CMSC 451: Lecture EXF
Final Review

The Final Exam will be **Monday, Dec 15, 10:30am-12:30pm in CSIC 2117** (our normal classroom). The exam will be closed-book and closed-notes, but you are allowed 2-sheets of notes, front and back.

Overview: We have covered a lot of material this semester! Our overall goal in this course is to provide a understanding of the tools used in designing and analyzing efficient algorithms. The intent has been to investigate basic algorithm design paradigms: depth-first search, greedy algorithms, dynamic programming, etc. And to consider how these techniques can be applied on a number of well-defined computational problems. We also studied network flow and the related problems of cuts and circulations. We studied the class of NP-complete problems, which are widely believed not solvable in polynomial time, but no one knows for sure. Finally we discussed studied methods for developing approximation algorithms for optimization problems, with a special focus on NP-hard problems.

How to use this information: The algorithms we have studied this semester are not always directly applicable in practice. Real world problems are often much messier, and have many domain-specific constraints. Nonetheless, I hope that the methods that we have discussed this semester will provide you with a strong foundation for tackling such problems. There are a number of important lessons to take away from this course.

Develop a clean mathematical model: Most real-world problems are messy. An important first step in solving any problem is to produce a simple and clean mathematical formulation. For example, this might involve describing the problem as an optimization problem on graphs, sets, or strings. If you cannot clearly describe what your algorithm is supposed to do, it is very difficult to know when you have succeeded.

Create good rough designs: Before jumping in and starting coding, it is important to begin with a good rough design. If your rough design is based on a bad paradigm (e.g. exhaustive enumeration, when depth-first search could have been applied) then no amount of additional tuning and refining will save this bad design.

Prove your algorithm correct: Many times you come up with an idea that seems promising, only to find out later (after a lot of coding and testing) that it does not work. Prove that your algorithm is correct before coding. Writing proofs is not always easy, but it may save you a few weeks of wasted programming time. If you cannot see why it is correct, chances are that it is not correct at all.

Can it be improved? Once you have a solution, try to come up with a better one: faster, simpler, or more general. Is there some reason why a better algorithm does not exist? (That is, can you establish a lower bound?) If your solution is exponential time, then maybe your problem is NP-hard.

Prototype to generate better designs: We have attempted to analyze algorithms from an asymptotic perspective, which hides many of details of the running time (e.g. constant factors), but give a general perspective for separating good designs from bad ones.

After you have isolated the good designs, then it is time to start prototyping and doing empirical tests to establish the real constant factors. A good profiling tool can tell you which subroutines are taking the most time, and those are the ones you should work on improving.

Still too slow? If your problem has an unacceptably high execution time, you might consider an approximation algorithm. The world is full of heuristics, both good and bad. You should develop a good heuristic, and if possible, prove a ratio bound for your algorithm. If you cannot prove a ratio bound, run many experiments to see how good the actual performance is.

Cheat Sheets: Recall that the exam is closed-book, closed-notes, but you are allowed a two sheets of notes. Use this sheet to write down things like:

Important formulas: Such as summations such as $\sum i$, $\sum c^i$, and facts about asymptotics.

Definitions: Definitions of concepts and important algorithmic elements (e.g., strong component, edge types in DFS, flows and cuts, terms from NP-completeness, reductsion, approximation ratio). It's also good to write out problem definitions (e.g., what is the k-center problem?) as well as the running times of algorithms we studied. Also, know what a vertex cover, dominating set, Hamiltonian path, coloring, etc. are.

Basic facts and lemmas: Such as, "The sum of vertex degrees in an undirected graph is equal to twice the number of edges," or "An independent set in a graph is a clique in the complement graph." I would not worry about the details in the proofs, but it is good to have an intuitive feeling for the "why" behind each lemma and theorem.

Short descriptions of algorithms: For each algorithm it is good to make note about the input format and any restrictions (e.g., a weighted digraph with nonnegative vertex weights), the output, and the asymptotic running time.

By the way, if you forgot a definition, formula, or general fact, you can ask about it on the exam.

Wow, that is a lot of material! You should think of your cheat sheets as a study aid in which to summarize the most relevant information. Once you have prepared it, try to "internalize" the material. How much of it can you reproduce directly from memory. Students have told me that the best cheat sheet is one that you prepared so carefully that you never needed to refer to it during the exam.

Material for the final exam:

Analysis Review: Summations, asymptotics, recurrences. I may ask you to solve a summation, put functions in asymptotic order, or give a (short) proof by induction.

Graph Traversals and Applications: We reviewed basic facts about graphs and digraphs, their representations (adjacency matrix and adjacency list). We presented depth-first search (DFS) and discussed a number of applications, including topological sorting and strong components.

Greedy Algorithms: Greedy algorithms are based on the notion of building up a structure by a series of choices based on a certain *greedy order*. The main issue with greedy algorithms is proving their correctness. We saw two common approaches to proving optimality.

We studied a number of greedy scheduling algorithms (Interval Scheduling, Interval Partitioning, and Scheduling to Minimize Lateness) and we also presented greedy algorithms for two NP-complete problems, Gonzalez's algorithm for the k-Center Problem and the greedy algorithm for Set Cover.

Dynamic Programming: We explored how to design efficient algorithms for the use of Dynamic Programming (DP). We also discussed implementation methods, such as memoization and bottom-up table construction. We discussed a number of examples of DP, including

- Weighted Interval Scheduling
- Longest Common Subsequence
- Chain Matrix Multiplication
- Floyd-Warshall Algorithm for all-pairs shortest paths

Network Flow: Know the many concepts that we introduced, such as s-t network, flow, residual network, augmenting path, and cuts. Also remember the basic Ford-Fulkerson algorith. Know the Max-Flow/Min-Cut Theorem, and how to apply it. Expect to answer questions that involve reducing problems to network flow (or variations, such as the circulation problem).

NP-completeness:

Basic concepts: Decision problems, polynomial time, the class P, the class NP, polynomial time reductions.

NP-completeness reductions: You are responsible for knowing the following reductions.

- 3-coloring to Clique-Cover
- 3SAT to Independent Set (IS)
- IS to Vertex Cover (VC) and IS to Clique
- VC to Dominating Set (DS)

We also discussed the Subset-Sum problem when discussing approximations, but you are not responsible for the reduction.

NP-complete reductions can be challenging. If you cannot see how to solve the problem, here are some suggestions for maximizing partial credit. All NP-complete proofs have a very specific form. Explain that you know the template, and try to fill in as many aspects as possible. Suppose that you want to prove that some problem B is NP-complete.

- B $\in$ NP. This almost always easy, so don't blow it. This basically involves specifying the certificate (that is, the guess), and then giving a polynomial-time algorithm to verify the problem specifications.
- For some known NP-complete problem A, prove that A $\leq_P$ B. This means that you want to find a polynomial time function f that maps an instance of A to an instance of B. (Make sure to get the direction correct!)

- Show that your reduction is correct, by showing that $x \in$ A if and only if $f(x) \in$ B. First, assuming that you have a solution to x, show how to map this to a solution for $f(x)$. Next, assuming that you have a solution to $f(x)$, show how to map this to a solution for x.

If you are stuck coming up with a function f, think about what you would like f to do. Explain which elements of problem A will likely map to which elements of problem B. Remember that you are trying to translate the elements of one problem into the common elements of the other problem.

Approximation: We discussed the concept of approximations and approximation ratios. We presented the following:

- A factor-2 approximation for Vertex Cover
- Two approximations for the metric TSP problem: A factor-2 approximation based on the twice-around tour of the MST and Christofides factor-(3/2) approximation based on combining this with a minimum-weight matching.
- A polynomial-time approximation scheme (PTAS) for the Subset Sum problem. Recall that this involved starting with an exact DP solution, and repeatedly pruning the resulting lists of sums, keeping only values that are sufficiently distinct in value.