CPSC 320 Notes, Reductions and Resident Matching: A Residentectomy

January 6, 2017

A group of residents each needs a residency in some hospital. A group of hospitals each need some number (one or more) of residents, with some hospitals needing more and some fewer. Each group has preferences over which member of the other group they'd like to end up with. The total number of slots in hospitals is exactly equal to the total number of residents.

We want to fill the hospitals slots with residents in such a way that no resident and hospital that weren't matched up will collude to get around our suggestion (and give the resident a position at that hospital instead).

Definitions: An *instance* of a problem is a particular input drawn from the space of possible inputs the problem allows. For example, the 4-element array $[5, 1, 4, 3]$ is an instance of the problem of sorting arrays of integers.

A *reduction* from problem P_1 to problem P_2 is an algorithm that solves P_1 using (but without needing to define) an algorithm A_{P_2} that solves P_2. Often, we put some restriction on the number of calls to A_{P_2} (e.g., a polynomial number).

We'll call a *simple reduction* a reduction in which only one call is made to A_{P_2}. This is the most common case we'll run into, and it's often easiest to think about this case by proceeding through these steps:

1. Change an instance of P_1 into an instance of P_2 by hand.

2. Change the corresponding solution to P_2 into a solution to P_1 by hand.

3. Design an algorithm to change instances of P_1 into instances of P_2.

4. Design an algorithm to change corresponding solutions to P_2 into solutions to P_1.

5. Prove that if the solution to the P_2 instance is correct, so too is the solution your algorithm creates for the original P_1 instance. (**Or equivalently**, if the P_1 solution generated is incorrect, then the P_2 solution must have been incorrect as well.)

1 Trivial and Small Instances

1. Write down all the **trivial** instances of RHP. We think of an instance as "trivial" roughly if its solution requires no real reasoning about the problem.

2. Write down two **small** instances of RHP. Here's your first:

   The other can be even smaller, but not trivial:

3. Hold this space for another instance, in case we need more.

2 Represent the Problem

1. What are the quantities that matter in this problem? Give them short, usable names.

2. Go back up to your trivial and small instances and rewrite them using these names.

3. Use at least one visual/graphical/sketched representation of the problem to draw out the largest instance you've designed so far:

4. Describe using your representational choices above what a valid instance looks like:

3 Represent the Solution

1. What are the quantities that matter in the solution to the problem? Give them short, usable names.

2. Describe using these quantities makes a solution **valid** and **good**:

3. Go back up to your trivial and small instances and write out one or more solutions to each using these names.

4. Go back up to your drawn representation of an instance and draw at least one solution.

4 Similar Problems

Give at least one problem you've seen before that seems related in terms of its surface features ("story"), problem or solution structure, or representation to this one:

5 Brute Force?

We have a way to test if something that looks like a solution but may have an instability stable. (From the "Represent the Solution" step.) That is, given a "valid" solution, we can check whether it's "good".

1. Sketch an algorithm to produce every valid solution. (It will help to **give a name** to your algorithm and its parameters, *especially* if your algorithm is recursive.)

   This will be similar to the brute force algorithm for SMP (from the challenge problems).

2. Choose an appropriate variable to represent the "size" of an instance.

3. Exactly or asymptotically, how many such "valid solutions" are there? (It will help to **give a name** to the number of solutions as a function of instance size.)

4. Exactly or asymptotically, how long will it take to test whether a solution form is valid and good with a naïve approach? (Write out the naïve algorithm if it's not simple!)

5. Will brute force be sufficient for this problem for the domains we're interested in?

6 Promising Approach

Unless brute force is good enough, describe—in as much detail as you can—an approach that looks promising. **THIS TIME**, we use reduction:

1. Choose a problem P to reduce to.

2. Change an instance of RHP into an instance of P.

3. Change the solution to P into a solution to RHP.

4. Design an algorithm to change any valid RHP instance into an instance of P.

5. Design an algorithm to change the P instance's solution into the RHP instance's solution.

6. Prove that the solution you get to the RHP instance is guaranteed to be correct. (Depending on your chosen reduction, you likely have a stable solution to an instance of P and need to show that you get a correct solution to the RHP instance, i.e., that the solution is "valid" (has the right form) and "good" (is stable). So, you'll prove either *if P's solution is correct, RHP's solution is correct* or the contrapositive, if RHP's solution is *incorrect*, then P's is *incorrect* as well.)

7 Challenge Your Approach

1. **Carefully** run your algorithm on your instances above. (Don't skip steps or make assumptions; you're debugging!) Analyse its correctness and performance on these instances:

2. Design an instance that specifically challenges the correctness (or performance) of your algorithm:

8 Repeat!

Hopefully, we've already bounced back and forth between these steps in today's worksheet! You usually *will* have to. Especially repeat the steps where you generate instances and challenge your approach(es).

9 Challenge Problems

1. **EASY:** Create a variant of RHP in which hospitals can have too much or too little capacity and reduce it to SMP. Establish or refute the quality of your reduction (i.e., prove or disprove that stable solutions to the underlying problem produce stable solutions to the general RHP).

2. **EASY:** Establish that reductions are transitive.

3. **MEDIUM:** Using the USMP-SMP reduction from the worked example and the G-S algorithm where **men** propose first: prove that no woman who ends up unmarried could be married in **any** stable match.

4. **MEDIUM-HARD:** Create a reduction from the "truncated" stable marriage to SMP. In truncated stable marriage, anyone can leave off as many people from the other category as they wish, all of whom are assumed to be worse than everyone they listed. Analyse the properties of your reduction.