Introduction to Algorithms	**Review information for Prelim 1**
CS 4820, Spring 2010	**Distributed Wednesday, February 24**

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 minimum spanning trees and shortest paths.* Those algorithms — Kruskal, Prim, and Bellman-Ford — are so central to algorithm design that 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, knowing 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.

Greedy: Unweighted interval scheduling, scheduling to minimize lateness, minimum spanning tree (Kruskal & Prim).

Divide and conquer: Closest pair of points, fast polynomial multiplication, cache-oblivious matrix transposition.

Dynamic programming: Weighted interval scheduling, edit distance, Bellman-Ford shortest path algorithm, RNA secondary structure.

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 Sections 5.1 and 5.2 of the book.)
- The cache-oblivious model. You should know the basic definitions and assumptions, and you should be able to design simple cache-oblivious algorithms and analyze their cache complexity.

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 or cache complexity, 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 either: greedy, divide and conquer, or dynamic programming. Here are some observations that could help with the basic process of designing the algorithm.

- I always start out by trying the greedy algorithm and seeing if I can find a counterexample. (By the way, one type of question that I might ask 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 I can't come up with a counterexample, I try proving the greedy algorithm is correct.
- If the greedy algorithm is incorrect, the next thing I 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 trivial minimization or maximization, assuming some other information was already pre-computed and stored in a table somewhere?
  3. What extra information needs to be pre-computed and stored in a table? What is the structure of the table? (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 or cache complexity is too inefficient compared to what the problem is asking for.

Instead of designing the algorithm from scratch, another option is to reduce to a known algorithm for some other problem. On this prelim there won't be any problems that *require* you to design a reduction, but you have the right to do so if you find it to be the easiest approach. One bit of advice on reductions: among the algorithms we've seen so far this semester, the Bellman-Ford algorithm is by far the best "workhorse" algorithm; a surprisingly wide variety of other problems can be reduced to shortest path problems in graphs with no negative-cost cycles. So if there is a problem on the prelim that can be solved by reducing to one of the algorithms you've already learned, most likely it will be a reduction to shortest paths, which you can then solve using Bellman-Ford. By the way, if solving a problem using a reduction to 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. When proving the correctness of a reduction, there are basically three steps.

1. If you are using a reduction to problem Foo, prove that your reduction creates a valid instance of Foo. For example, if you are reducing to a shortest-path problem and then

using the Bellman-Ford algorithm, prove that your reduction creates a graph with no negative-cost cycles.

2. Prove that every solution of problem 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 earlier on this information sheet. In general, this is an easy process. Just be careful about the *base case* of the recurrence. Ordinarily the base case refers to the case when the input size is $O(1)$, and the algorithm's running time is also $O(1)$, and there's nothing else to say about the base case. However, analyzing cache complexity in the cache-oblivious model is different. To figure out the base case of the recurrence, you need to ask yourself when the problem size is small enough that the entire algorithm can execute without any cache misses, other than the inevitable cache misses that happen when loading the input into cache and writing the output back to memory. Thus, the base case of the recurrence is often an important step in analyzing the cache complexity of algorithms and it should not be overlooked.

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 is associated with 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 a "local improvement" operation that can be applied to any solution that differs from the one produced by the greedy algorithm. The proof shows that this local improvement never makes the solution quality worse, and if the solution quality is exactly the same before and after performing the local improvement, then the solution after the local improvement is "more similar" to the greedy solution. (This requires defining what "more similar" means.) The algorithm's correctness is then established using a proof by contradiction. If the greedy solution is suboptimal, then there is an alternative solution whose quality is better. Among all the alternative solutions of optimal quality, let x be the one that is most similar to the greedy solution. Using a local improvement operation, we can either improve the quality of x of preserve its quality while making it more similar to the greedy solution. This contradicts our choice of x.

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 always 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 always as follows. (As a running example, I'll 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, i.e. assuming that all previous values were correct, then we're using the correct formula to fill in the current value.
5. Remaining important issue: prove that the table is filled in the correct order, i.e. that when computing the value of $T[i, j]$, the algorithm only looks at table values that were already computed in earlier steps.