Lecture 14

Interlude: Problem Solving

Supplemental reading in CLRS: None

This lecture was originally given as a pep talk before the take-home exam. In note form, this chapter will be light reading, a break in which we look back at the course material as veterans.

14.1 What to Bring to the Table

In most technical undergraduate courses, questions are posed in such a way that all solutions are equally valid; if a student is able to come up with any solution to a given problem, she is considered to be in good shape. In 6.046, this is not the case. For most algorithmic problems, there is an obvious but inefficient solution; your task is to find a *better* solution. It is worth appreciating that this way of thinking may be new to you, and taking a moment to reflect metacognitively on strategies to help you solve algorithmic problems.

There is no recipe for solving problems, but there are many things you can do which often, eventually, lead to good solutions.

- *Knowledge.* In academic research, one will usually pick up the relevant background material over the course of working on a given problem. In an exam setting, you should come to the table with as complete an understanding of the background material as possible; you will not have time to learn many new concepts. In either case, you should strive to understand the relevant background as deeply as possible. You should be able to implement a given algorithm correctly in your computer language of choice. You should be able to answer CLRS-style exercises about the pseudocode: What would be the effect on performance if we changed this line? Would the algorithm still be correct? What would be the effect on performance if we used a different data structure?

- *Belief.* Be optimistic. Remind yourself that you might solve this problem, even if it doesn't look tractable at first.

- *Motivation.* There is no substitute for a strong motivation to solve the problem at hand. Given that you're trying to learn algorithms from a set of lecture notes, I probably don't even need to tell you this.

14.2 How to Attack a Problem

We'll now discuss several tips that might help you to crack a problem open. We'll keep the following concrete example in the back of our mind:

Problem 14.1 (Bipartite Matching)**.** In a group of n heterosexual people, each woman has a list of the men she is interested in marrying, and each man has a list of the women he is interested in marrying. Find an algorithm that creates the largest possible number of weddings. (Each person can marry at most one other person. In a couple, each member must be on the other member's list.)

14.2.1 Understand the Problem

- Note everything that is given.
- Find some upper bounds and lower bounds. For example:
  - If the algorithm has to read the entire input, then the running time is at least $\Omega(n)$. (This is not always the case, however! For example, binary search runs in $O(\lg n)$ and does not read the entire input.)
  - If the output has to output k things, then the running time is at least $\Omega(k)$.
  - If the algorithm can be used to perform a comparison-based sort, then the running time is at least $\Omega(n \lg n)$.[1]
  - There are usually a finite number of possibilities for the output. In the case of our current problem, arrange the men in a line (by alphabetical order of last name, say). For each man who has a wife, place his wife behind him. Place the unmarried women at the front of the line (by alphabetical order of last name, say). In this way, each possible matching gives a different line. Thus, the total number of possible matchings is at most $n!$, the number of ways to arrange all n people in a line.
  - Usually there is an obvious algorithm that runs in exponential time. Sometimes it runs faster.
  - Sometimes it is easy to reduce the problem to another problem whose solution is well-known. (Be careful, though—your special case may be amenable to a more efficient solution than the general problem!)
  - Whatever your solution is, its running time must of course fall within the lower and upper bounds you find.
- Is there a useful diagram?

[1] A *comparison-based sorting algorithm* is a sorting algorithm which makes no assumptions about the items being sorted except that there is a well-defined notion of "less than" and that, for objects a and b, we can check whether $a < b$ in constant time. The fact we are using here is that any comparison-based sorting algorithm takes at least $\Omega(n \lg n)$ time. The proof is as follows. Suppose we have a comparison-based sorting algorithm A which takes as input a list $L = \langle a_1, \ldots, a_n \rangle$ and outputs a list of indices $I = \langle i_1, \ldots, i_n \rangle$ such that $a_{i_1} \leq \cdots \leq a_{i_n}$. Let k be the number of comparison operations performed by A on a worst-case input of length n (where k depends on n). Because each comparison is a yes-or-no question, there are at most 2^k possible sets of answers to the at most k comparison operations. Since the output of a comparison-based sorting algorithm must depend only on these answers, there are at most 2^k possible outputs I. Of course, if A is correct, then all $n!$ permutations of the indices $\langle 1, \ldots, n \rangle$ must be possible, so $2^k \geq n!$. Thus $k \geq \lg(n!)$. Finally, Stirling's approximation tells us that $\lg(n!) = \Theta(n \lg n)$. Thus the running time of A is $\Omega(k) = \Omega(n \lg n)$.

- Is there a graph?
- Try examples!

14.2.2 Try to Solve the Problem

- Can we solve the problem under a set of simplifying assumptions? (What if every vertex has degree 2? What if the edge weights are all 1?)
- Are there useful subproblems?
- Will a greedy algorithm work? Divide-and-conquer? Dynamic programming?
- Can this problem be reduced to a known problem?
- Is it similar to a known problem? Can we modify the known algorithm to suit it to this version?
- Can the problem be solved with convolution? If so, the fast Fourier transform will improve efficiency.
- Is there any further way to take advantage of the given information?
- Can randomization help? It is often better to find a fast algorithm with a small probability of error than a slower, correct algorithm.

Remember that *a hard problem usually cannot be solved in one sitting*. Taking breaks and changing perspective help. In the original setting of a pre–take-home exam pep talk, it was important to remind students that "incubation time" is completely necessary and should be part of the exam schedule. Thus, students should start the test early—the away-time when they are not actively trying to solve test questions is crucial.

14.2.3 Work Out the Details

- What is the algorithm? (Write the algorithm down in as much detail as possible—it is easy to end up with an incorrect algorithm or an incorrect running time analysis because you forgot about a few of what seemed like unimportant implementation details.)
- Why is it correct?
- What is its asymptotic running time?

14.2.4 Communicate Your Findings

- Don't be shy about writing up partial solutions, solutions to simplified versions of the problem, or observations. This is the most common sort of writing that occurs in academia: often, the complete solution to a problem is just the finishing touch on a large body of prior work by other scientists. And even more often, no complete solution is found—all we have are observations and solutions to certain tractable special cases. (In an exam setting, this sort of writing could win you significant partial credit.)

- To be especially easy on your reader, you can format your algorithm description as an essay, complete with diagrams and examples. The essay should begin with a high-level "executive summary" of how the algorithm works.

Now that you have been thinking about the bipartite matching problem for a while, here is a solution:

Algorithm:
Let M be the set of men and let W be the set of women; let s and t be objects representing a source and a sink. Create a bipartite directed graph* G with vertex set $M \cup W \cup \{s,t\}$. Draw an edge from s to each man and an edge from each woman to t. Next, for each man m and each woman w, draw an edge from m to w if m is on w's list and w is on m's list. Give all edges capacity 1. The graph G is now a flow network with source s and sink t (see Figure 14.1).

Note that there is a bijection between valid matchings and integer flows in G. In one direction, given a matching, we can saturate each edge between a man and a woman who have been matched. The flows out of s and into t are then uniquely determined. In the other direction, given an integer flow f in G, the fact that each edge out of s or into t has capacity 1 means that there is at most one edge out of each man and at most one edge into each woman. Thus, we obtain a valid matching by wedding a man m to a woman w if and only if $f(m,w) = 1$. The magnitude of f is equal to the number of couples, so the number of couples in any matching is bounded above by the maximum flow in G.

This bound is tight if and only if there exists a maximum flow f^* in G which is an integer flow. Such a flow does exist, and is found by the Ford–Fulkerson algorithm, by Proposition 13.3. Since the number of couples is at most $O(n)$, we have $|f^*| = O(n)$, and the running time of the Ford–Fulkerson algorithm is $O(E \cdot |f^*|) = O\left(n^2 \cdot n\right) = O\left(n^3\right)$. Actually, assuming each person has a relatively small list of acceptable spouses, we ought to be more granular about this bound: we have

$$E = O\left(n + \sum_{i=1}^{n} \begin{pmatrix}\text{size of the } i\text{th} \\ \text{person's list}\end{pmatrix}\right),$$

and the running time is at most

$$T = O\left(n \cdot \left(n + \sum_{i=1}^{n} \begin{pmatrix}\text{size of the } i\text{th} \\ \text{person's list}\end{pmatrix}\right)\right).$$

* A graph $G = (V,E)$ is called **bipartite** if the vertex set V can be written as the union of two disjoint sets $V = V_1 \sqcup V_2$ such that every edge in E has one endpoint in V_1 and one endpoint in V_2.

14.2.5 Reflect and Improve

- Can we achieve a better running time?[2]

[2] For the bipartite matching problem, there exist better algorithms than the one given above. Still, I love this solution. It shows how useful flow networks are, even in contexts that seem to have nothing to do with flow. There are many more examples in which this phenomenon occurs.

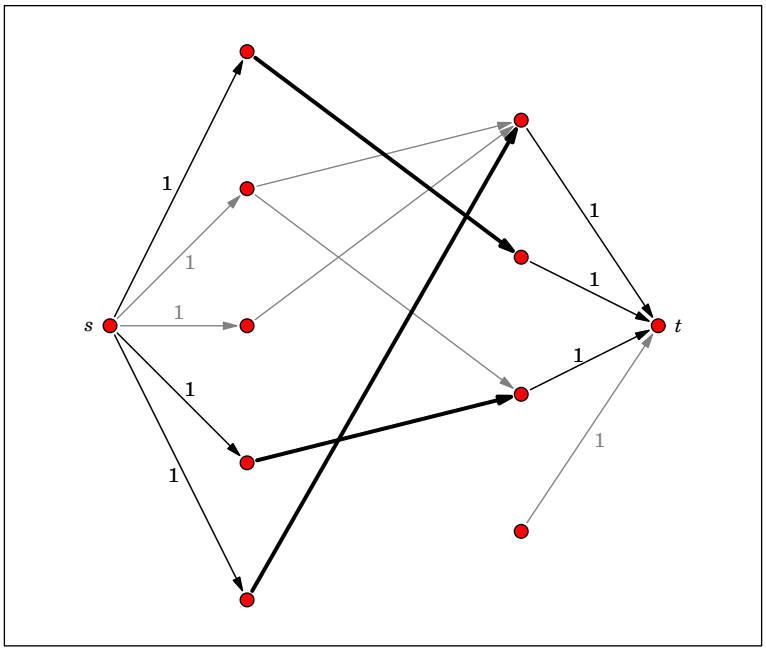

Figure 14.1. In this flow network, the nodes on the left (other than s) represent women and the nodes on the right (other than t) represent men. All edges have capacity 1. A maximum flow is highlighted in black; it shows that the maximal number of couples is 3.

- Can randomization help?
- Can amortization help?
- Can we use less space?

14.3 Recommended Reading

There are many great references and exercise books for algorithmic problem solving and mathematical problem solving in general. Two particular selections are:

- George Pólya, *How to Solve It*
- *The Princeton Companion to Mathematics*, VIII.1: "The Art of Problem Solving."

Another good source of algorithms practice is web sites designed to help people prepare for technical interviews. It is common for tech companies to ask algorithm questions to job applications during interviews; students who plan to apply for software jobs may especially want to practice. Even for those who aren't looking for jobs, web sites like TechInterview.org can be a useful source of challenging algorthmic exercises.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012