1 Format

You will have 3 hours to complete the exam. The exam will have true/false questions, multiple choice, example/counterexample problems, run-this-algorithm problems, and problem set style present-and-prove problems.

Whenever you are asked to give an algorithm for a problem, you are expected to analyze its correctness and running time. More efficient correct solutions receive more credit. Less efficient but correct solutions would receive partial credit. We understand and expect that proofs of correctness will be more concise on exams than they would be on homework.

The exam is long, so be sure to work efficiently. Do not be discouraged, however, if you do not finish.

2 Studying Strategies

tl;dr DO PRACTICE PROBLEMS! CLRS and Kleinberg-Tardos are great resources, and solutions may or may not be extremely easy to find online. But definitely try problems without looking at the solutions as much as possible.

There are two general areas the final will test: the understanding of the algorithms and the application of the algorithms/paradigms covered in class.

When studying the details of specific algorithms:

• Understand the main idea behind the algorithm, the runtime, and the space complexity for any algorithm that we’ve covered.
  – Trying to boil down what the algorithm is doing to a 1-3 sentence summary can help to ensure you understand the key point of the algorithm.
  – For space and time complexity, you want to be able to identify what the most expensive operations are or what the recursion leading to the complexity is.

• Focus on the big ideas of the analysis of the algorithms above all the details.
  – A good example is Linear Programming; you’ll probably get more utility from understanding what kind of problem it solves (maximizing/minimizing a linear equation based on linear constraints) and what kind of algorithm solves the problem (Simplex Method) than to memorize how to actually solve it.

• Try thinking about variations of the algorithms. For example, if you have a major step in the algorithm, try thinking about what would happen if tweaked that step. Alternatively, if the algorithm uses one data structure, what would happen if you replaced it with a different one?

When practicing solving algorithmic problems:

• Practice applying algorithms from class.
One thing that’s helpful here is to also solve problems where you don’t know what tool you’re supposed to be using, so that you can practice choosing them.

Practice in an exam-like setting. On the exam you won’t have collaborators, your notes, or the answer keys, and you’ll be constrained for time. Try to do problems with as few resources as possible, and avoid the temptation to look at the answer after a few minutes!

• Try to break down the process into pieces (e.g. the three steps of dynamic programming, the steps for showing NP-completeness).

• Understand what the key characteristics are that make a tool work well.
  - For example, dynamic programming tends to work better when you can break your problem down into slightly smaller subproblems (say one size smaller), whereas divide and conquer works when you can break your problem down into significantly smaller problems (say half the size).

• When you’re working on the problems, a lot of times the most difficult piece is not the details of analysis/proof, but the initial choices you make.
  - For dynamic programming and divide-and-conquer, the first step is to choose a subproblem. However, a lot of times the difficulty in getting to the answer is not in the analysis after this point, but in choosing the subproblem itself (so if you find yourself getting stuck, try using a different subproblem).
  - For dealing with P/\textbf{NP}, when doing reductions you generally want to choose a problem that’s as close to the original as possible; choosing a subproblem that’s further away will probably force you to do a lot of extra work when you’re trying to prove the reduction.

3 Topics Covered

The following list is not comprehensive and is meant to serve as a starting template. Anything in lecture up to and including 4/19 or problem sets is fair game, with emphasis on the last 1/3 of the course. The lecture on 4/24 will not be tested.

Problems on material up to and including dynamic programming will be part of final review section 1. Problems on material starting at randomization (post Quiz 2) will be in final review section 2.

3.1 Math/Fundamentals

• Induction. If \( P(n) \) is a statement (“\( 2n \) is even”), \( P(1) \) is true, and \( \forall n, P(n) \rightarrow P(n + 1) \), then \( \forall n, P(n) \) is true.

• Big-O Notation. \( o, O, \omega, \Omega, \Theta \) and identifying which of these relations hold for two functions.

• Recurrence Relations. Solve simple ones by finding a pattern and proving them with induction. More complicated recurrences can be solved using the Master Theorem (must memorize).

• Integer Multiplication. Perform 3 multiplications on \( n/2 \) digit numbers, and then do some additions.
• **Fast Powering.** Use repeated squaring to find an \( n \)th power in \( O(\log n) \) time.
• **Merge Sort.** Sort a list of \( n \) numbers in \( O(n \log n) \) time. Implement recursively or iteratively.

### 3.2 Graph Search

• **Representation.** Adjacency list versus adjacency matrix
• **Depth First Search (DFS).** Uses a stack to process nodes, Pre and post order numbers, labels for the edges (tree, forward, back, cross).
  *Important Applications.* Detecting cycles, topological sorting, finding strongly connected components (relate to 2SAT).
• **Breadth First Search (BFS).** Uses a queue to process nodes, can be used to find the shortest path when all edge weights are 1.
• **Dijkstra’s Algorithm.** Single source shortest path for non-negative edge weights. Uses a heap or priority queue to process nodes. Does not work when there are negative edge weights.
• **Heaps.** Binary heap implementation, operations: \textsc{deleteMin}, \textsc{insert}, \textsc{decreaseKey}, how they are used in Dijkstra’s algorithm.
• **Bellman-Ford Algorithm.** Single source shortest path for general edge weights, edge relaxation procedure (referred to as \textit{update} in the lecture notes), detecting negative cycles.
• **Floyd-Warshall Algorithm.** All pairs shortest paths via dynamic programming, detecting negative cycles.
• **Shortest Path in DAG.** Can be done in linear time via dynamic programming regardless of edge weights.

### 3.3 Minimum Spanning Trees, Union Find

• **Basic Properties.** Connected, acyclic, \( |E| = |V| - 1 \).
• **Cut Property.** The basis for why our MST algorithms are correct, allows us to greedily add edges to a sub-MST.
• **Prim’s Algorithm.** Implemented in a similar fashion as Dijkstra’s, builds MST from a single vertex.
• **Kruskal’s Algorithm.** Uses union-find data structure (see next subsection). Takes \( O(|E| \log |E|) \) time to sort but then takes \( O(|E| \log^* |E|) \) time for the union-find part.
• **Union Find.** A disjoint set data structure supports Union and Find. Using Union by Rank and Path Compression, we are able to achieve \( O(m \log^* n) \) for \( m \) operations on \( n \) objects.

### 3.4 Greedy

• **Main Idea.** At each step, make a locally optimal choice in hope of reaching the globally optimal solution. Remember that greedy algorithms can often seem correct, so it’s more important to prove the optimality of your algorithm.
• **Examples.** Horn Formula, Huffman Coding, Set Cover Approximation.
3.5 Divide and Conquer

• **Main Idea.** Divide the problem into smaller pieces, recursively solve those, and then combine them in the right way.

• **Examples.** Mergesort, integer multiplication, Strassen's matrix multiplication algorithm, median-finding.

3.6 Dynamic Programming

• **Main Idea.** Maintain a lookup table of correct solutions to sub-problems and build up this table in a particular order.

• **Examples.**
  - String Reconstruction: Tries to find where the first dictionary word ends by checking all possible breaking points.
  - Edit Distance. Tries all possibilities of insert, delete, change on letters that are different.
  - All Pairs Shortest Paths. Uses the idea that the shortest path to a node must have come via one of the neighboring nodes.

You should treat dynamic programming problems as having three parts. Your goal is to find a function $f$ which can be computed recursively so that evaluation of $f$ on a certain input gives the answer to the stated problem.

(a) Define $f$ in words (without mention of how to compute it recursively). You should clearly state how many parameters $f$ has, what those parameters represent, what $f$ evaluated on those parameters represents, and what parameters you should feed into $f$ to get the answer to the stated problem.

(b) Give a recurrence relation showing how to compute $f$ recursively.

(c) Give the running time and space for solving the original problem using computation of $f$ via memoization or bottom-up dynamic programming. If you need to use certain data structures to make computation of $f$ faster, you should say so. If there are multiple solutions to solve the stated dynamic programming problem, you should describe the most time-efficient one you know. If there are multiple solutions with the same asymptotic time complexity, you should describe the implementation that gives the best asymptotic space complexity.

See problem set 5 solutions for two examples of this (Floyd-Warshall and problem 2).

3.7 Randomized Algorithms

• **Main Idea.** We can sometimes achieve polynomial time randomized algorithms for hard (NP-complete) problems, or come up with simpler solutions than deterministic algorithms.

• **Examples.**
  - **Freivald’s Algorithm.** We verify that a matrix product $A \times B$ equals $C$ by choosing a random vector $x$ and checking $Cx = ABx$.
  - **QuickSort, QuickSelect.** These choose a pivot point and sort around that pivot point.
– **Skip Lists.** These provide a simpler solution to the predecessor problem than balanced binary search trees, and achieve expected $O(\log n)$ time.

– **Hashing.** We use hashing to solve the dynamic dictionary problem.

– **Karger’s algorithm.** This algorithm uses edge contraction to find the global min-cut.

– **Randomized 2SAT.** Using worst-case analysis on a random walk, we can find a satisfying solution if it exists with probability $1/c$ in $cn^2$ steps.

- **Probability.** Good things to remember: expectation, conditional expectation, linearity of expectation, Markov bound, random walks.

### 3.8 Max Flow

- **Main Idea.** Given a directed, weighted graph, find the maximum amount of flow that can go through the network from two specified vertices $s$ to $t$.

- **Ford-Fulkerson.** Using DFS on integral edge weights, this runs in $O(|E|f^*)$ time where $f^*$ is the amount of maximum flow. Using BFS yields $O(|V||E|^2)$ time.

- **Min-Cut/Max-Flow Duality.**

### 3.9 Linear Programming

- **Main Idea.** We have a polynomial-time black box (the simplex algorithm) to solve linear programs which consist of a linear objective function and linear constraints.

- **Applications.** Production scheduling, max flow.

- **Duality.** Linear programs can be dual, such as the max-flow/min-cut LPs or the LPs for zero-sum games.

### 3.10 NP-Completeness

- **Main Idea.** #CS 121. NP-complete problems are hard to solve, with no known polynomial time algorithm.

- **NP-completeness.** A yes/no problem is NP-complete if (1) it is in NP (has a polynomial-time verifier) and (2) is NP-hard (an NP-hard problem can be polynomial-time reduced to it).

- **Sample NP-complete problems.** VERTEXCOVER, 3SAT, INDEPENDENTSET, CLIQUE.

### 3.11 Approximation Algorithms

- **Main Idea.** Because NP-complete problems are hard, we can consider polynomial-time algorithms to solve NP-complete problems up to a constant factor of optimality.

- **Examples.** VERTEXCOVER 2-approx, max cut 2-approx, Euclidean Traveling Salesperson 1.5-approx.