1 Format

You will have 90 minutes 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.

The exam is difficult. In previous years, the class average has been around 60%.

2 Topics Covered

Disclaimer: I have not seen the exam, nor do I know what will be covered on it. This is not a comprehensive list of the topics that will or will not be on the exam. Please refer to your own notes when studying.

2.1 Math Fundamentals

- **Coin Flipping** and basic statistics.
- **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.
- **Recurrence Relations.** We can solve simple ones by hand (or perhaps by writing out a few terms). More complicated recurrences can be solved using the **Master Theorem**.
- You should be comfortable with the definitions for $O$, $o$, $\Omega$, $\omega$, and $\Theta$.

2.2 Data Structures

- Arrays, Linked Lists, Stacks, and Queues.
- Heaps and Priority Queues
- Disjoint Set, including the “union by rank” and “path compression” optimizations.

2.3 Algorithms

You should be able to run any of these algorithms by hand on the exam.

- **Mergesort.** Divide and Conquer, merging sorted sublists together.
- **Depth First Search.** Key idea is to use a stack to enumerate nodes. Assign preorder and postorder when pushing and popping nodes from the stack. Know what tree edges, forward edges, back edges, and cross edges are.
- **Topological Sort.** Useful for ordering nodes in a Directed Acyclic Graph
• **Strongly Connected Components.** Turn any graph into a DAG of SCCs.

• **Breadth First Search.** Key idea is to use a queue to store unprocessed nodes.

• **Dijkstra’s Algorithm.** Use DFS with a priority queue. Only works for positive edge weights.

• **Bellman Ford.** Run update rule $|V| - 1$ times. Works on arbitrary graphs. Bonus: finds negative cycles!

• **Prim’s Algorithm.** Build a tree from a single seed vertex. Uses the Cut Property to choose the smallest edge leaving the group.

• **Kruskal’s Algorithm.** Build a tree by sorting the edges, joining disjoint groups of vertices. Uses the Disjoint Set data structure.

• **Huffman’s Algorithm.** Build a tree that is used to determine codewords to compress strings.

### 2.4 Algorithm Strategies

• **Greedy.** Pick the best option at each step. e.g., Horn Formula Algorithm

• **Divide and Conquer.** Split the problem into smaller subproblems (often in half), solve the subproblems, and combine the results. e.g., Mergesort, Integer Multiplication, Strassen’s

• **Dynamic Programming.** Useful for when there are many overlapping subproblems. e.g., Fibonacci, Edit Distance, Floyd-Warshall (all pairs shortest paths)

### 3 Practice Problems

#### 3.1 2SAT

*From Problem Set 2*

The input to 2SAT is a logical expression of a specific form: it is the conjunction (AND) of a set of clauses, where each clause is the disjunction (OR) of two literals. (A literal is either a Boolean variable or the negation of a Boolean variable.) For example, the following expression is an instance of 2SAT:

$$(x_1 \lor x_2) \land (\overline{x_1} \lor \overline{x_3}) \land (x_1 \lor x_2) \land (x_4 \lor x_3) \land (x_4 \lor \overline{x_1})$$

A solution to an instance of a 2SAT formula is an assignment of the variables to the values T (true) and F (false) so that all the clauses are satisfied that is, there is at least one true literal in each clause. For example, the assignment $x_1 = T, x_2 = F, x_3 = F, x_4 = T$ satisfies the 2SAT formula above.

**Exercise.** Derive an algorithm that either finds a solution to a 2SAT formula, or returns that no solution exists.

*Hint: Reduce to an appropriate problem. It may help to consider the following directed graph, given a formula $I$ in 2SAT: the nodes of the graph are all the variables appearing in $I$ and their negations. For each clause $(\alpha \lor \beta)$ in $I$, we add a directed edge from $\overline{\alpha}$ to $\beta$ and a second directed edge from $\overline{\beta}$ to $\alpha$. How can this be interpreted?*
Solution.
Each edge is an implication. If \( \alpha \) is true, then \( \beta \) must be true for the 2SAT to hold.

In this graph, strongly connected components correspond to values that must all be selected (or not selected). If a strongly connected component contains both \( \alpha \) and \( \bar{\alpha} \), then the formula is not satisfiable, since it contains a contradiction.

We now group our graph by SCCs, forming a DAG. We remove implication sinks, setting values appropriately. When removing \( \alpha \), we also remove \( \bar{\alpha} \). This doesn’t affect our ability to fulfill our implication obligations, since by construction if \( \beta \leftarrow \bar{\alpha} \), then \( \alpha \leftarrow \beta \).

### 3.2 Minimum Spanning Trees

**Exercise.** Let \( G = (V, E) \) be a connected, undirected graph with distinct edge weights. Consider a cycle \( v_1, v_2, \ldots, v_k, v_1 \) in \( G \), and let \( e = (v_i, v_{i+1}) \) be the edge in the cycle with the largest edge weight. Prove that \( e \) is not in any minimum spanning tree \( T \) of \( G \).

**Solution.**
Proof by contradiction. Remove \( e \) from \( T \) and construct a lighter MST using other edges in the cycle.

### 3.3 Cutting Wood

*From MIT 6.006, Spring 2011*

Given a log of wood of length \( k \), Woody the woodcutter will cut it once, in any place you choose, for the price of \( k \) dollars. Suppose you have a log of length \( L \), marked to be cut in \( n \) different locations labeled 1, 2, \ldots, \( n \). For simplicity, let indices 0 and \( n + 1 \) denote the left and right endpoints of the original log of length \( L \). Let the distance of mark \( i \) from the left end of the log be \( d_i \), and assume that \( 0 = d_0 < d_1 < d_2 < \ldots < d_n < d_{n+1} = L \).

**Exercise.** Determine the sequence of cuts to the log that will (1) cut the log at all the marked places, and (2) minimize your total payment to Woody.

**Solution.**
DP with subproblem \( \text{Min}(i, j) \), the minimum cost of cutting a log with endpoints \( i, j \) at all its cut points.

### 3.4 Guitar Hero

*From MIT 6.046, Fall 2009*

You are training for the World Championship of Guitar Hero World Tour, whose first prize is a real guitar. You decide to use algorithms to find the optimal way to place your fingers on the keys of the guitar controller to maximize the ease by which you can play the 86 songs. Formally, a note is an element of \( [A; B; C; D; E] \) (representing the green, red, yellow, blue, and orange keys on the guitar). A chord is a
nonempty set of notes, that is, a nonempty subset of \([A; B; C; D; E]\). A song is a sequence of chords: \([c_1; c_2; \ldots; c_n]\). A pose is a function from \([1; 2; 3; 4]\) to \([A; B; C; D; E; \emptyset]\), that is, a mapping of each finger on your left hand (excluding thumb) either to a note or to the special value \(\emptyset\); meaning that the finger is not on a key. A fingering for a song \([c_1; c_2; \ldots; c_n]\) is a sequence of \(n\) poses \([p_1; p_2; \ldots; p_n]\) such that pose \(p_i\) places exactly one finger on each note in \(c_i\), for all \(1 \leq i \leq n\).

You have carefully defined a real number \(D[p; q]\) measuring the difficulty of transitioning your fingers from pose \(p\) to \(q\), for all poses \(p\) and \(q\). The difficulty of a fingering \([p_1; p_2; \ldots; p_n]\) is the sum \(\sum_{i=2}^{n} D[p_{i-1}, p_i]\).

**Exercise.** Give an \(O(n)\)-time algorithm that, given a song \([c_1; c_2; \ldots; c_n]\), finds a fingering of the song with minimum possible difficulty.

**Solution.**
Solution: DP with subproblem \(\text{Opt}(i, p_i)\): the minimum possible difficulty for the song \([c_1; \ldots; c_i]\) when the \(i\)-th pose is \(p_i\). This is a classic Markov Chain DP.