Final Report & Presentation Instructions

COMP 221 — Suhas Arehalli

Spring 2025

Overview

The goal of the final report is to provide an opportunity for you to do more than just analyze algorithms we discuss together. Rather, this is a place for you to demonstrate that you can quickly understand and analyze an *unfamiliar* algorithm using the tools, techniques, and vocabulary we've developed. After all, in practice, you're likely be researching existing algorithmic solutions to domain-specific problems you run into rather than trying to design one from scratch (or recall the perfect algorithm from this class). My contention is that what we've done in this class — learn tools for reasoning about correctness and surveying common algorithmic design skills — will help you accomplish this task. This assignment is where you get to (hopefully) make that connection!

At a high level, there are 3 things you'll spend your time doing:

1. **Formalizing your problem statement.** What are the expected inputs and outputs? What does correctness for an algorithm look like — does your solution need to satisfy some properties? Does something need to be optimal, or is there just some bound we need to be better than? Do we need to be correct *all of the time*, or is a probabilistic solution adequate? Some of these notions of correctness we won't, or haven't yet, explored in class, so this may involve extending your understanding!

2. **Analyzing an existing solution.** You should not construct a solution from scratch! What are canonical solutions to this kind of problem? How do they work? Do they use design principles we've seen before? How much better are they than a *naive* or *brute force* solution? Why does this algorithm work?

3. **Teaching the Algorithm.** You take my job for a bit! Using the formal language of algorithms we've developed, you will be expected to be able to communicate your understanding of the algorithm to an audience with an undergraduate algorithms background (as you might be expected to do in an industry or research setting!). This will take the form of the two deliverables for this assignment: A short (2–4 page) **report** that should act as a primer on the problem, where it shows up in practice, and your chosen algorithm solution, as well as a final **presentation** where you'll be expected to give a short (tentatively 8–10 minutes, though this will depend on the final number of groups) presentation that introduces the algorithm to your classmates.

The rest of this document will spell out the requirements of the project more formally.

1 Getting Started

1.1 Team Composition & Guidelines

You will be expected to work in teams of 2, with very few exceptions based on the parity of a section's enrollment. Each member of each team will be expected to contribute to most, if not all, parts of the project, and will be expected to be responsible for **all** parts of the project at the time of submission. That is, it is fine to feel more confident about some parts of the project than others when working on the project, but you will be responsible for understanding and being capable of explaining all parts of your report and presentation when you submit/present your group's work.

1.2 Choosing a Topic

While there are few explicit restrictions on how to find a particular algorithm or problem to tackle, here are a couple of (strong) recommendations for how to go about that task.

1. **Be mindful of how much your algorithm/problem goes beyond the undergraduate algorithms curriculum!** Many algorithms used in practice utilize concepts and ideas we don't have the time to properly develop in this course. Spending time to get comfortable with *some* of these extensions is encouraged, but be mindful of *how many* new things you'll need to learn before the required pieces of this assignment are doable! You will have approximately 5 weeks total to complete this assignment while keeping up with other class material!

2. **Choose well-studied problems/algorithms.** The more obscure or specific your problem of interest is, the harder it will be to find resources for understanding the runtime/correctness/formalization of the algorithm! This also goes for algorithmic techniques that have no or few formal guarantees — common in fields like machine learning! Your target algorithm should be something that *could* fit into an undergraduate or early graduate algorithms course that we didn't cover.

3. **Find problems that you're interested in.** It's always easiest to learn things when the applications of that thing are related to your interests or goals! There are plenty of algorithms that would satisfy this project's requirements related to fields that may be of interest: Computer Graphics, Computational Biology, Cryptography, Machine Learning, and many more!

4. **Don't cover algorithms/problems we've covered!** In general (though there are some exceptions), don't attempt to solve a problem we've already analyzed in class. For instance, presenting Borůvka's Algorithm for MSTs would stray far too close to our discussion of MSTS and Prim's/Kruskal's algorithms. However, there are some cases where the design idea is far enough away from what we've covered that presenting an alternative algorithm is of interest — For instance, there are many string matching algorithms or flow-related algorithms that would be sufficiently novel relative to what we've covered in class that I would support a project based on them!

There is also a question of *where* you should be looking to find these algorithms. My recommended starting point is part II (Ch. 14–21) of Skiena, which acts as a catalog of common algorithmic problems and standard solutions. Algorithms covered in the alternative text — CLRS,

linked to in the syllabus, or *Algorithms* by Dasgupta, Papadimitriou, & Vazirani, or any really other undergraduate algorithms text — would provide you with an algorithm that is guaranteed to be at the right level of difficulty while ensuring there is a reliable resource to learn about it.

Beyond this, Wikipedia's List of Algorithms is not an unreasonable starting point, though you will likely want alternative sources as well!

1.3 Sample Topics

In an effort to model what a reasonable problem/algorithm for this assignment looks like, here are a few examples of algorithms that would be at the right level to base a project on. Of course, these are not exhaustive — the most common use of this sections should be to get an idea of what an appropriate algorithm looks like before beginning your own research!

1. **The Stable Matching Problem/Gale-Shapely**. When medical students apply for residencies, each residency program ranks candidates and each candidate ranks programs. How do you find an assignment of candidates to programs such that no candidate/residency would turn down their assigned matching for each other? This is an instance of the Stable Matching problem (or, historically, the Stable Marriage problem). The canonical algorithm for solving it is **Gale-Shapely**. Variations of this problem are common — What if you want to maximize total preference rather than ensure stability? Then you get to the **Assignment Problem** and can use, for instance, the **Hungarian Algorithm** to solve it (which has echoes of Ford-Fulkerson!).

2. **Asymmetric-key Cryptography/RSA**. When we transmit sensitive information over the internet, we need to ensure that that information is not readable to any system through which that information transits. It's also infeasible to have you establish a secret encryption key with every online retailer you want to do business with offline. Asymmetric-key encryption schemes like **RSA** allow a retailer to share a public key that allows you to encrypt your data in a way where it can't be decoded without the retailer's secret key. Using this kind of algorithm requires a non-trivial notion of correctness that depends on (1) decryption with the secret key being correct **and** (2) decryption without the secret key being (as far as we know) computationally intractable, provable via showing integer factorization is in complexity class NP. This is an example of a problem that uses a lot of beyond-course knowledge (i.e., some basic number theory and some extensions of our notions of correctness), but is totally reasonable for the right group!

3. **Text-Compression/Huffman Coding** In a class like Computer Systems, you learn that (among many other things) strings of characters must be encoded into binary strings on disk. Usually each character is given a fixed-length binary code (like ASCII or Unicode), but for long texts it's helpful to compress documents by giving different symbols different length codes in an effort to make the entire text (with a representation of the encoding table) as short as possible. The **Huffman Coding** algorithm provides a scheme for determining the optimal variable-length encoding scheme for a particular text by framing the problem in terms of paths along binary trees. Correctness here also depends on both (1) the correctness of a decoding scheme and (2) the optimality of encoded document length. Unfortunately, **this topic is off-limits for students** since this will be the topic of my sample report!

4. **Convex Hulls** The convex-hull of a set of points is the smallest convex polygon that contains all of those points. Convex hulls are common intermediate representations in a number of applications — analysis of economic markets, determination of home ranges for animals, computation of Bezier Curves in computer graphics, etc. — without even mentioning it's interest to computational geometry generally. Correctness here depends on building an object with the requisite mathematical properties — normal for us! — but we must extend our intuitions to 2D geometry, which requires some comfort with geometric problem solving that we haven't developed here. Two algorithms to solve this are **Graham's Scan** and **Jarvis' March**.

This is not to mention a handful of other exciting problems with interesting applications — Simplex methods for solving matrix games, Odd's algorithm for solving the optimal stopping problems (i.e., the Secretary/Hiring Problem), and so on.

2 Report and Presentation Requirements

2.1 Report Requirements

You must submit before the day of your presentation a 3–5 page report that acts as a primer on the algorithm for other students who have taken the same course. It should be typeset in LaTeX, as you have been doing for the homeworks, and should be proofread thoroughly for clarity and presentability (minimal typos and formatting errors). The following sections are expected:

- An **Introduction and/or Applications** section that introduces (informally) the problem and a brief description of application areas where the problem arises.
- A **Formal Problem Definition** — as we've seen in class — that explicates the expected input and output for the problem, including any properties that define correctness for your choice of algorithm. Note that in some cases, correctness for your algorithm may be defined in a way that may be different than another algorithm for the same problem (i.e., an approximation algorithm for an NP problem may have correctness defined in terms of how far off the solution should be from an optimal one rather than requiring an optimal solution).
- A section describing your **Algorithm**. This should minimally include both an **informal description** of the key ideas underlying the algorithm's design (i.e., some way of looking at the problem, some design principle, or, like in Max-Flow, a mathematical result that motivates the technique) and **pseudocode** at an appropriate level of detail.
- A **Minimal Worked Example**. This should include a clear description of a small problem instance as well as a worked example of your chosen algorithm solving that instance. It should focus on clarity and exposition: What parts of the algorithm are trickiest, and how can you use this example to clarify how those steps work!
- An **Analysis of Runtime**, which provides a time complexity using the Big-O language we've developed as well as a brief mathematical argument for that time complexity. This should probably involve a **comparison with a naive solution** — typically a brute force solution to the problem.

- An **Analysis of Correctness**, which provides a mathematical argument that your algorithm meets the definition of correctness you've presented.

Depending on the problem/algorithm you choose, other things may be useful to include: For example, a graph or geometric problem may need some kind of visualization to efficiently present the algorithmic technique. For more mathematical problems (i.e., RSA), the level of formality may have to change to be able to introduce sufficient background while matching length requirements. **Beyond fulfilling the above content requirements, the primary grade determinant will be your technical communication skills:** How effectively can you present new technical algorithms material to peers with an undergraduate level algorithms background?

2.2 Final Presentation Requirements

The goal of the final presentation is to practice your oral technical communication skills, as well as to expand our survey of algorithms to those that were of interest to your peers. Thus you have two sets of responsibilities: One as a presenter, and another as an audience member.

As a presenter:

1. Present the problem, it's applications, and your chosen algorithm informally.
2. Briefly describe the motivation for the design of your algorithm, such that an attentive audience member understands how one gets from the problem definition to the core idea of your algorithm.
3. Briefly cover the formal requirements of a solution to the problem.
4. Sketch the core ideas for demonstrating the correctness of the algorithm. This should **not** involve going through a proof line by line, but should instead have you taking advantage of the fact that your audience has had a semester of a formal algorithms course. You should aim to spend much of your time on this requirement conveying the novel ideas of the proof, and relatively little gesturing at proof techniques and ideas we've covered extensively in class.
5. Present the efficiency gains of your algorithm over a naive solution. In an ideal presentation of many algorithms, this can be done with a single slide and a few sentences: "From this pseudocode, you can see that the outer loop runs X times and each loop takes Y time, giving us an XY time complexity. A brute-force solution would have to scan through Z potential solutions, requiring a solution that is big-omega of Z."

Note that your presentation should re-use much of the content of your report, but may need additional modifications and visualizations to fit the presentation format. Slides are not strictly required, but should be the default mode of presentation you consider. Deviation from this format should be justified.

As an audience member:

1. Before the presentation, be a responsible peer-reviewer of your assigned team's draft report. Roughly a week before the deadline, there will be an in-class day dedicated to reading a working draft of another team's report and commenting on clarity and readability (and having your draft read by that team). More details on the procedure will be available nearing that class session.

2. During the presentation, be an attentive audience member. You'll be asked to identify in-advance a presentation to assess, and then fill out a form identifying relevant details from that team's presentation.

Note that most of the presentation requirements will be evaluated on a pass/fail basis. If you're worried about aspects of your presentation not meeting these requirements, you should feel free to discuss this with me so I can clarify in advance. You should **not** be going into your presentation unsure whether you're meeting requirements!

2.3 Other Grading Factors

In addition to these content-based requirements, note that I reserve the right to deduct points based on things like failure to meet intermediate deadlines.

3 Timeline

For exact dates, check the course Moodle for relevant assignments. This section will summarize the rough timeline to give you an idea of how to manage your time commitment. Each entry discusses **beginning of week** deadlines!

Week 11: Instructions are released, teams and topics formation begin.

Week 12: Tentative teams and topics (in the form of a short paragraph pitch) are submitted to me for review.

Weeks 13/14: Progress is made outside of class on researching the algorithm and writing the report.

Week 14: In-class peer-review of draft reports.

Final Timeslot: Report due, Presentation day.

Note that final report and presentation work will be done in parallel to normal in-class content. Homework length has been adjusted to account for this expectation, but you should account for this in your own planning. **Final projects like this will require you to properly manage your time commitment to avoid sinking an unbounded amount of time into researching new, interesting algorithms!** Much of the work understanding the algorithm need not take an extraordinary amount of time — compare this assignment's requirement to learn a new algorithm to the time you spend on an individual lecture day's pre-class reading assignment! You should spend much of your time working on how to present the algorithm and it's correctness rather than doing additional research.