Exam study sheet for CS6783

The second test will cover both the topics that were covered by the first test, and additional topics subject of assignment 3 and lectures after that (you can expect to see long questions on the topics of assignment 3, as well as previous topics, and yes/no or "example" questions on the subsequent topics). To study, refer to the notes (when available), as well as Kleinberg-Tardos book ("KT") and Cormen-Leiserson-Rivest-Stein ("CLRS") books, or any other books on algorithms.

Here is the list of topics we have covered. We talked about algorithm design paradigms and specific problems with algorithms that solve them.

For each paradigm, please be able to do the following

1. Give an example of a problem and an algorithm from this paradigm (e.g., a greedy algorithm for min. spanning. tree, divide-and-conquer algorithm for sorting)

2. Know, roughly, the type of problems solvable within it (e.g., dynamic programming and greedy algorithms can sometimes approximate and sometimes solve restricted versions of optimization problems)

3. Know the basic steps of designing an algorithm within this paradigm (for greedy: sort, then go through elements without retracing to previously considered elements; for dynamic programming create an array, etc.)

For the applications, please be able to name and describe algorithms solving a specific problem. Give the time complexity of the algorithms in O-notation, and mention, if relevant, data structures being used. For example, Kruskal's algorithm for min. spanning tree uses Union-Find data structure, and with it can run in time $O(m \log n)$.

Make sure you know how to solve every problem on assignments 1, 2 and 3, and the first test.

List of topics

1. **Stable matching (stable marriage) problem**
   Know the statement of the problem, that the input is $2n$ preference lists of length n each. Know what is a matching, what is a stable/unstable matching). Be able to give examples. Know the algorithm to solve it (Gale-Shapley algorithm). Know some applications (e.g., medical interns assignment) and extensions; which data structures are needed to implement it.

2. **Background**
   We are considering asymptotic worst-case performance of algorithms. Know O-notation. Know some lower bounds such as sorting, and the difference between an upper bound in O-notation (e.g., bubble sort and merge sort solve the same problem in different time), and lower bounds in Ω-notation (no comparison-based sorting faster than $\Omega(n \log n)$. When O and Ω coincide, use $\Theta(f(n))$ (e.g., sorting is $\Theta(n \log n)$).

3. **Basic data structures**
   Arrays, lists, priority queue based on heap data structure, and union-find data structure.

4. **Graph algorithms**
   Directed unweighted graphs: DFS, BFS, topological sort, strongly connected components, testing bipartiteness.
   Undirected weighted graphs: Minimal spanning trees: greedy algorithms by Kruskal and Prim.
   Directed weighted graphs: Dijkstra's single-source shortest-path (greedy, no negative weights). Bellman-Ford single-source shortest-path algorithm (allows negative weights, can detect a negative cycle, solves systems of difference constraints). Floyd-Warshall all-pairs shortest path (dynamic programming).

5. **Greedy algorithms**
   Know the main idea (selecting a locally optimal solution at each step leads to a global optimum). Know how to design a greedy algorithm and prove its correctness. Know examples of both exact and approximation greedy algorithms. List of problems we considered: interval scheduling, scheduling with deadlines and profits, 1/2 approximating Knapsack, Dijkstra's algorithm, Huffman code, Kruskal's and Prim's algorithms for min. spanning. tree. Use notes for his topic.

6. **String matching**
   Know Rabin-Karp algorithm and Knuth-Morris-Pratt. Which of them would you use for matching multiple patterns? Which is related to finite automata? Read this topic in CLRS (or online resources).

7. **Dynamic programming**
   Know the steps of constructing a dynamic programming algorithm solution. Be able to define an array (as in "A(i,j) stores the value of... ", give a recurrence (as in "A(i,j) = ...", not pseudocode!). Know when it gives a polynomial time solution and when it does not. Applications: interval scheduling, all-pairs shortest path, scheduling with deadlines, profits, and durations, longest common/increasing subsequence, sequence alignment (from bioinformatics lectures), etc. Give an example of a situation when a dynamic programming algorithm does not run in polynomial time. What would be another approach in this case? (backtracking). Use notes for this topic.

8. **Bellman-Ford**
   Know the algorithm, understand why it works, running time, etc. Use the standard, CLRS, version that can detect negative cycles. Know how to use it to solve systems of difference equations. Read this topic in CLRS, not KT.

9. **Network flows.**
   Know what is a flow, how to compute a flow using Ford-Fulkerson, what is an augmenting path, what is a residual network. Give example where Ford-Fulkerson runs in exponential time; know about the example where it does not terminate (but it works on integers). Know how to make it polynomial-time (choose paths by breadth first search); you don't need to know the proof. Know the max flow min cut theorem, and applications to disjoint paths, bipartite matching, project selection, etc. Be able to solve problems similar to the assignment 3. Use CLSR, KT and notes (especially KT for applications; notes or CLRS for general definitions, theorems and algorithms).

10. **Divide-and-conquer**
    Know the paradigm (split the problem in several pieces, solve each piece recursively, combine results). Be able to give examples (E.g., mergesort, multiplying integers). Know the closed form for recurrences $T(n) = aT(n/b) + n^c$, at least for $b = 2$; look it up under "master theorem" (just know the three cases).

11. **FFT**
    Know what Discrete Fourier Transform is, and that polynomial coefficients form a convolution of the original coefficient vectors. Know how to do polynomial multiplication using FFT. Note: please read it in CLRS or elsewhere, rather than KT.

12. **Limits of tractability**
    Know what NP-complete and NP-hard is (NP-hard: any language in NP reduces to it. Can be optimization problem. NP-complete: a language in NP which is NP-hard. Has to be a language. E.g. MaxIndependentSet (G) is NP-hard; IndependentSet(G,k) is NP-complete).

    Methods of dealing with intractability: restrict the problem (e.g., Ind.Set on trees); consider approximations (e.g., 1/2 approx for knapsack), parameterize (e.g., k-Vertex Cover), use randomness (no proof that it actually helps, but can't disprove yet either; e.g. polynomial identity testing). Use local search heuristics (simulated annealing, gradient descent; know that local minima for MaxCut are 1/2 approx of global minimum). Read KT on these topics.