-ended. You also need to decide on your constraints yourself, unlike the constraints given in competitive programming problems. **What does it mean for your algorithm to be "fast enough"?** Even further, **what does it mean for your algorithm to be "correct"?** In CP, the answer is usually "passes all test cases", but in the real world, the answer is not so clear-cut.

Given such discrepancies, I am proposing solving puzzles by devising algorithms for them as an alternative to CP problems. The puzzles are actually fun to work with, they aren't designed to be solved algorithmically, so there is no solution you have to discover, you actually have to invent a solution. These puzzles can be anything from variants of Sudoku to small Chess problems, any puzzle that doesn't require any special knowledge to solve. The fact that you have to define your own constraints makes solving puzzles a much better approximation of real-world problem solving than CP problems. Solving a puzzle algorithmically means that **(1)** you must define your constraint for correctness and performance, as well as the space of inputs you are interested in, **(2)** you must devise a robust testing strategy, **(3)** you must model the problem space as a data structure in the programming language, perhaps think about the trade-offs between different representations of the problem space, **(4)** you must devise an algorithm that solves the problem, and **(5)** you must implement the algorithm and test it. **Together, I believe these steps are a much better approximation of real-world problem solving than CP problems.**

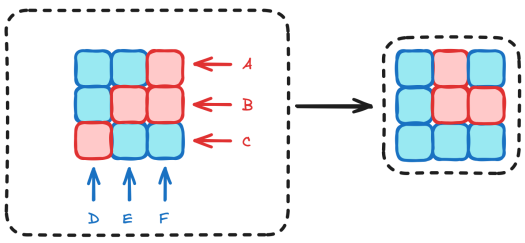

Figure 1: Paintbrush Puzzle

Figure 1 an example of a puzzle that I solved recently. Your task is to find a set of *brushes* that paint the source into the target canvas. For this particular example, brushing order A-F-B-D is the solution. **Can you devise a general algorithm for solving this puzzle for any source and target canvas in any size?** If you're interested, you can also read my solution at https://www.alperenkeles.com/blog/paintbrush.