Study Guide 2:

Greedy Algorithms, Amortized Analysis

Exam #2 will cover material from Chapters 16, 17, 21, 23, 26, as well as homeworks 6-9. Sections of the text *not* included on the exam are sections 16.4, 16.5, 21.2, 21.4, 26.3, 26.4, and 26.5.

Greedy Algorithms

In general, a greedy algorithm will not always result in an optimal solution. Therefore, you need to prove that the problem for which you have developed a greedy solution exhibits the required characteristics for which a greedy algorithm *will* produce an optimal result.

Characteristics of problems for which *Greedy Algorithms* are appropriate:

- Problem exhibits *greedy-choice property*: a globally optimal solution can be arrived at by making a locally optimal (i.e., greedy) choice.
- Problem exhibits *optimal substructure*.
- Desire is to find optimal solution amongst many possible solutions (i.e, objective is *optimization*)

Comparing Greedy Algorithms with Dynamic Programming:

- If Greedy Algorithm is appropriate, so is Dynamic Programming (except that DP will be less efficient). But if Dynamic Programming is appropriate, we can't say anything about whether Greedy Algorithms are appropriate (i.e., without further understanding of the problem)
- In Dynamic Programming, a choice is made at each step, and that choice is dependent upon solutions to subproblems. In Greedy Algorithms, we make whatever choice seems best at the moment, and then solve the subproblem after the choice is made.
- Dynamic Programming is bottom-up, while Greedy Algorithms are top-down.
- Both Dynamic Programming and Greedy Algorithms operate on problems that exhibit *optimal substructure.*
- Greedy Algorithms are similar to Dynamic Programming, except that we only make the optimal choice at each step, and the result of that optimal choice is a single subproblem that might be non-empty.
- Problems for which a greedy approach *will* work can often appear very similar to problems for which a greedy approach *will not* work. An example is the *0-1 knapsack* problem, compared to the *fractional knapsack* problem. Greedy algorithms work for the *fractional knapsack* problem, but not the *0-1 knapsack* problem. Dynamic programming *can* solve the *0-1 knapsack* problem.

The sequence of steps for designing a greedy algorithm is as follows:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate that, having made the greedy choice, what remains in a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Example problems to which we applied Greedy Algorithms:

- Activity selection [$\Theta(n \lg n)$ for sort + $\Theta(n)$ for activity selection ➔ $\Theta(n \lg n)$]
- Huffman codes [$\Theta(n \lg n)$, assuming priority queue implementation as a binary heap]
- Kruskal's algorithm for Minimum Spanning Trees [$O(E \lg V)$, assuming disjoint-set data structure is disjoint-set-forest with union by rank and path compression]
- Prim's algorithm for Minimum Spanning Trees [$O(E \lg V)$, assuming use of binary heap for priority queue. Can achieve $O(E + V \lg V)$ if implement priority queue as a Fibonacci heap.]
- Generic algorithm for Minimum Spanning Trees (you should understand the theory underlying the generic MST algorithm)
- Edmonds-Karp algorithm for maximum flow [$O(VE^2)$]

In leading up to the greedy Edmonds-Karp algorithm for maximum flow, we also studied:

- Ford-Fulkerson method for maximum flow [$O(E \mid f^*\mid)$, where f^* is the maximum flow] (which isn't really a greedy algorithm)

For all these problems, you should know:

- The greedy algorithm
- How the problem exhibits optimal substructure
- How the greedy choice property applies
- How to use the algorithms (in a practical sense) to solve a specific problem (e.g., compute optimal Huffman code for a given alphabet with frequencies defined for each character)

Data Structures for Disjoint Sets – you should know:

- How to apply MAKE-SET, UNION, and FIND-SET operations using disjoint-set forests with the *union by rank* heuristic and the *path compression* heuristic – that is, know how this implementation works for a practical application.

- The running time of disjoint-set forests with union by rank and path compression – i.e., $O(m\ \alpha(n))$, where m is the total number of operations and n is the number of MAKE-SET operations, and $\alpha(n) = O(\lg n)$. (You do not need to understand how this running time is derived.)

You *will not* be required:

- To generate proofs from scratch for a new problem that are as complex as, for example, the proof of correctness of the Huffman algorithm. (However, you should *understand* the proofs we discussed in class, such as the proof of the Huffman algorithm, generic minimum spanning trees, etc.)
- To know the derivation of the disjoint set data structure runtime analysis

Amortized Analysis

You should know following:

- The 3 main approaches for amortized analysis – aggregate method, accounting method, and potential method
- The analysis of 2 primary applications studied in the context of these methods – stack operations and the binary counter
- The use of amortized analysis in the design of the solution to dynamic tables
- More generally, how amortized analysis can help design efficient data structures and operations

You are also expected to be able to work amortized analysis problems like those in your homework, and those we worked in class.