CSCI 270 Spring 2016 Midterm Review (Joe Bebel - bebel@usc.edu):

(1) Topics:
- Optimization vs decision problems
- Divide-and-conquer algorithms
- Lower Bounds on Comparison Sort
- Greedy algorithms
- Dynamic programming algorithms
- NP-completeness
- **Read the course outline to make sure you review each topic**

(2) You should know the **definitions/terms** that have been defined in this class. For example, definitions/terms really worth knowing the definition of:
   (a) A sorted list
   (b) An undirected graph, a directed graph, a weighted graph, a connected graph
   (c) A path in a graph
   (d) A cycle in a graph
   (e) A tree (as a kind of graph) and a binary tree
   (f) A spanning tree of a graph
   (g) A *minimum* spanning tree of a graph
   (h) A longest common subsequence of a string
   (i) Variables, literals, clauses in a boolean formula
   (j) A boolean formula in 3-CNF
   (k) A truth assignment
   (l) Boolean formula satisfied by a truth assignment
   (m) A directed Hamiltonian cycle in a graph
   (n) A k-clique in a graph
   (o) A k-vertex cover in a graph
   (p) Decision problem
   (q) Polynomial time algorithm
   (r) Polynomial time reduction

(3) We only cover a few **computational problems** in this class (about a dozen by my count) so you should be familiar with all of them. Remember a computational problem is *different* than an algorithm that solves it. A computational problem is also different than the definitions. For example, a 'shortest path' is not the same thing as 'The Shortest Path Problem'

   Important computational Problems to study:
   (a) The Sorting Problem
   (b) The Matrix Multiplication Problem
   (c) The 'McDonald's' Coin Counting Problem
   (d) The Minimum Spanning Tree Problem
   (e) The Length of the Longest Common Subsequence Problem
   (f) NP-complete Problems:
      (i) The 3-SAT Problem
      (ii) The k-Vertex Cover Problem
      (iii) The Directed Hamiltonian Cycle Problem
      (iv) The k-Clique Problem

(v) The Subset-sum Problem

(4) Finally we have **algorithms** to study:
   (a) Insertion Sort and Merge Sort algorithms
   (b) Strassen's algorithm
   (c) The greedy algorithm that solves the 'McDonald's' Coin Counting Problem
   (d) The greedy algorithm that solves the Minimum Spanning Tree Problem
   (e) The dynamic programming algorithm that solves the Length of the Longest Common Subsequence Problem
   (f) Polynomial time reductions:
      (i) 3SAT to Subset-sum
      (ii) k-Clique to k-Vertex Cover
      (iii) 3SAT to k-Clique
      (iv) 3SAT to Directed Hamiltonian Cycle

   Make sure you know how each algorithm works (including the reductions). Each one is based on some important principle or observation.

(5) Consider the difference between a computational problem and an algorithm that solves it. For example, the Sorting Problem defines a required input/output, but there are many algorithms that solve the Sorting Problem. Insertion Sort and Merge Sort have the same input/output behavior, but are different algorithms

(6) Lower bounds on comparision sort: we discussed in class how to show that every *comparision sort* - that is, a sorting algorithm that figures out how to sort the list by comparing pairs of elements of the list - must run in worst-case time $\Omega(n \log n)$. Insertion sort and Merge sort are comparison sorts.

   The basic idea was that there are $n!$ different ways a list of length n can be ordered. An informal description follows: each time a comparison sort makes a comparison, we can think of the outcome of the comparison as a branch of a binary tree which describes the algorithm's behavior when taking that branch. Each leaf represents a permutation of the input list's elements. Since this tree has at least $n!$ leaves, it must have depth $\log(n!)$ We used Stirling's approximation to prove , which can be summarized as: $\log(n!) = \Theta(n \log n)$

   This does not rule out that some other kind of sorting algorithm can run faster than $\Theta(n \log n)$. But it must do something more clever with the list than just comparing elements.

(7) Divide-and-conquer, Greedy, Dynamic Programming. They are all *recursive* methods based on a principle of *assuming you already have an algorithm for smaller problems*:
   (a) Divide-and-conquer: Split the problem into smaller chunks. Solve each smaller chunk. Combine the solutions to each chunk into a solution for the original problem.
   (b) Greedy: For optimization problems: assuming you have an optimal solution to a smaller subproblem, make 'the best choice available' to get an optimal solution to a slightly bigger subproblem.
   (c) Dynamic programming: For optimization problems: give an optimal solution recursively in terms of optimal solutions to subproblems, where the total number of distinct subproblems is small.

These are just informal descriptions of these methods. You'll likely find slightly different descriptions elsewhere.

(8) The matrix multiplication problem is:

The Matrix Multiplication Problem:
INPUT: Two $n \times n$ matrices A, B
OUTPUT: The $n \times n$ matrix AB

For simplicity we assume that $n = 2^m$ for some integer m.

First we observe that the 'traditional' matrix multiplication algorithm takes $O(n^3)$ arithmetic operations.

The basic idea of Strassen's algorithm is that we can use a divide-and-conquer strategy. We can rewrite the matrices:

$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ and $N = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$

where A, B, C, D, E, F, G, H are all $\frac{n}{2} \times \frac{n}{2}$ matrices. It can be shown the product MN matrix is the same as the 2×2 matrix product:

$MN = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix}$

So for example, the upper-left $\frac{n}{2} \times \frac{n}{2}$ submatrix of MN is equal to $AE + BG$

The key insight of Strassen is that the four $\frac{n}{2} \times \frac{n}{2}$ submatrices of MN can be computed using only 7 matrix multiplies instead of the traditional 8. By calling Strassen's algorithm recursively, we get a multiplication algorithm that runs in time:

$T(n) = 7T(n/2) + 18n^2$

Which if you solve with the standard substitution method (I won't solve it here) you get $T(n) = O(n^{\log_2 7})$ which is better than $O(n^3)$

(9) Now we move on to optimization problems. Optimization problems are usually characterized by some of the following words: 'smallest', 'shortest', 'fewest', 'largest', 'longest', 'minimize', 'maximize'

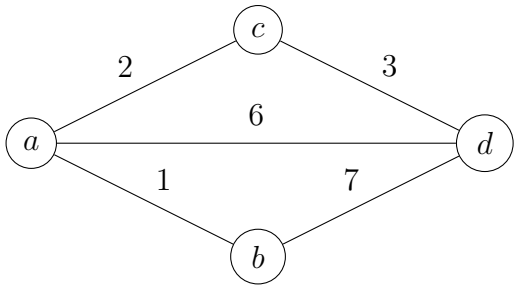
Optimization problems are concerned with finding the 'best' possible answer among some collection of potential answers.

(10) Recall the 'McDonald's' Coin Counting Problem. Informally, you're supposed to find the way to make n cents with as few coins as possible, where you have a supply of 1 cent, 5 cent, 10 cent, and 25 cent coins. As practice, try writing the formal statement of this problem in the 'box':

The 'McDonald's' Coin Counting Problem:
INPUT:
OUTPUT:

It turns out that a greedy algorithm works in this case, because $\{1, 5, 10, 25\}$ is a 'nice' set of coins. The greedy algorithm won't work for all sets of coins, though. The proof that it works in this case is long and can be found in the homework.

(11) The second example of a greedy algorithm is finding a minimum spanning tree. Recall a spanning tree of a graph G is a subset of edges of G that includes every vertex and forms a connected tree. A minimum spanning tree of a weighted graph is a spanning tree that has the smallest sum of edge weights. Example:



We adopt a greedy strategy. We choose a vertex (doesn't matter which) and greedily choose the smallest edge adjacent to that vertex. Then we look at all remaining edges that connect a vertex in the tree to a vertex not in the tree, and we choose the smallest. Repeat until all vertices are in the tree.

No matter what vertex we start from, we get that $(a, b), (a, c), (c, d)$ is the minimum spanning tree.

(12)

The Longest Common Subsequence Problem:
INPUT: Two strings x, y
OUTPUT: The longest common subsequence of x, y

A string s is a subsequence of t if s can be obtained by deleting characters/elements of t without changing the relative order of the remaining characters/elements. Every pair of strings has at least one common subsequence (the empty string), so the Longest Common Subsequence problem is to find the *longest* such common subsequence (an optimization problem.

Trying every possible subsequence would take exponential time. But by writing the longest common subsequence recursively, we can solve this problem using dynamic programming in polynomial time:

We write each string as $x = x_1 \dots x_n$ and $y = x_1 \dots y_m$. We will denote the longest common subsequence of $x_1 \dots x_i$ and $y_1 \dots y_j$ as $OPT(i, j)$. We know that the base case $OPT(1, 1)$ is either the empty string (if $x_1 \neq y_1$, or else $OPT(1, 1) = x_1$ if $x_1 = y_1$.

Then we can write recursively:

$$OPT(i,j) = \begin{cases} OPT(i-1, j-1)x_i, & \text{if } x_i = x_j \\ \text{longer}(OPT(i-1, j), OPT(i, j-1)) & \text{if } x_i \neq x_j \end{cases}$$

This recursion can be computed in time $O(1)$ assuming the values of OPT it references have already been computed. Since there are $O(nm)$ different values of OPT that need to be computed in order to compute the desired value $OPT(n, m)$, this algorithm runs in time $O(nm)$.

(13) Decision problems. Unlike optimization problems, decision problems are not concerned with the 'best' answer. A decision problem is only asked to output 1 or 0. (which can be thought of as 'yes/no', or 'true/false', so on)

(14) Polynomial time reductions are algorithms that have some specific properties. Pay close attention to the requirements for an algorithm to be a polynomial time reduction from A to B:

- A, B are both decision problems
- INPUT: An instance x of the problem A

- OUTPUT: An instance y of the problem B
- If the answer to x (as an instance of problem A) is 1, then the answer to y (as an instance of problem B) must be 1
- If the answer to x is 0, then the answer to y must be 0
- Must run in polynomial time

An instance of a computational problem is just some input to that computational problem. So if we consider the decision problems Even and Odd:

The Even Problem:	The Odd Problem:
INPUT: An integer n	INPUT: An integer m
OUTPUT: 1 if n is even	OUTPUT: 1 if m is odd
0 otherwise	0 otherwise

Then an instance of the Even problem is an integer n, and an instance of the Odd problem is an integer m. Consider the following algorithm:

```
EvenToOdd(Integer n)
    return n+1
```

Note that EvenToOdd satisfies the following properties:

- The Even Problem and The Odd Problem are both decision problems
- This algorithm's input is an instance of The Even Problem (an integer n)
- This algorithm's output is an instance of The Odd Problem (and integer m)
- If the answer to Even on input n is 1 (that is, n is even), then the answer to Odd on input $n + 1$ is 1 (that is $n + 1$ is odd)
- If the answer to Even on input n is 0 (that is, n is not even), then the answer to Odd on input $n + 1$ is 0 (that is $n + 1$ is not odd)
- Adding 1 to an integer can be done in polynomial time

Therefore, The Even Problem is polynomial time reducible to The Odd Problem.

(15) NP-completeness. Note that NP-completeness is a property of *only* decision problems. It doesn't make sense to call an optimization problem NP-complete.

We've given 5 NP-complete decision problems in this class. Three are about graphs, one is about boolean logic and one is about sets of numbers. We've also given some algorithms that convert, or *reduce*, one problem into another, in polynomial time. Some of the reductions are pretty simple and others are more complex. I suggest reviewing each of the problems and reductions that we've given in class.

(16) The key thing about NP-completeness is that a decision problem A is NP-complete if *every other* NP-complete problem is polynomial time reducible to A. (Make sure you understand the order! We're reducing problems *to* A) We've given some reductions between problems, but we know there's more. So for example, since k-Clique is NP-complete, we know there's a polynomial time reduction from Directed Hamiltonian Cycle to k-Clique, even though we haven't written one out explicitly.