Study Guide for CS 401 / MCS 401 Final Exam — Summer 2007

1. **Mathematics for analysis of algorithms**

   Logarithmic, polynomial, and exponential functions.

   The factorial function and its logarithm, Stirling's approximation to $n!$. Be sure that you know and can use Stirling's approximation $n! \approx (n/e)^n \text{ sqrt}(2\pi n)$.

   Binomial coefficients, probability of k successes in n independent trials.

   Rate of growth of functions ($O(g)$, $\Omega(g)$, $\Theta(g)$, $o(g)$, $\omega(g)$); L'Hopital's rule; relative growth rates of logarithmic, polynomial, and exponential functions.

   Determining change in the running time of an algorithm caused by change in the input size, given $\Theta(T(n))$.

   Finding the running time of an algorithm, given pseudo-code. (For a recursive algorithm, you will obtain a recurrence.) Worst case vs expected case.

2. **Important Algorithms**

   Fast exponentiation: You should understand how this algorithm improves upon the standard method of exponentiation, and be able to illustrate how it may be used to compute a^n for specific n.

   Straight insertion sort, *merge* and *mergesort*, *max-heapify* and *heapsort*, *partition* and *quicksort*: You should be familiar with the performance of each algorithm, and with its advantages and disadvantages (stability, extra space required, ability to take advantage of order present. You should be able to illustrate the operation of max-heapify and heapsort, or partition and quicksort. (I will not ask you to illustrate the other sorting algorithms.)

   Selecting the k^{th} smallest in linear expected time. This algorithm also uses partition, and is similar to quicksort. I will not ask you to illustrate its operation (on midterm).

   Polynomial multiplication in $\Theta(n \lg(n)$ time using DFT and DFT^{-1}. You should know that this algorithm runs in $\Theta(n \lg(n))$ time, versus $\Theta(n^2)$ time if we compute the product directly from the definition and $\Theta(n^{1.59})$ time using a simpler divide-and-conquer algorithm. Beyond this, the only thing I might ask you about is how to evaluate a polynomial of degree bound n ($n = 2^k$) at the n^{th} roots of unity by evaluating two polynomials of degree $n/2$ at the $n/2^{\text{th}}$ roots of unity, plus linear divide/combine time.

   All pairs shortest paths: How the algorithm works, optimal substructure.

   Prim's and Kruskal's minimal spanning tree algorithms: You should be familiar with the definition and properties of an MST (as developed in a handout), and be able to illustrate the construction of an MST by Prim's algorithm.

   Depth-first search (DFS) in (di)graphs: You should understand how this algorithm is based on a stack, and be able to illustrate its operation on a digraph. You should understand the discover and finish times of a vertex, and the relationship of these times to the classification of edges. You should also understand the relation between edge types and strongly connected components.

   Topological sort (application of DFS): You should understand what it means to topologically sort a digraph, and be able illustrate the topological sort algorithm based on depth-first search.

   String matching by finite automata: You should be able to construct an FA for matching a short pattern in a text. You should understand the advantages of the FA algorithm.

   To limit the number items you will need to study, I will <u>not</u> ask you any details of the following important algorithms, which we went over in class: Euclid's algorithm for finding the gcd, finding the k^{th} smallest in guaranteed linear time, optimal order for matrix multiplication, longest common subsequence, typesetting neatly, breadth-first search (BFS) in (di)graphs, the algorithm for finding the strongly connected components in a digraph (but you should know the definition of strongly connected components,and their relation to cycles and edge types)), the Knuth-Morris-Pratt string-matching algorithm (to be covered if time permits).

3. **Algorithm design techniques**

   Divide-and-conquer.

   Obtaining recurrences for the number of basic operations, or running time (worst / expected cases).

   Solving the recurrences that arise in divide-and-conquer algorithm (Master Theorem, substitution).

When not to use divide-and-conquer: overlapping subproblems.

You should know the Master Theorem and how to apply it. You should be able to solve simple recurrences (such as those on the quizzes) directly, without using the Master Theorem. I will not ask you about the extended version of the Master Theorem developed in class.

Dynamic programming.

Choosing the subproblems, and the order of solving them.

Finding the optimal substructure, i.e., the equation expressing the (optimal) solution to a larger problem in terms of the (optimal) solutions to smaller problems.

Designing a (usually non-recursive) algorithm based on optimal substructure.

Given the description of a problem, you should (with some hints) be able to define the subproblems and derive the optimal substructure.

Greedy method.

Examples where the greedy method works (Huffman encoding, Prim's and Kruskal's MST algorithms). Why the method fails for most problems.

Algorithms based on searching in graphs.

4. Lower Bounds

Lower bounds for comparison sorting algorithms; decision trees.