DS 320 Algorithms for Data Science **Lecture #25: Final Exam Prep**
Spring 2023 Exam: 11:59pm May 3 – 3pm May 10

The rules of the exam are as follows:

- You are free to consult class notes, the textbook, homework solutions, and any other resources posted on the course website or in the "Resources" section of the course Piazza page. **You may not consult with other students or the internet.** You may also ask Prof. Goldner questions during office hours or virtually.
- Post clarifying questions as a **private note** on Piazza, or email them to Prof. Goldner. She will respond within 24 hours. Direct all questions to her and not to TAs.
- Solutions must be typeset in LaTeX.
- When asked for a runtime in big-Oh notation, give the tightest and simplest possible bound you can. If your algorithm is $O(n)$ and also $O(n^2)$, just say it is $O(n)$. If it is $O(2n)$, it is also $O(n)$, so omit the factor of 2. And so on.
- Submit the exam via gradescope by **3:00pm on Wednesday, May 10**. Late exams will not be accepted.
- Do not discuss the exam with your classmates until after the due date, even if you have finished.
- Whenever an algorithm is given, it must be accompanied with justification and a runtime.

Topics to be Covered

- Asymptotic runtime ($O/\Omega/\Theta$)
- Reading and writing pseudocode
- Sorting
- Induction
- Loop Invariants
- Data structures: stacks, queues, graphs
- Graph search
  - Algorithms: DFS, BFS (unweighted shortest path)
- Greedy algorithms
  - Algorithms: Dijkstra (non-negative weighted shortest path), caching, interval scheduling, scheduling to minimize lateness, Huffman codes

  - Techniques: Greedy stays ahead proof, Greedy exchange proof

- Divide & Conquer
  - Algorithms: Mergesort, Closest pair of points, Integer Multiplication, Matrix Multiplication
  - Techniques: Prove the recurrence via induction, solve the runtime recurrence (recurrence trees, master theorem)
- Dynamic Programming
  - Algorithms: Bellman-Ford (all weighted shortest path), weighted interval scheduling, segmented least squares, knapsack
  - Techniques: Find the DP recurrence, proof via 7 part solution
- Linear Programming
  - Techniques: formulating problems as LPs, taking duals, using properties from duality

Covered in much less depth:

- NP-Completeness
  - Concepts: What is P, NP, NP-Hard, NP-Complete?
  - Techniques: How do we relate hard problems to each other?
- Remaining lectures:
  - Zero Sum Games
  - Multiplicative Weight Update
  - Randomized Algorithms
  - Online Algorithms

Types of Questions Asked

a. Standard homework questions (give an algorithm, prove its correctness, analyze its runtime).

b. True/False about properties of algorithms, theory, or proofs.

(a) And justify why.

(b) And correct the statement if wrong.

c. Short answer, e.g., of counter examples, algorithms, proofs, and runtimes.

Final Exam Review

For **all** algorithms, *always* give:

(1) a clear enough description that someone could code it up without knowing any specific language (even if it just an english description, it must be that clear to understand!),

(2) a justification of why it gives the guarantees it does, and

(3) an analysis of its running time.

Reiterating, "design and analyze" means given a word problem, introduce necessary notation, design an algorithm to solve the problem, analyze runtime, and analyze accuracy (along with any other problem specific requirements of the algorithm or solution).

Intuition and Reminders

Greedy:

- Appropriate when the next best choice ("myopic") leads you to optimality.
- "Best" has to be by some metric—in interval scheduling, we scheduled by earliest finish time, *not* by shortest job time, so picking which metric to use as "best" is important.
- You should be thinking: What if we just take the next available thing that meets X criteria?
- Pitfalls to look out for: when you can't just look at each piece individually/successively, when you need to be able to "look ahead" somehow.

Divide and Conquer:

- Appropriate when you have subproblems that can be solved independently.
- Usually for a D&C problem, brute force (e.g., check all pairs) should already be efficient (polynomial) for the problem; you just want to speed up over that.
- Runtime recurrences $T(n) = aT(n/b) + f(n)$ should remind you that you're splitting the problem in a subproblems of size n/b, solving them (further recursively), and then combining the solutions, and at this level taking computing time $f(n)$.
- You should be thinking: If each subproblem was already solved, this would be easy. I just wish I could break it down smaller...
- Pitfalls to look out for: merging the solutions shouldn't be too difficult, or the subproblems probably aren't independent.

Dynamic Programming:

- Appropriate when you have recursive subproblems that are not independent, and when there's a clever order that allows us to build up the answers to avoid recursive computation.

- You should be thinking: I can't figure out whether this thing is in my optimal solution or not!! Wait, so there are multiple cases to maximize over... either this thing is in my optimal solution, or it's not (could be more than two cases)—leads to your recurrence!
- Pitfalls to look out for: Is your recurrence (1) well-defined (base-cases), (2) built in the right order (memo-table), and (3) correct (read it to yourself in English)?

Linear Programming:

- The *fractional* relaxation can be solved in polynomial-time—but not if we constrain to only integral solutions! (Because we can express NP-Hard problems this way.)
- The objective function only improves with fractional solutions.
- The dual problem of a maximization gives an *upper bound*, and its *objective function* is equal when they are both optimal. It is *not* an equivalent problem, however. It is a very different problem, asking a different question, with different types of solutions. The values are just equal when optimal (and only when optimal).
- Weak duality (upper bound), strong duality (equality), and complementary slackness (primal constraint vs. dual variable and vice versa) are the parts of duality theory we talked about.

Homework 7 Part 2: Dual Greed

Re: Maximum Spanning Tree (Kruskal) and Shortest Path (Dijsktra).

a. Consider a connected undirected graph $G = (V, E)$ in which each edge e has a weight w_e. For a subset $F \subseteq E$, let $\kappa(F)$ denote the number of connected components in the subgraph (V, F).

Prove that the spanning trees of G correspond to the integer solutions to the following linear program with the same objective function value (with decision variables $\{x_e\}_{e \in E}$):

$$\max \sum_{e \in E} w_e x_e$$

subject to

$$\begin{aligned} \sum_{e \in F} x_e &\leq |V| - \kappa(F) && \forall\, F \subseteq E \\ \sum_{e \in E} x_e &= |V| - 1 \\ x_e &\geq 0 && \forall\, e \in E. \end{aligned}$$

(While this linear program has a huge number of constraints, we are using it purely for the analysis of Kruskal's algorithm.)

b. What is the dual of this linear program?

c. What are the complementary slackness conditions?

e. Now consider the problem of computing a shortest path from s to t in a directed graph $G = (V, E)$ with a nonnegative cost c_e on each edge $e \in E$. Prove that every simple s-t path of G corresponds to an integer solution of the following linear program with the same objective function value:[1]

$$\min \sum_{e \in E} c_e x_e$$

subject to

$$\sum_{e \in \delta^+(S)} x_e \geq 1 \qquad \forall\, S \subseteq V \text{ with } s \in S, t \notin S$$

$$x_e \geq 0 \qquad \forall\, e \in E.$$

(Again, this huge linear program is for analysis only.)

f. What is the dual of this linear program?

g. What are the complementary slackness conditions?

[1] Recall that $\delta^+(S)$ denotes the edges sticking out of S.

Review Problems

Part 1

You're working at an investment company that asks you: given the opening price of the stock for n consecutive days in the past, days days $i = 1, 2, \ldots, n$, given the opening price of the stock $p(i)$ for each day i, on which day i should the company have bought and which later day j should they have sold shares in order to maximize their profits? If there was no way to make money during the n days, you should report this instead.

For example, suppose $n = 3, p(1) = 9, p(2) = 1, p(3) = 5$. Then you should return "buy on 2, sell on 3" (buying on day 2 and selling on day 3 means they would have made \$4 per share, the maximum possible for that period).

Clearly, there's a simple algorithm that takes time $O(n^2)$: try all possible pairs of buy/sell days and see which makes them the most money. Your investment friends were hoping for something a little better. Show how to find the correct numbers i and j in time $O(n \log n)$.

Part 2

Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.

a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

b. Give a set of coin denominations (so *not* penny, nickel, dime, quarter) for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

Part 3

A palindrome is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are: (1) all strings of length 1, (2) civic, (3) racecar, and (4) aibohphobia (fear of palindromes). Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input "character," your algorithm should return "carac." What is the running time of your algorithm?