DS 320 Algorithms for Data Science
Spring 2025

Lecture #25: Final Exam Prep
Prof. Kira Goldner and Jeffrey Considine

The rules of the exam are as follows:

- When asked for a runtime in big-O 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.
- Please be sure to write your name at the top of the page. You may wish to do so on the top of each page just in case.
- You should read all the problems as soon as possible; problems are not necessarily in order of difficulty.
- Please solve ALL the problems. Keep your answers clear and concise.
- Please clearly mark all the answers to all your questions **and make sure that they are where they are expected to be**, and keep answers separated from scratch work. Scratch work will not be graded.
- Please write legibly. If we cannot read it, we cannot grade it.
- Absolutely no devices. If you need to use the bathroom during the exam, please leave it with the instructor.

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.

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).

How long would a brute force examination of all pairs of days take?

A brute force examination of all pairs of days would take $O(n^2)$ time. Specifically, $\binom{n}{2}$ choices if you require the stock to be held at least one day. Buying and selling the same day would give \$0 profit which only is helpful if all daily returns are negative.

Solve it now with divide and conquer in $O(n \log n)$ time or better.

Subproblem:

$\textsc{opt}(i, j)$ will be the best profit from buying and selling within days i through j.

Base case:

If $i \geq j$, then $\textsc{opt}(i, j) = 0$. For the $i = j$ case, the stock was bought and sold on the same day at the same price, so the profit is zero. For the $i > j$ case, the range is invalid and treated as zero.

Recurrence:

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 $n/2+1, \ldots, n$.

Let $\text{mid}(i,j) = \left\lfloor \frac{i+j}{2} \right\rfloor$. Then

$$\textsc{opt}(i,j) = \max \begin{cases} \textsc{opt}\,(i, \text{mid}(i,j)) \\ \textsc{opt}\,(\text{mid}(i,j)+1, j) \\ \max_{\text{mid}(i,j)+1 \le k \le j} p(k) - \min_{i \le k < \text{mid}(i,j)} p(k) \end{cases}$$

Proof of correctness:

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. □

Running time:

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.

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(Sell3) $-$ p(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

A puzzle for you: this problem can actually be solved in linear time.

Part 2

Youve been studying non-stop for finals, and youve gotten it down to a science. Whenever you drink coffee, you reset to h hours until you pass out. (You can never go for more than h hours without drinking coffee even if you load up in advance.) After scouring websites, you find a list of times when coffee is available around campus: $t_1, t_2, ..., t_n$. You have N hours to go to make it through the semester, and h full hours left until you pass out.

a. Give an efficient method by which you can determine at what times to get coffee in order to minimize the number of times you drink coffee without passing out.

> **Algorithm:** Initialize your latest (zeroeth) break at time B = 0. Given coffee breaks available at times $t_1 < t_2 < \ldots < t_n$ and let $t_{n+1} = N$, add a break at the latest time t_i such that $t_i - B \leq h$-the latest time possible such that you go at most h hours without coffee. Set $B = t_i$ and repeat until N hours have passed.

b. Prove that your strategy yields an optimal solution.

> We prove this by algorithms optimality by a Greedy Exchange proof.
>
> **Proof.** Let $A = a_1, \ldots, a_k$ be the breaks picked by our greedy algorithm (from now until end of term) and let $O = o_1, \ldots, o_m$ be the breaks picked by an optimal solution. We will compare A and O by how far from now it is. Consider the soonest break i at which O and A differ. If $a_i > o_i$ theres no harm in exchanging o_i with ai in Os solution, since we know that a_i is within h hours of o_{i-1} because $o_{i-1} = a_{i-1}$ and A is a feasible solution. Because this only brings o_i closer to o_{i+1}, they are still within h hours as well. It cannot be that $a_i < o_i$, since the greedy algorithm picks the latest break within h hours of the previous break, and the previous breaks are the same. Repeat this process until $O = A$it does not increase the number of breaks. If anything, it decreases the number of breaks in the event that somehow we increase o_i to $o_i' > o_{i+1}$, in which case we may delete any breaks weve passed.

c. Give its running time.

> Our algorithm scans through the list of times exactly once, and so has runtime $O(n)$

Part 3

Pascal's Triangle is a way to visualization the coefficients taking $x+1$ or $x+y$ to different powers.

$$\begin{array}{lcl|ccccccc} (x+y)^0 & = & 1 & & & & 1 & & \\ (x+y)^1 & = & x+y & & & 1 & & 1 & \\ (x+y)^2 & = & x^2+2xy+y^2 & & 1 & & 2 & & 1 \\ (x+y)^3 & = & x^3+3x^2y+3xy^2+y^3 & 1 & & 3 & & 3 & & 1 \end{array}$$

There is an analytical formula formula for each coefficient. For $(x+y)^n$, the coefficient of $x^k y^{n-k}$ is $\binom{n}{k}$ for $0 \le k \le n$. As a reminder, $\binom{n}{k=\frac{n!}{k!(n-k)!}}$.

Design a really fast algorithm using dynamic programming to generate the first n rows of Pascal's Triangle.

Subproblem:

$\text{coef}(n,k)$ is the coefficient of $x^k y^{n-k}$ in $(x+y)^n$ for $0 \le k \le n$.
(Yes, we know $\text{coef}(n,k) = \binom{n}{k}$.)

Recurrence:

$$\text{coef}(n,k) = \begin{cases} 1 & \text{if k=0} \\ \text{coef}(n-1,k-1) + \text{coef}(n-1,k) & \text{if } 1 \le k \le n \\ 0 & \text{otherwise} \end{cases}$$

Base cases:

$\text{coef}(0,0) = 1$

Proof of Recurrence:

There are a couple obvious ways to prove this recurrence. We could prove $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ from the factorial-based formula.

Or we could reason inductively on n and using the coefficients from $(x+y)^{n-1}$ and consider what happens when multiplying that by $(x+y)$ again. With that approach, observe we can get $x^k y^{n-k}$ terms from either multiplying the $x^{k-1}y^{n-k}$ terms of $(x+y)^{n-1}$ by x or multiplying the $x^k y^{n-k-1}$ terms of $(x+y)^{n-1}$ by y. So the x and y coefficients in $x+y$ are just one, then we add those previous coefficients together.

Solution to the Original Problem:

For row r from 0 to n, the rth row of Pascal's Triangle is $\text{coef}(r, 0), \ldots, \text{coef}(r, r)$.

Running time:

There are $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$ entries to compute, and each entry can be computed in constant time.

Running time lower bound:

Since there are $\Theta(n^2)$ entries to output, a minimum of $\Omega(n^2)$ time is required.