Chapter 7

Greed Is Good? Prove It!

It’s not a question of enough, pal.

—Gordon Gekko, Wall Street

So-called greedy algorithms are short-sighted, in that they make each choice in isolation, doing what looks good right here, right now. In many ways, eager or impatient might be better names for them, because other algorithms also usually try to find an answer that is as good as possible; it’s just that the greedy ones take what they can get at this moment, not worrying about the future. Designing and implementing a greedy algorithm is usually easy, and when they work, they tend to be highly efficient. The main problem is showing that they do work—if, indeed, they do. That’s the reason for the “Prove It!” part of the chapter title.

This chapter deals with greedy algorithms that give correct (optimal) answers; I’ll revisit the design strategy in Chapter 11, where I’ll relax this requirement to “almost correct (optimal).”

Staying Safe, Step by Step

The common setting for greedy algorithms is a series of choices (just like, as you’ll see, for dynamic programming). The greed involves making each choice with local information, doing what looks most promising without regard for context or future consequences, and then, once the choice has been made, never looking back. If this is to lead to a solution, we must make sure that each choice is safe—that it doesn’t destroy our future prospects. You’ll see many examples of how we can ensure this kind of safety (or, rather, how we can prove that an algorithm is safe), but let’s start out by looking at the “step by step” part.

The kind of problems solved with greedy algorithms typically build a solution gradually. It has a set of “solution pieces” that can be combined into partial, and eventually complete, solutions. These pieces can fit together in complex ways; there may be many ways of combining them, and some pieces may no longer fit once we’ve used certain others. You can think of this as a jigsaw puzzle with many possible solutions (see Figure 7-1). The jigsaw picture is blank, and the puzzle pieces are rather regular, so they can be used in several locations and combinations.

Now add a value to each puzzle piece. This is an amount you’ll be awarded for fitting that particular piece into the complete solution. The goal is then to find a way to lay the jigsaw that gets you the highest total value—that is, we have an optimization problem. Solving a combinatorial optimization problem like this is, in general, not at all a simple task. You might need to consider every possible way of placing the pieces, yielding an exponential (possibly factorial) running time.
Let’s say you’re filling in the puzzle row by row, from the top, so you always know where the next piece must go. The greedy approach in this setting is as easy as it gets, at least for selecting the pieces to use. Just sort the pieces by decreasing value and consider them one by one. If a piece won’t fit, you discard it. If it fits, you use it, without regard for future pieces.

Even without looking at the issue of correctness (or optimality), it’s clear that this kind of algorithm needs a couple of things to be able to run at all:

- A set of candidate elements, or pieces, with some value attached
- A way of checking whether a partial solution is valid, or feasible

So, partial solutions are built as collections of solution pieces. We check each piece in turn, starting with the most valuable one, and add each piece that leads to a larger, still valid solution. There are certainly subtleties that could be added to this (for example, the total value needn’t be a sum of element values, and we might want to know when we’re done, without having to exhaust the set of elements), but this’ll do as a prototypical description.

A very simple example of this kind of problem is that of making change—trying to add up to a given sum with as few coins and bills as possible. For example, let’s say someone owes you $43.68 and gives you a hundred dollar bill. What do you do? The reason this problem is a nice example is that we all instinctively know the right thing to do here:1 we start with the biggest denominations possible and work our way down. Each bill or coin is a puzzle piece, and we’re trying to cover the number $56.32 exactly. Instead of sorting a set of bills and coins, we can think of sorting stacks of them, because we have many of each. We sort these stacks in descending order and start handing out the largest denominations, like in the following code (working with cents, to avoid floating-point issues):

---

1 No, it’s not to run away and buy comic books.