How to Solve a 320 Problem

January 2, 2018

1. **Trivial and small instances:**
   (a) Write down at least one trivial problem instance (and enough to cover all the problem's trivial cases).
   (b) Write down at least one small but non-trivial problem instance (and enough instances to understand the simplest points where the problem starts requiring decisions).
   (c) Find elements missing from your instances. Have you explored all aspects of the problem? Everything in the structure of an instance? Generate the smallest instances you can to "push on" these questions.
   (d) Write down the largest realistic-looking example you can handle fairly easily by hand.
2. **Represent the problem:**
   (a) Give parameterized names to quantities that matter in the problem.
   (b) Express your small (and, if reasonable, trivial) instances using these names.
   (c) Design at least one graphical (i.e., sketched) representation of the problem.
   (d) Sketch your small (and, if reasonable, trivial) instances using this representation.
   (e) Characterize a valid problem instance in terms of these named quantities and (potentially) the graphical representation.
3. **Represent the solution:**
   (a) Give parameterized names to quantities that matter in the solution.
   (b) Design at least one graphical (i.e., sketched) representation of the solution.
   (c) Characterize a solution in terms of your problem and solution represesenations. That is, give an algorithm or metric to determine whether a potential solution is actually a solution and, if relevant, to determine how good a solution it is.
   (d) Give an example solution and—if relevant—a **best** solution for each of your trivial instances and at least one of your small instances.
4. **List similar problems and classes of problems:**

   Throughout this part, keep track of not only what's similar, but what's **different**.
   (a) Give at least one example of a problem you've seen that has similar **surface** features: "story", inputs, outputs, constraints on inputs and outputs, etc.
   (b) Give at least one example of a problem you've seen that has similar structures to its trivial instances, small instances, quantity names, or graphical representation. (I.e., an example problem that has similar **structural** features.)

(c) Identify at least one search term based on some feature of the problem you could look for (in a textbook index, a web search, or a digital library search) to find other similar problems.

(d) List promising classes of solution approaches, based on the problems you noted above.

(e) If you haven't already, find a way to describe the input, output, or other aspects of the problem as a graph. (Graphs are **so** useful that this is worth asking for specially!)

5. **Give the brute force algorithm:**

   A brute force algorithm lists and test all possible solutions, with no effort to be clever. More generally, give the most straightforward algorithm you think of, not optimizing for anything.

   (a) Find a way to enumerate all potential solutions. (You already know how to measure the correctness/quality of a solution from your work representing solutions above.)

   (b) Choose an appropriate variable (or variables!) to represent the "size" of an instance.

   (c) Analyse the performance of this algorithm.

6. **Lower-bound all solutions:**

   (a) Determine the minimum amount of data necessary to review in order to solve the problem and use this to establish a lower bound.

   (b) Count the number of distinct solutions for a particular problem size and pair this with a decision tree argument to establish a lower-bound.

7. **Develop a promising approach:**

   (a) Using your work above, select an approach that looks promising in comparison to brute force. **NOTE**: You might instead establish (1) that brute force is good enough in your context or (2) that the problem is hard (e.g., NP-complete).

   (b) Characterize what you need to develop your approach (e.g., a recurrence relation for a dynamic programming approach or an ordering of choices and quality metric for a greedy approach).

   (c) **Give an algorithm to solve the problem!**
   Take the steps specific to the promising approach you chose (laid out in the previous step).

8. **Challenge your approach:**

   Throughout this step, you are debugging your algorithm (or proof or other approach) and looking for features of the algorithm, instances, or analysis that might inspire a better approach.

   (a) **Carefully** walk through your algorithm on the small (and possibly the trivial) instances above. Analyse its correctness and performance on these.

   (b) Try to design problem instances that challenge design features of your algorithm. (E.g., for a greedy algorithm, you might observe how it solves a small instance correctly and try to inject information later in the instance that would make its early choices incorrect.)

   (c) Try to prove your algorithm correct. Then, go back and use features of your proof to design more challenging problem instances (to debug your proof).

   (d) Analyse your algorithm's performance.

9. **Repeat!** Especially repeat the steps starting at "choose a promising approach", but revisiting **all** of these steps can be important!

   Many steps will inform others. For example, as you make your modest-sized instance, you may find interesting properties of that instance that aren't demonstrated by any of your small instances. OK; try to build the smallest instance you can that demonstrates this new property you noticed!