Introduction to Analysis of Algorithms — Prelim 1 Review
CS4820 Spring 2013 — Thursday, February 21, 2013

Prelim 1 focuses on three topics:

- greedy algorithms,
- divide-and-conquer algorithms, and
- dynamic programming.

The prelim tests for two things. First of all, several algorithms were taught in the lectures and assigned readings, and the prelim will test your knowledge of those algorithms. Secondly, the first part of the course has taught concepts and techniques for designing and analyzing algorithms, and the prelim will test your ability to apply those concepts and techniques. CS 4820 emphasizes this second type of learning (applying general concepts and techniques for designing and analyzing algorithms) much more than the first type of learning (memorizing specific algorithms) and accordingly, the prelim will place much more emphasis on testing concepts and techniques than on testing your knowledge of the specific algorithms that were taught so far. In particular, *you are not responsible for memorizing any algorithms except the algorithms for stable matchings, minimum spanning trees, and shortest paths.* Those algorithms—Gale–Shapley, Kruskal, Prim, and Bellman–Ford—are central to algorithm design, so you should know how they work, know their running time, and be prepared to run the algorithms by hand on simple inputs. Concerning the other algorithms from the lectures and assigned readings, you are only responsible for general knowledge of what type of algorithms they are. For example, you should know that unweighted interval scheduling can be solved by a greedy algorithm but weighted interval scheduling requires dynamic programming. For the record, the following is a list of algorithms that were taught so far.

Stable matching: Gale–Shapley.

Greedy: Unweighted interval scheduling, minimum spanning tree (Kruskal, Prim, reverse delete).

Divide and conquer: Closest pair of points, fast polynomial multiplication.

Dynamic programming: Weighted interval scheduling, sequence alignment (aka edit distance), Bellman–Ford shortest path algorithm, RNA secondary structure optimization.

In the course of going over these algorithms, we covered several other related topics that are fair game for the prelim.

- Basic properties of minimum spanning trees. (Example: for any partition of the vertices into two nonempty sets, the MST contains the min-cost edge joining the two pieces of the partition.)
- Fast data structures: priority queues and union–find. You will not be responsible for knowing how they are implemented, only for knowing what operations they implement and the running time per operation.
- Solving recurrences. (You will not have to solve anything more complicated than the ones discussed in §5.1 and §5.2 of K&T.)

As stated earlier, the most important part of the prelim is applying the general concepts and techniques we've been teaching. These break down into: designing algorithms, analyzing their running time, and proving their correctness. We address each of these topics below.

Designing algorithms. A basic question to consider when approaching an algorithm design problem is, "What type of algorithm do I think I will need to design?" On this prelim, the answer will be one of four options: greedy, divide and conquer, dynamic programming, or a reduction to an already-solved problem (such as minimum spanning tree or stable matching). Here are some observations that could help with the basic process of designing the algorithm.

- Start out by trying the greedy algorithm and seeing if you can find a counterexample. (By the way, one type of question that might be asked on a prelim is to present an incorrect algorithm for a problem and ask you to provide a counterexample showing that the algorithm is incorrect.) If you can't come up with a counterexample, try proving the greedy algorithm is correct.
- If the greedy algorithm is incorrect, the next thing to try is dynamic programming. The thought process of designing a dynamic programming algorithm can be summarized as follows.
    1. What is the last decision the algorithm will need to make when constructing the optimal solution?
    2. Can that decision be expressed as a simple minimization or maximization, assuming some other information was already precomputed and stored in a table somewhere?
    3. What extra information needs to be precomputed and stored in a table? What is the structure of the table? E.g., is it a one-dimensional or multi-dimensional array? Are its entries indexed by natural numbers, vertices of a graph, some other index set?
    4. How must the table be initialized, and in what order do we fill in the entries after initialization?
- Divide and conquer algorithms tend to be applicable for problems where there is an obvious solution that doesn't use divide-and-conquer, but its running time is too inefficient compared to what the problem is asking for.

Instead of designing the algorithm from scratch, another option is to use a known algorithm for some other problem. On this prelim there won't be any problems that *require* you to do this, but you have license to do so if you find it to be the easiest approach.

Among the algorithms we've seen so far this semester, Bellman–Ford and Dijkstra are by far the best "workhorse" algorithms; a surprisingly wide variety of other problems can be expressed as shortest path problems in graphs. So if there is a problem on the prelim that can be solved by using one of the algorithms you've already learned, most likely it will be shortest paths, which you can then solve using Bellman–Ford or Dijkstra. By the way, if solving a problem involves using an algorithm that was already taught, your pseudocode is allowed to contain a step that says, for example, "Run the Bellman–Ford algorithm." You don't need to repeat the entire pseudocode for that algorithm.

If you reduce your problem to a known problem, then to prove correctness, there are basically three steps.

1. If you are using problem Foo, prove that your algorithm creates a valid instance of Foo. For example, if you are using Bellman–Ford, prove that you are applying it to a graph with no negative-cost cycles.
2. Prove that the solution of Foo can be transformed into a valid solution of the original problem. (For example, if transforming a path in some graph back to a solution of the weighted interval scheduling problem, make sure to show that the resulting set of intervals has no conflicts.)
3. Prove that this transformation preserves the property of being an optimal solution. (For example, prove that a min-cost path transforms to a max-weight schedule.) Often this step is easy.

Analyzing the running time of algorithms almost always boils down to counting loop iterations or solving a recurrence. We've already discussed solving recurrences above. The only other case of runtime analysis that deserves special mention is when you are using a known algorithm. In that case, you must account for

1. the amount of time it takes to transform your problem into an instance of some other problem,
2. the time it takes to run the subroutine that solves that problem,
3. the time to transform the solution back to a solution of the original problem.

When accounting for step 2, recall that you need to account for the size of the transformed problem, and "plug in" those parameters into the expression for the running time of the algorithm that you're using in step 2. For example, if you are working on a problem that concerns n subintervals of a timeline containing T

time steps, and your reduction constructs a graph with n^2 vertices and nT edges and then runs Bellman–Ford on that graph, then the running time of that step is $O(n^3T)$. The reasoning is as follows. Bellman–Ford on a graph with n vertices and m edges runs in time $O(mn)$. Let's rewrite the number of vertices and edges as V and E, to avoid confusion with the parameter n that refers to the number of subintervals in the original problem instance. (Confusion over the meaning of the letter n in the original problem versus the transformed problem is the cause of a surprising number of mistakes in this class!) Then the running time of Bellman–Ford is expressed as $O(VE)$, and the reduction produces a graph with $V = n^2$ and $E = nT$, hence $O(VE) = O(n^2 \cdot nT) = O(n^3T)$.

Finally, we come to the issue of proving correctness. For every style of algorithm that you've learned so far, there is a prototypical style of correctness proof. Actually, for greedy algorithms, there are two prototypical correctness proofs. Here is a bare-bones outline of the steps in each style of proof.

Proving correctness of greedy algorithms using "greedy stays ahead." The algorithm makes a sequence of decisions—usually, one decision for each iteration of the algorithm's main loop. (A decision could be something like scheduling a job or selecting an edge of a graph to belong to a spanning tree.) We compare the algorithm's solution against an alternative solution that is also expressed as a sequence of decisions. The proof defines a measure of progress that can be evaluated each time one decision in the sequence is made. The proof asserts that for all k, the algorithm's progress after making k decisions is better than the alternative solution's progress after k decisions. The proof is by induction on k.

Proving correctness of greedy algorithms using exchange arguments. The proof works by specifying an "exchange" operation that can be applied to any solution that differs from the one produced by the greedy algorithm. The proof shows that this exchange never makes the solution quality worse, and the solution after the exchange is "closer" to the greedy solution. This requires defining what "closer" means. The algorithm's correctness then follows by induction. Starting from an optimal solution, if that solution is not the same as the greedy solution, one can transform it by a sequence of exchanges to the greedy solution without destroying optimality, so the greedy solution is also optimal.

Proving correctness of divide and conquer algorithms. The proof is always by strong induction on the size of the problem instance. The induction step requires you to show that, if we assume each recursive call of the algorithm returns a correct answer, then the procedure for combining these solutions results in a correct answer for the original problem instance.

Proving correctness of dynamic programs. The proof is by induction on the entries of the dynamic programming table, in the order that those entries were filled in by the algorithm. The steps are as follows. (As a running example, assume a two-dimensional dynamic programming table. The same basic outline applies regardless of the dimensionality of the dynamic programming table.)

1. Somewhere in your proof, you should define what you think the value of each entry in the table means. (Example: "$T[i, j]$ denotes the minimum number of credits that must be taken to fulfill the prerequisites for course i by the end of semester j.")
2. The induction hypothesis in the proof is that the value of $T[i, j]$ computed by your algorithm matches the definition of $T[i, j]$ specified earlier.
3. Base case: the values that are filled in during the initialization phase are correct, i.e. they satisfy the definition of $T[i, j]$.
4. Induction step: the recurrence used to fill in the remaining entries of T is also correct; that is, assuming that all previous values were correct, then the recurrence gives the correct formula to fill in the current value.
5. Remaining important issue: prove that the table is filled in the correct order; that is, when computing the value of $T[i, j]$, the algorithm only looks at table values that were already computed in earlier steps.