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

The final exam will cover seven topics.

1. greedy algorithms
2. divide-and-conquer algorithms
3. dynamic programming
4. network flow
5. NP-completeness
6. Turing machines and undecidability
7. approximation algorithms

The exam tests for three things. First of all, several algorithms were taught in the lectures and assigned readings, and the final will test your knowledge of those algorithms. Secondly, the course has taught concepts and techniques for designing and analyzing algorithms, and the final will test your ability to apply those concepts and techniques. Thirdly, the course has covered techniques for proving that a problem is hard, and the final will test your ability to show that a problem is NP-complete or undecidable.

The final will place much more emphasis on testing concepts and techniques than on testing your knowledge of the specific algorithms that were taught so far, but a few algorithms (Kruskal, Prim, Bellman-Ford, Ford-Fulkerson) are so central to algorithm design that you should know how they work, know their running times, 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.

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

- 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.)
- Flow related topics. You should be familiar with residual graphs, augmenting paths, s-t cuts, the max-flow min-cut theorem, and the flow integrality theorem.

- Rice's theorem (undecidability). You should know what the theorem says. You won't be responsible for knowing the details of the proof.

As stated earlier, the most important part of the final is applying the general concepts and techniques we've been teaching.

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?" Here are some observations that could help with the basic process of designing the algorithm.

- Greedy algorithms and dynamic programming algorithms tend to be applicable in cases where the problem has something to do with ordered sequences, paths, or trees. 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 an exam 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, and the problem involves ordered sequences, paths, or trees, 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? (It often involves deciding something at the end of an ordered sequence, last hop of a path, or root of a tree.)
    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?

- A final type of problem that lends itself to dynamic programming is *problems with numerical values in the input*, such as Subset Sum or Knapsack. (See Section 6.4 of Kleinberg & Tardos.) The dynamic programming algorithms for these problems use a table whose size scales with the magnitude of the numbers specified in the input. Thus, the algorithm has exponential running time in general, but runs in polynomial time when the absolute values of the numbers occurring in the input are bounded by a polynomial function of the input size.

- Divide and conquer algorithms tend to be applicable for problems where there is an obvious solution that doesn't use divide-and-conquer, but it is too inefficient compared to what the problem is asking for.

- Flow reductions tend to be applicable for problems that involve assigning objects of type A to objects of another type B, like professors to committees or wins to baseball teams. In general, the objects of type A must all be of the same size. (Unless you are allowed to split an object of type A into more than one piece and assign the pieces to different type B objects.) The load balancing problem, where you try to assign jobs of different sizes to different machines so that no machine has too large a load, is an NP-complete problem(described in 11.1), so trying to solve it by reducing to network flow will not work.

- When none of the previous strategies work, it is likely that the problem is NP-complete. NP-complete problems have the property that solutions to subproblems can't be combined efficiently to create an answer to the original problem. If a problem is NP-complete, don't try to design an algorithm to solve it. Instead prove that it is NP-complete by pretending that there is an algorithm for solving it and showing that under that assumption some NP-complete problem would be solvable in polynomial time. (For details see the section on NP-completeness below).

- While NP-complete problems are believed not to have efficient algorithms in the general case, narrowing the types of input allowed can make them solvable. If you are asked to solve an NP-complete problem but on a very restricted input (like "trees") you should try dynamic programming first.

- Although we don't know any computationally efficient procedure for solving NP-complete problems, at least there is a "brute-force approach" that is guaranteed to work. In contrast, undecidable problems cannot be solved by any algorithm. Undecidable problems are typically, but not exclusively, of a "meta-computational" nature. In other words, rather than asking a question about sets, graphs, numbers, sequences, or whatever, they ask a question in which *the input specifies an algorithm (i.e., Turing machine)* and the question concerns *what happens when executing that algorithm on some input.*

Designing an algorithm that approximately solves an NP-complete problem. We covered 3 main methods of designing approximation algorithms in class:

- If there is an obvious greedy approach, try to find a way to bound the answer given by the greedy algorithm. An upper or lower bound might be based on a trivial property (like the number of edges) or on something that is easy to compute (like the minimum spanning tree) or on a property that is revealed in the course of running your greedy algorithm (like the number of edges in the matching chosen by the "conservative" algorithm for Minimum Vertex Cover).

- Designing a linear program to approximate an NP-complete problem is a matter of defining the "decision variables" and then figuring out how to encode the problem constraints as linear equations and inequalities. The hard part here is to avoid the temptation to use nonlinear functions. This can be avoided by defining extra variables. The extra variables, and the constraints they participate in, behave like "gadgets" in a reduction. Note, the linear program gives an approximation; an integer program could give an exact answer, but solving general integer programs is also NP-complete.

- Designing a polynomial-time approximation scheme is a good idea for numerical problems where the problem is only NP-complete because of the exponentially large numerical values that might occur in the input. Designing a polynomial-time approximation scheme has 6 steps:

1. Round the numerical values to multiples of some common value b.
2. Design an algorithm, usually a dynamic program, that finds the exact optimum when all the values are multiples of b. The running time of this step is allowed to depend on the size of the largest number divided by b.
3. Figure out how much additive error is introduced by the rounding.
4. Set the value of b so that this additive error amounts to only ε fraction of the optimum solution value.
5. Plug this value of b into the running time analysis of the dynamic program, to obtain the final running time analysis of your algorithm.
6. Prove correctness by justifying the estimates in steps (3) and (4) as well as provimg the correctness of the dynamic program in step (2).

Step (4) often requires a little bit of extra thought. How do you compare the rounding error to the optimum value, when the algorithm can't compute the optimum value? To do this one commonly needs to compute a simple upper or lower bound on the optimum solution value, similar to the way we proceed when designing and analyzing greedy approximation algorithms.

Using reductions to design algorithms.

The problems you are likely to want to reduce to are Bellman-Ford and the max flow algorithms (Ford-Fulkerson, Edmonds-Karp, preflow-push). One bit of advice on reductions: as you have probably discovered by now, 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.

Problems that can be solved using reductions can be broken into two categories.

1. problems that must have at least one solution (the goal is to find the best solution)
2. problems that may or may not have a solution (the goal is to either correctly say that there is no answer or to give one example of a correct answer).

Proofs of correctness for these two types of reductions are different. If the goal is to find the optimal solution, the reduction's proof of correctness will have 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.

If the goal is to decide whether there is an answer or not, then the reduction's proof of correctness will have 2 important steps:

1. Prove that if there is a solution to the original problem, that solution can be transformed into a valid solution to the translation of the original problem
2. Prove that if there is a solution to the translation of the original problem then that solution can be transformed into a valid solution to the original problem

Running Times

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.

When an algorithm is based on reducing to some other problem (e.g. network flow), the process of analyzing its running time is different. You need to do the following steps.

1. Account for the amount of time to transform the given problem instance into an instance of some other problem (e.g. a shortest-path problem, in the case that you're reducing to the Bellman-Ford algorithm).
2. Account for the amount of time it takes to run the other algorithm (e.g. Bellman-Ford) on the transformed problem instance.
3. Express the overall running time in terms of the size *original problem instance*, not the transformed problem instance created by your reduction.

For example, if your reduction takes a scheduling problem with J jobs and T time slots, and transforms it into a flow network with $n = 2J + T$ vertices and $m = JT + J^2$ edges, and then you compute a maximum flow using the Edmonds-Karp algorithm, whose running time is $O(m^2 n)$, then the running time of your algorithm should be expressed as $O\left((JT + J^2)^2 \cdot (2J + T)\right)$, rather than $O(m^2 n)$.

Proving Correctness

Next, we come to the issue of proving correctness of algorithms. For every style of algorithm that you've learned so far, there is a prototypical style of correctness proof. 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.

Proving correctness of reductions to max flow (and min cut). Above, in the discussion of using reductions to design algorithms, we've already addressed the issue of how to prove correctness of a reduction. For feasibility problems (i.e. problems that ask you whether there exists a solution satisfying some constraints) the proof consists of proving both directions of the statement: there is a solution to the original problem iff there is a solution to the translated version of the problem. The first direction is generally easier. It consists of transforming an arbitrary solution to the original problem into a valid solution to the flow instance of the problem. To do this, assume that a solution to the original problem exists. Describe how to build a flow solution from it, and argue that the flow solution is valid (i.e. obeys capacity constraints, obeys flow conservation, and has the desired flow value). The second direction is generally slightly harder. It consists of transforming an arbitrary solution to the flow instance of the problem into a valid solution to the original problem. It often requires you to use the fact that if all the capacities are integers then there is an integer flow. To do this, assume that a solution to the flow instance exists. Use it to construct a solution to the original problem. Argue that the solution obeys all the required constraints. Reductions to min cut are similar.

NP-completeness proofs

Sometimes the goal is not to design an algorithm, but to prove that designing an algorithm is the wrong strategy. In NP-completeness problems the goal is to prove that the given problem is at least as hard as other NP-complete problems (for which no polynomial algorithms are known). The proof that a problem Foo is NP-complete has 5 steps:

Foo is in NP: Show that there is a polynomial time verifier for Foo. The polynomial time verifier takes an instance of the problem Foo and a possible answer to that instance and, in polynomial time, answers 'yes' if the possible answer is a correct solution and 'no' if the possible answer is not a correct solution.

Design a reduction FROM some NP-complete problem TO Foo: Pick an NP-complete problem (more on this below). Use an oracle that solves Foo to solve the NP-complete problem. Describe how to translate an arbitrary instance of the NP-complete problem into an equivalent instance of Foo. Sometimes the translation will be straightforward and will basically consist of mapping a type of object in one problem to a type of object in the other problem (like in the dodgeball problem where we map a pair of team assignments to a color). Other times the reduction will require the construction of gadgets (like in the reduction from 3-SAT to IND-SET). When reducing from 3-SAT in general, there will be gadgets for binary decisions, and gadgets for enforcing the "three-strikes-and-you're-out" rule for clauses.

The reduction takes polynomial time: Calculate the running time of the reduction. Make sure that you aren't creating exponentially many nodes (or actors or routers...).

Translate arbitrary soln to NP-complete problem instance into soln to Foo instance: This is generally the easier direction.

Translate arbitrary soln to Foo instance into soln to NP-complete problem instance: This is generally the harder direction.

Don't reduce the wrong way. Remember, you want to reduce some NP-complete problem to the target problem. If there is a way to reduce the NP-complete problem to the target problem, then the target problem is at least as difficult as the NP-complete problem which is what we want. If you reduce the wrong way (i.e. design an algorithm that solves the target problem using an oracle for an NP-complete problem) then you are proving that the NP-complete problem is at least as difficult as the target problem, which in most cases was already obvious.

At the risk of being repetitive, let me say this another way. When a problem says, "Prove the FOO is NP-complete," the problem is **not** asking you to give an algorithm to solve FOO. It is not even asking you to show how FOO can be solved, assuming that you already have a subroutine to solve 3SAT (or HAMILTONIAN PATH, or whatever). **When you are asked to prove that Foo is NP-complete, it means you should present an algorithm for solving *some other* NP-complete problem, such as 3SAT, assuming that you *already have an algorithm that solves Foo.***

Choosing a problem to reduce from. NP-complete problems tend to break up into a few general categories. If you can identify the category of a problem, that will tell you which NP-complete problems are most similar. These similar problems are often the easiest to reduce from. Below is a list of the general categories with a few of the examples which are most likely to be useful to you.

1. Packing problems (Independent Set)
2. Covering problems (Vertex Cover, and Set Cover)
3. Partitioning problems (Graph Coloring)
4. Sequencing problems (Hamiltonian Path)
5. Numerical problems (Subset Sum)
6. Constraint satisfaction problems (3-SAT)

For a more complete list see the Partial Taxonomy of Hard Problems in section 8.10 of the text book.

If no problem is obviously a trivial translation, think about what gadgets you can make and what kinds of constraints those gadgets could enforce (and therefore what problems the target problem can mimic).

Some other tips on choosing what problem to reduce from.

1. If a problem asks you to decide if there exists a set of *at least* k objects satisfying some properties, try reducing from another problem that involves picking at least k objects, e.g. INDEPENDENT SET or CLIQUE.
2. Similarly, if a problem asks you to decide if there exists a set of *at most* k objects satisfying some properties, try reducing from another problem that involves picking at most k objects, e.g. VERTEX COVER or SET COVER.
3. When a problem doesn't easily fit into any of the general categories listed above, the most useful starting point for reductions is the 3SAT problem. For some reason, it's unusually easy to design gadgets reducing 3SAT to other problems that don't bear any obvious relation to Boolean variables and clauses.

Undecidability

You may be asked to answer a question about Turing machines, languages, and decidability. These questions should be easy to distinguish from other types of questions. The two types of questions you may encounter are the following.

Judging if a language is recursive, r.e., or neither. Judging which of the three alternatives usually boils down to:

1. Can I decide whether or not x belongs to $\mathcal{L}$, by simulating a computation that is guaranteed to finish in a bounded amount of time? (In other words, there is some finite time such that, if the computation hasn't finished by then, I can be sure it's never going to finish.) If so, $\mathcal{L}$ is recursive.
2. Can I simulate a computation such that, if something happens during the simulation, then I know for sure that x is in $\mathcal{L}$, but there's no way to know in advance how long I have to wait for that thing to happen? If so, $\mathcal{L}$ is r.e.
3. If there's no finite amount of evidence that could convince me that x is in $\mathcal{L}$, then $\mathcal{L}$ is neither r.e. nor recursive.

   (Example, suppose $\mathcal{L}$ is the set of all $M; x$ such that M does not halt when its input is x. There's no finite amount of evidence I could provide you with, that would convince you that M is never going to halt on input x. So this set is neither r.e. nor recursive.)

Proving a language is recursive, r.e., or neither. The proof works either by constructing a Turning machine which solves the problem or by assuming the existence of a Turning machine which solves the problem and using it to solve either the Halting problem or its complement.

1. Proving that a language $\mathcal{L}$ is r.e. or recursive boils down to describing, in plain English, a computation that accepts or decides $\mathcal{L}$. To do this, you're allowed to embed useful things such as a universal Turing machine as subroutines of your computation. You're also allowed to assume that your Turing machine has any (bounded, independent of the input size) number of extra tapes on which to store auxiliary information, e.g. a counter that counts the number of steps in a simulation that a universal Turing machine is running.
2. Proving that $\mathcal{L}$ is not recursive usually boils down to reducing **FROM** the halting problem to $\mathcal{L}$.
3. Proving that $\mathcal{L}$ is not even r.e. usually boils down to reducing **FROM** the complement of the halting problem to $\mathcal{L}$.

   In both cases, the standard advice about reductions (see NP-completeness section) applies. The difference is that instead of gadgets (as in NP-completeness reductions) the notion of "reduction" here tends to revolve around tweaking a universal Turing machine simulation so that some "taboo behavior" happens if and only if the universal Turing machine wants to halt and output "yes". (An example of this type of tweaking is the reduction used to prove that the language R_3 is not recursive in Homework 8, problem 1.)