# Rules for Algorithm Design

The secret to successful algorithm design, and problem solving in general, is to make sure you ask the right questions. Below, I give a possible series of questions for you to ask yourself as you try to solve difficult algorithm design problems:

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 some examples small enough to solve by hand? What happens when I solve them?
   (d) Are you trying to solve a numerical problem? A graph algorithm problem? A geometric problem? A string problem? A set problem? Might your problem be formulated in more than one way? Which formulation seems easiest?
2. Can I find a simple algorithm for the problem?
   (a) Can I find the solve my problem exactly by searching 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?

iv. If I can't find a slow, *guaranteed* correct algorithm, am I sure that my problem is well defined enough to permit a solution?

(b) Can I solve my problem by repeatedly trying some heuristic 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 rule work well? Do these correspond to the types of inputs that might arise in the application?

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

iii. How fast does my heuristic rule come up with an answer?

3. Are there special cases of this problem I know how to solve exactly?

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

(b) What happens when I set some of the input parameters to trivial values, such as 0 or 1?

(c) Can I simplify the problem to create a problem

I can solve efficiently? How simple do I have to make it?

(d) If I can solve a certain special case, why can't this be generalized to a wider class of inputs?

4. Which of the standard algorithm design paradigms seem most relevant to the problem?

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

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

   (c) Are there certain operations being repeatedly done on the same data, such as searching it for some element, or finding the largest/smallest remaining element? If so, can I use a data structure of speed up these queries, like hash tables or a heap/priority queue?

5. Am I still stumped?

   (a) Why don't I go back to the beginning of the list and work through the questions again? Do any of my answers from the first trip change on the second?