DS 320 Algorithms for Data Science **Lecture #25**
Spring 2022 Prof. Kira Goldner

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 $T(n) = a\,T(n/b) + f(n)$ 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)?

Review Problems

Part 1

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 2

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 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?