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.

Algorithm: Starting with no coins, take the largest denomination coin given that it is worth at most n minus how much the coins you've taken so far are worth.

Proof by Greedy Exchange. Let $A = (c_1, \ldots, c_k)$ be the coins generated by the greedy algorithm and let $O = (o_1, \ldots, o_m)$ be the coins generated by some other algorithm. Let q_A, q_O be the number of quarters generated by each, and dimes d_A, d_O, and so on. We can write

$$n = 25q_A + 10d_A + 5n_A + p_A \quad \text{and} \quad n = 25q_O + 10d_O + 5n_O + p_O.$$

By definition of the greedy algorithm, $q_A \geq q_O$ and $n - 25q_A < 25$. If $q_O < q_A$ and $n - 25q_O \geq 25$, we exchange multiple smaller coins for one larger quarter, improving its solution. We do this until $q_O = q_A$ (or it already does), then we set aside quarters. We then continue process on dimes, then nickels, until $O = A$, and we have only reduced the number of coins, hence A is optimal. □

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.

Let the coin denominations be 1, 5, 7. Suppose you want to make change for $n = 10$. The greedy algorithm will use one 7 coin and three 1 coins, for a total of four coins. It is optimal to use two 5 coins.

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 $O(n \log n)$.

Whenever we see that brute-force is $O(n^2)$ and a speed-up is $O(n \log n)$, we should think of divide and conquer.

A natural approach would be to consider the first $n/2$ days and the final $n/2$ days separately, solving the problem recursively on each of these two sets, and then figure out how to get an overall solution from this in $O(n)$ time. This would give us $T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$, and hence $O(n \log n)$.

Our main observation is that there are three cases when we split the days into two sets:

- We buy then sell within the first $n/2$ days—this is the optimal solution on the days $1, ..., n/2$.
- We buy then sell within the last $n/2$ days—this is the optimal solution on the days $n/2 + 1, \ldots, n$.
- We buy in the first $n/2$ days and sell in the last $n/2$ days: then the day we buy i is the *minimum* price among days $1, ..., n/2$ and the day we sell j is the *maximum* among days $n/2 + 1, \ldots, n$.

The first two alternatives are computed in time $T(n/2)$, each by recursion, and the third alternative is computed by finding the minimum in the first half and the maximum in the second half, which takes time $O(n)$. Thus the running time $T(n)$ satisfies

$$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n),$$

as desired.

This is actually not the best running time achievable for this problem. In fact, one can find the optimal pair of days in $O(n)$ time using dynamic programming—do you see how? (But, this question was definitely set up to get you to use D&C. Either answer would be correct.)

Algorithm 1 investing($p(1), \ldots, p(n)$).

```
Input: Prices $p(i)$ for days $i = 1, \ldots, n$.
if $n = 1$ then
    return (Null, Null)
else
    Let Buy1, Sell1 = investing($p(1), \ldots, p(n/2)$) and Buy2, Sell2 = investing($p(n/2+1), \ldots, p(n)$)
    Let Buy3 = argmin$\{p(1), \ldots, p(n/2)\}$ and Sell3 = argmax$\{p(n/2+1), \ldots, p(n)\}$
    if $p(\text{Sell3}) - p(\text{Buy3}) < 0$, then (Buy3, Sell3) = (Null, Null)
    return (Buy, Sell) $\in$ argmax{p(Sell1) - p(Buy1), p(Sell2) - p(Buy2), p(Sell3) - p(Buy3)}
end if
```

Claim 1. For any natural number n days, given prices $p(1), \ldots, p(n)$, the above algorithm returns the optimal day Buy and day Sell to maximize profits $p(Sell) - p(Buy)$.

Proof by strong induction on n.

Base Case: $k = 1$. When there is only 1 day, we cannot both buy and sell, so we return Null for the days to buy and sell on—do not trade.

Inductive Hypothesis: Assume the algorithm correctly finds the best Buy and Sell days in order if there are some, and otherwise returns (Null, Null) on $k < n$ days for some n.

Inductive Case: Given n days of prices, the algorithm considers 3 cases: when we buy and sell in the first $n/2$ days, when we buy and sell in the second $n/2$ days, and when we buy in the first $n/2$ days and sell in the second $n/2$ days. In the third instance, the revenue will be maximized by choosing the minimum price from the first $n/2$ days and the maximum from the second $n/2$ days, as the algorithm does. The first instance requires the algorithm's solution the first $n/2$ days, which is correct by the inductive hypothesis, and the second instance the algorithm's solution on the second $n/2$ days, again correct by the inductive hypothesis. We then take the maximum of these these profits and return the days that give this. If none of these three options give profits, then we return Null—no days should be bought/sold on—which is the same as the solutions on the first and second $n/2$ days. This implies that the algorithm is correct. □

Runtime Analysis: Let $T(n)$ denote the algorithm's worst-case runtime on two lists of size n. There are two recursive subproblems of size $n/2$ and $O(n)$ computational steps outside the recursive call (finding min/max). Hence, we have the recurrence: $T(n) = 2T(n/2) + O(n)$. This can be solved to obtain $T(n) = O(n \log n)$: each layer of recursion has $O(n)$ work, and there are $\log n$ layers.

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?

Subproblem: Let $\textsc{opt}(i, j)$ be the length of the longest palindrome of the input string from character i through character j.

Recurrence: $\textsc{opt}(i, j) = \begin{cases} \textsc{opt}(i+1, j-1) + 2 & \text{if } i = j \\ \max\left\{\textsc{opt}(i+1, j), \textsc{opt}(i, j-1)\right\} & \text{otherwise.} \end{cases}$

Proof of Recurrence: Consider any subsequence from i to j—either the first character is equal to the last, or it is not. If they are, the longest palindrome subsequence from i to j contains these characters, so we count them (+2), remove them from the ends, and then count the longest palindrome subsequence on the remaining $i + 1$ to $j - 1$. If not, then the longest palindrome subsequence on i to j is either the longest palindrome subsequence on i to $j-1$ or the longest palindrome subsequence on $i+1$ to j, as we know the ends are not the same.

Base Cases: $\textsc{opt}(i, i) = 0$ and $\textsc{opt}(i, i + 1) = 1$ for all i—all sequences of length 0 count 0 and all sequence of length 1 are length 1 palindromes.

Solution to Original Problem: $\textsc{opt}(1,n)$

Algorithm: See below.

Runtime: The algorithm fills a table of size $O(n^2)$, doing $O(1)$ table lookups to fill each entry. Base cases and returns are constant. The total runtime is therefore $O(n^2)$.

Computing a Solution: See below, return computePalindromeSol(String, 1, n, memo).

Algorithm 2 longestPalindromeSubsequence($String$).

```
Input: A string String of length $n$.
Memo[ ][ ] = new int[$n$][$n$]
for $\ell$ from 0 to $n-1$ do
    for $i$ from 1 to $n-\ell$ do
        $j = i + \ell$
        if $\ell = 0$ then
            Memo[$i$][$i$] = 0
        else if $\ell = 1$ then
            Memo[$i$][$i+1$] = 1
        else if String[$i$] = String[$j$] then
            Memo[$i$][$j$] = Memo[$i+1$][$j-1$] + 2
        else
            Memo[$i$][$j$] = max{Memo[$i+1$][$j$], Memo[$i$][$j-1$]}
        end if
    end for
end for
return Memo[1][$n$]
```

Algorithm 3 computePalindromeSol(String, i, j, memo).

```
Input: String, indices $i \leq j$, and filled out memo table.
if $i == j$ then
    return Null
else if $j == i+1$ then
    return String[$i$]
else
    if String[$i$] = String[$j$] then
        return String[$i$] + computePalindromeSol(String, $i+1$, $j-1$, memo) + String[$i$]
    else if Memo[$i+1$][$j$] > Memo[$i$][$j-1$] then
        return computePalindromeSol(String, $i+1$, $j$, memo)
    else
        return computePalindromeSol(String, $i$, $j-1$, memo)
    end if
end if
```