Study Guide 1:

Asymptotic Notation, Recurrences, Divide and Conquer, Dynamic Programming

General Study Notes:

- As a practice, you should always be able to give the asymptotic runtime notation for an algorithm that you are given, or that we have studied.

Asymptotic Notation – What you should know:

- Formal definitions of Θ, O, Ω, o, ω
- How to convert an equation to asymptotic notation, and how to compare 2 equations using asymptotic notation (e.g., Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?)
- You do not need to memorize all the mathematical formulas given in Section 3.2 or Appendix A, although you should understand how to use these properties if they are given to you explicitly (e.g, given the formula for solving a geometric series, you should be able to apply that formula to simplify an equation).

Recurrences – What you should know:

- How to convert a statement of a divide-and-conquer algorithm to a recurrence (e.g., given a statement of the Merge Sort algorithm, write down the corresponding recurrence)
- How to apply the Master Method to solve recurrences
- How to apply iteration (i.e., recursion tree method) to solve recurrences.

Divide-and-conquer approaches:

- Break the problem into *independent* subproblems that are similar to the original problem, but smaller in size
- Solve the subproblems recursively
- Combine the subproblem solutions to generate a solution to the original problem

Example divide-and-conquer problems and their best running times that we studied:

- Merge sort [$\Theta(n \lg n)$]
- Maximum subarray problem [$\Theta(n \lg n)$]
- Matrix multiplication using Strassen's method [$\Theta(n^{\lg 7})$]

Strassen's Method – What you should know:

- Runtime requirements for "ordinary" matrix multiplication (i.e., $\Theta(n^3)$).

- How matrix multiplication can be stated as a divide-and-conquer algorithm (i.e, using submatrices, as described on page 76).
- The basic Strassen's algorithm, and a general understanding of how Strassen's method managed to eliminate one multiplication from the "ordinary" computation (although you don't have to memorize the various formulas for the S_i and P_j matrices).
- Reasons why Strassen's method would, or would not, be used in practice.

Know characteristics of problems for which *Dynamic Programming* is appropriate:

- Polynomial number of subproblems
- Subproblems share subproblems
- Desire is to find optimal solution amongst many possible solutions (i.e, objective is *optimization*)

Know *Dynamic Programming* approach:

- Characterize the structure of an optimal solution
- Recursively define the value of an optimal solution
- Compute the value of an optimal solution in a bottom-up fashion
- Construct an optimal solution from computed information

Know how to build a subproblem graph for a dynamic programming solution, and how to use its size to determine the runtime of the algorithm.

Know example problems to which we applied Dynamic Programming:

- Rod cutting [$\Theta(n^2)$]
- Matrix-chain multiplication [$\Theta(n^3)$]
- Longest common subsequence (LCS) [$\Theta(mn)$]
- Optimal binary search trees [$\Theta(n^3)$]
- All-pairs shortest path, based on matrix multiplication [$\Theta(n^3 \lg n)$]
- All-pairs shortest path, Floyd-Warshall algorithm [$\Theta(n^3)$]
- All example problems from homework exercises and extra problems covered in class

For all these problems, you should know:

- How the problem exhibits optimal substructure,
- The definition of the recursive solution,
- How to apply the recursive solution to compute the value of the answer from the bottom up, and

- How to extract the optimal solution from the computed information.
- How to use the algorithms (in a practical sense) to solve a specific problem (e.g., compute optimal parenthesization of matrix chain for a given problem, or optimal LCS for a given problem, etc.)

Based on these examples, you should also be able to provide the above for a new problem to which you want to apply dynamic programming.

Memoization is a variant of dynamic programming, which:

- Solves problem recursively, from the top down (instead of bottom up)
- Stores subproblem solutions as they are solved.
- When a new subproblem is generated, a table lookup is done first to see if the answer has already been calculated.