Problem Solving Techniques

Winter 2023

In order to design algorithms effectively, you will often be faced with problems that do not have easy, cookie-cutter solutions and will need to be able to come up with new ideas in order to be able to solve these problems. Unfortunately, this kind of problem solving skill is not often taught directly in curriculums. Partly, because many courses are focused on covering some particular material, but also because one of the best ways to learn problem solving is by working problems on your own (which is why CSE 101 tends to give challenging homework and exam problems).

However, if you don't have much background in this kind of problem solving, here are some techniques to get you started.

Step 1: Read the Problem

Seriously. Read the problem carefully. If you are spending a long time working on the wrong problem you can be wasting a lot of time without making progress towards a solution. Things to check:

- Do you know the meaning of all the words in the problem statement?
- Can you restate what is being asked for in your own words?
- Can you figure out what the problem is asking for in simple examples?
- Can you list any assumptions that you are allowed to make?

Step 2: Solve the Problem

How do you actually solve a problem once you understand it? Well... usually this involves a lot of trial and error. You try a bunch of ideas, most of which won't work. Hopefully, why they don't work teaches you something about the problem and you try again. Eventually, you build up enough tools and stumble onto enough of the right ideas that you solve it. As long as you are trying out new ideas (rather than just repeating the same things over and over), you are making progress.

But what do you do if you get stuck. Well here are some basic techniques to fall back on:

- Try working out small examples or special cases.
- Try designing *some* algorithm that works even if it is not fast enough (this will also often get you partial credit on homework/exam problems).
- Consider the toolbox of algorithm design techniques. Which ones might apply?
- Consider other algorithms you know. Does this problem look similar to any of them? Maybe one can be adapted to solve it, or maybe one can be used to help solve it.
- See what relevant things you *can* compute efficiently.
- Look for structure in the problem. Do solutions all have some property? Is there some equivalent quantity you can try to compute instead?

- Work backwards. What would allow you to solve this problem? Can you prove/compute that? What would allow you to do that?
- Use proof by contradiction: Try to construct a counterexample. You will probably fail, but the *reason* why you fail might tell you something about why the original thing you were trying to prove is true.
- Re-read the problem statement. Are there assumptions that you aren't currently using?
- Consider extremal cases. Look at the biggest/smallest object or the best/worst case. What does this tell you about the general case?
- Use induction.

Note: unless you are experienced with it, it is easy to write sloppy or incorrect inductive proofs. There is a standard format for writing inductive proofs that helps make them easier to read and harder to mess up. For most students, this format is recommended. It goes like this:

We will prove [[statement that you want to prove]] by induction on [[inductive variable]]
Base Case: [[state and prove the base case]]
Inductive Step: Assume [[inductive statement for n]]
[[prove inductive statement for $n + 1$]]
This completes our inductive step and finishes the proof of [[statement you want to prove]].