10

How to Design Algorithms

Designing the right algorithm for a given application is a major creative act—that of taking a problem and pulling a solution out of the ether. The space of choices you can make in algorithm design is enormous, leaving you plenty of freedom to hang yourself.

This book is designed to make you a better algorithm designer. The techniques presented in Part I of this book provide the basic ideas underlying all combinatorial algorithms. The problem catalog of Part II will help you with modeling your application, and tell you what is known about the relevant problems. However, being a successful algorithm designer requires more than book knowledge. It requires a certain attitude—the right problem-solving approach. It is difficult to teach this mindset in a book, yet getting it is essential to becoming a successful designer.

The key to algorithm design (or any other problem-solving task) is to proceed by asking yourself questions to guide your thought process. *What if we do this? What if we do that?* Should you get stuck on the problem, the best thing to do is move onto the next question. In any group brainstorming session, the most useful person in the room is the one who keeps asking, *"Why can't we do it this way?"* not the person who later tells them why, because she will eventually stumble on an approach that can't be shot down.

Towards this end, we provide a sequence of questions to guide your search for the right algorithm for your problem. To use it effectively, you must not only ask the questions, but answer them. The key is working through the answers carefully by writing them down in a log. The correct answer to *"Can I do it this way?"* is never "no," but "no, because. . . ." By clearly articulating your reasoning as to why something doesn't work, you can check whether you have glossed over a possibility that you didn't want to think hard enough about. It is amazing how often the reason

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_10,

you can't find a convincing explanation for something is because your conclusion is wrong.

The distinction between *strategy* and *tactics* is important to keep aware of during any design process. Strategy represents the quest for the big picture—the framework around which we construct our path to the goal. Tactics are used to win the minor battles we must fight along the way. In problem-solving, it is important to check repeatedly whether you are thinking on the right level. If you do not have a global strategy of how you are going to attack your problem, it is pointless to worry about the tactics. An example of a strategic question is "Can I model my application as a graph algorithm problem?" A tactical question might be, "Should I use an adjacency list or adjacency matrix data structure to represent my graph?" Of course, such tactical decisions are critical to the ultimate quality of the solution, but they can be properly evaluated only in light of a successful strategy.

Too many people freeze up in their thinking when faced with a design problem. After reading or hearing the problem, they sit down and realize that they *don't know what to do next*. Avoid this fate. Follow the sequence of questions provided below and in most of the catalog problem sections. We'll *tell* you what to do next!

Obviously, the more experience you have with algorithm design techniques such as dynamic programming, graph algorithms, intractability, and data structures, the more successful you will be at working through the list of questions. Part I of this book has been designed to strengthen this technical background. However, it pays to work through these questions regardless of how strong your technical skills are. The earliest and most important questions on the list focus on obtaining a detailed understanding of your problem and do not require specific expertise.

This list of questions was inspired by a passage in [Wol79]—a wonderful book about the space program entitled *The Right Stuff*. It concerned the radio transmissions from test pilots just before their planes crashed. One might have expected that they would panic, so ground control would hear the pilot yelling *Ahhhhhhhhhhhh* —, terminated only by the sound of smacking into a mountain. Instead, the pilots ran through a list of what their possible actions could be. *I've tried the flaps. I've checked the engine. Still got two wings. I've reset the* —. They had "the Right Stuff." Because of this, they sometimes managed to miss the mountain.

I hope this book and list will provide you with "the Right Stuff" to be an algorithm designer. And I hope it prevents you from smacking into any mountains along the way.

1. Do I really understand the problem?
   (a) What exactly does the input consist of?
   (b) What exactly are the desired results or output?
   (c) Can I construct an input example small enough to solve by hand? What happens when I try to solve it?
   (d) How important is it to my application that I always find the optimal answer? Can I settle for something close to the optimal answer?

(e) How large is a typical instance of my problem? Will I be working on 10 items? 1,000 items? 1,000,000 items?

(f) How important is speed in my application? Must the problem be solved within one second? One minute? One hour? One day?

(g) How much time and effort can I invest in implementation? Will I be limited to simple algorithms that can be coded up in a day, or do I have the freedom to experiment with a couple of approaches and see which is best?

(h) Am I trying to solve a numerical problem? A graph algorithm problem? A geometric problem? A string problem? A set problem? Which formulation seems easiest?

2. Can I find a simple algorithm or heuristic for my problem?

   (a) Will brute force solve my problem *correctly* by searching through all subsets or arrangements and picking the best one?

      i. If so, why am I sure that this algorithm always gives the correct answer?

      ii. How do I measure the quality of a solution once I construct it?

      iii. Does this simple, slow solution run in polynomial or exponential time? Is my problem small enough that this brute-force solution will suffice?

      iv. Am I certain that my problem is sufficiently well defined to actually *have* a correct solution?

   (b) Can I solve my problem by repeatedly trying some simple rule, like picking the biggest item first? The smallest item first? A random item first?

      i. If so, on what types of inputs does this heuristic work well? Do these correspond to the data that might arise in my application?

      ii. On what types of inputs does this heuristic work badly? If no such examples can be found, can I show that it always works well?

      iii. How fast does my heuristic come up with an answer? Does it have a simple implementation?

3. Is my problem in the catalog of algorithmic problems in the back of this book?

   (a) What is known about the problem? Is there an implementation available that I can use?

   (b) Did I look in the right place for my problem? Did I browse through all pictures? Did I look in the index under all possible keywords?

(c) Are there relevant resources available on the World Wide Web? Did I do a Google web and Google Scholar search? Did I go to the page associated with this book, *http://www.cs.sunysb.edu/∼algorith*?

4. Are there special cases of the problem that I know how to solve?

   (a) Can I solve the problem efficiently when I ignore some of the input parameters?

   (b) Does the problem become easier to solve when I set some of the input parameters to trivial values, such as 0 or 1?

   (c) Can I simplify the problem to the point where I *can* solve it efficiently?

   (d) Why can't this special-case algorithm be generalized to a wider class of inputs?

   (e) Is my problem a special case of a more general problem in the catalog?

5. Which of the standard algorithm design paradigms are most relevant to my problem?

   (a) Is there a set of items that can be sorted by size or some key? Does this sorted order make it easier to find the answer?

   (b) Is there a way to split the problem in two smaller problems, perhaps by doing a binary search? How about partitioning the elements into big and small, or left and right? Does this suggest a divide-and-conquer algorithm?

   (c) Do the input objects or desired solution have a natural left-to-right order, such as characters in a string, elements of a permutation, or leaves of a tree? Can I use dynamic programming to exploit this order?

   (d) Are there certain operations being done repeatedly, such as searching, or finding the largest/smallest element? Can I use a data structure to speed up these queries? What about a dictionary/hash table or a heap/priority queue?

   (e) Can I use random sampling to select which object to pick next? What about constructing many random configurations and picking the best one? Can I use some kind of directed randomness like simulated annealing to zoom in on the best solution?

   (f) Can I formulate my problem as a linear program? How about an integer program?

   (g) Does my problem seem something like satisfiability, the traveling salesman problem, or some other NP-complete problem? Might the problem be NP-complete and thus not have an efficient algorithm? Is it in the problem list in the back of Garey and Johnson [GJ79]?

6. Am I still stumped?

   (a) Am I willing to spend money to hire an expert to tell me what to do? If so, check out the professional consulting services mentioned in Section 19.4 (page 664).

   (b) Why don't I go back to the beginning and work through these questions again? Did any of my answers change during my latest trip through the list?

Problem-solving is not a science, but part art and part skill. It is one of the skills most worth developing. My favorite book on problem-solving remains Pólya's *How to Solve It* [P57], which features a catalog of problem-solving techniques that are fascinating to browse through, both before and after you have a problem.