Study Guide 3:

Multithreaded Algorithms, Linear Programming, NP Completeness

---

Multithreaded Algorithms

(Covers Chapter 27 of the text, plus HW 10)

You should know how to do the following:

- How to create a computation dag, and analyze it in terms of work and span.
- How to write parallel code using **parallel, spawn,** and **sync**, without race conditions.
- How to analyze parallel code in terms of work, span, and parallelism.
- How to determine whether code has a race condition.
- How to use the formula for greedy scheduling to analyze algorithms.

We studied parallel algorithms for multithreaded matrix multiplication and multithreaded merge sort; you should understand these algorithms and their analysis.

You are also expected to be able to work multithreaded problems like those in your homework.

You are not responsible for the following:

- The proof of equation 27.9 (page 803).
- The concept of "coarsening" (page 787).

Linear Programming

(Covers Chapter 29 of the text, plus HW 11)

You should know the definitions/meanings and formulations of the following, relative to linear programming:

- Linear program (i.e., what constitutes a linear program)
- Linear equalities, linear inequalities, and linear constraints
- Feasible region, convexity of feasible region, simplex
- Infeasible solution, feasible solution, objective value
- Standard and slack forms (and how to convert into these forms)

Other knowledge you should have:

- How to formulate problems as linear programs
- How simplex algorithm works (including concepts such as basic variables, non-basic variables, pivoting, entering variable, leaving variable, basic solution, objective value, termination, cycling, and unbounded solution)
- Runtime of simplex algorithm in worst case (i.e., exponential)

- Names of alternative linear programming techniques (i.e., ellipsoid algorithm and interior-point methods), and their runtimes (i.e., polynomial; no more detail needed than this)
- Lemma 29.4 (i.e., that slack form is uniquely determined by basic variables)
- Lemma 29.5 (i.e., max number of iterations of the simplex algorithm)
- How to formulate the dual
- How to generate an initial basic feasible solution (i.e., using auxiliary linear program), or determining that the linear program is infeasible
- The significance of linear-programming duality (i.e., how is the concept of duality used to prove optimality of the simplex algorithm, from a high-level perspective)
- The definition of the integer linear programming problem (pg. 850), and the fact that it has no known polynomial-time algorithm.

You do not have to be able to recreate the following:

- Detailed proofs of optimality of simplex algorithm (except for knowing the high-level ideas behind the concept of duality, as noted above)
- Detailed proofs of correctness of simplex algorithm (but you should understand it generally, at a high level)

NP-Completeness

(Covers Chapter 34 of the text, plus HW 12)

This guide does not exhaustively list the facts you should know about NP-Completeness. But, the following facts are examples of the types of information you should be very comfortable with:

- P is the class of languages that are solvable in polynomial time.
- NP is the class of languages that are verifiable in polynomial time. An instance, x, of a problem is verified in polynomial time using a certificate y, where $|y| = O(|x|^c)$, for a constant c – that is, the certificate is polynomial in the size of the input.
- $P \subseteq NP$, since any problem solvable in polynomial time can be verified in polynomial time.
- We do not know the relationships between the complexity classes P and NP.
- If $A \leq_p B$, and $B \in P$, then $A \in P$.
- NP-completeness applies to decision problems – that is, problems in which the answer is either "yes" or "no".
- If $A \leq_p B$, and $A \in NPC$, then $B \in NPC$.

You should know how to prove a problem, L, is NP-complete:

1. Show $L \in NP$.
2. Show L is NP-hard. This can be done in two primary ways:
    a. Prove that $L' \leq_p L$, for every $L' \in NP$.
    b. Or, Prove that a known NP-complete language, A, is polynomial-time reducible to L – i.e., $A \leq_p L$. This involves 4 steps:
        i. Select a known NP-complete language A.
        ii. Describe an algorithm that computes a function *f* mapping every instance of A to an instance $f(x)$ of L.
        iii. Prove that *f* satisfies $x \in A$ if and only if $f(x) \in L$, for all *x*.
        iv. Prove that the algorithm computing *f* runs in polynomial time.

You should understand the definitions of the NP-complete problems we have studied:

- Circuit satisfiability – CIRCUIT-SAT = {<C> | C is a satisfiable Boolean combinational circuit}
- Formula satisfiability – SAT = {<ϕ> | ϕ is a satisfiable Boolean formula}
- 3-CNF satisfiability – 3-CNF-SAT = {<ϕ> | ϕ is a satisfiable Boolean formula in 3-conjunctive normal form}
- Clique problem – CLIQUE = {<G,k> | G is a graph with a clique of size k}
- Vertex-cover problem – VERTEX-COVER = {<G,k> | G is a graph with a vertex cover of size k}
- Hamiltonian cycle problem – HAM-CYCLE = {<G> | G is a graph with a Hamiltonian cycle}
- Traveling salesman problem – TSP = {<G, c, k> | G is a complete graph with cost function c, which has a tour of cost at most k}
- Subset sum problem – SUBSET-SUM = {<S, t> | there exists a subset of S whose elements sum to t}

You should be able to work problems like those on Homework #12, and show reductions to new problems to prove NP-Completeness.

You will not be tested over the details of abstract problems, encodings, and formal languages, as covered on pages 1054-1060. You also will not have to re-create the NP-Completeness proof of CIRCUIT-SAT.