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?

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

Subproblem:

Base case:

Recurrence:

Proof of correctness:

Running time:

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.

b. Prove that your strategy yields an optimal solution.

c. Give its running time.

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 $0 \leq k \leq n$. As a reminder, $\left(_{k=\frac{n}{k!(n-k)!}}^{n_{n!}}\right)$.

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

Subproblem:

Recurrence:

Base cases:

Proof of Recurrence:

Solution to the Original Problem:

Running time:

Running time lower bound: