

CS256 PROBLEM SET ADVICE

Instructor: Bill Jannen

1 General Advice

Start early. Algorithm design takes time, and even algorithms for small problems can be surprisingly tricky to develop. I recommend reading over the problems as soon as the problem set is released so that you have the time to ponder them over the course of the week.

From my experience as a student in 256, passively thinking about the problem in the background of your day-to-day activities helps, even if you aren't consciously aware that you're doing it.

Collaborate and seek help, but explore on your own first. Although I strongly encourage collaboration, group discussions, and seeking help from TAs, I recommend that you attempt each of the problems and give them some independent thought before reaching out to others. If you let your thinking be “seeded” by other ideas, you won't have your own inspirations to bring to the discussion.

The homework problems tend to have solutions that are not particularly complicated but that require some insight to discover. If you immediately start working on the assignments in a group, you will miss out on the opportunity to come up with these insights on your own. Group work will help you hone your ideas to the more concise and beautiful versions; it can also serve to get you “un-stuck”.

Don't submit your first draft. When you come up with an algorithm and its analysis, your first iteration will likely have some rough edges and extraneous content. Taking the time to make your proofs as concise as possible and clarify your algorithm will cement your understanding of the material. It will also help your overall assignment grade—you will likely notice gaps in your logic or bugs that you did not see the first time.

In contrast to programming assignments, unfortunately there is no autochecker for proofs—so you have to ‘debug’ your proofs by rereading them yourself.

2 Algorithms and Proofs

Algorithm write up. In this course, you will often be asked to *give, develop, design or describe* an algorithm for a particular problem. The expectation is that you give a clear concise and complete description in prose of how your algorithm works. As you write up an algorithm, you need to present enough detail so that the reader can accurately analyze the algorithm's correctness and run-time, but not so much detail that the high-level idea isn't clear. Low-level detailed pseudo-code is often hard to understand (and is insufficient on its own).

Running time and correctness. If you are asked to design an algorithm, unless otherwise stated, you must also analyze its running time and correctness. Often the problem will require that your algorithm be of a certain time complexity, e.g. linear time, in which case you must argue why it meets those requirements.

- **Running time.** For running time analysis, you do not need to invoke formal first-principles of how O and Ω are defined but rather a more high-level argument that accounts for the various operations of your algorithm. As the course proceeds, we'll learn several techniques to analyze running time.
- **Space.** While it is good practice to include an analysis of the space complexity of the algorithm (and we will often do so in class), you are not required to include it unless the question asks for it explicitly.
- **Correctness.** When proving that your algorithm works correctly, you must give a rigorous mathematical proof. It is always a good idea to also include the general idea of why your algorithm works, but that does not constitute a proof. We will learn various techniques for proving correctness of different types of algorithms throughout the course, supplementing those you have already learned (e.g. induction in CS136).

3 Grading

Solutions to assignment questions should be written cleanly and concisely. Writing good proofs is both about having the correct logic, and effectively expressing that logic to the reader.

Your solutions will be graded on their correctness and clarity. An example rubric for a 10-point proof-based question is presented below.

- 10:** The solution is clear and correct.
- 9:** The solution is clear but contains a few mistakes, but they are minor or of little significance.
- 8:** The solution hits on the main points, but has at least one logical gap.
- 7:** The solution is significantly unclear or contains major gaps, but parts of it are salvageable.
- 4:** The solution is just plain wrong or so unclear it cannot be followed.
- 0:** No attempt is made at solving the problem.

4 LaTeX Typesetting Requirements

Each assignment will also have a (small but noticeable) number of points dedicated to LaTeX typesetting. This is to encourage good habits and correct usage. The following is a list of common mistakes to bear in mind while typesetting LaTeX—the list is not meant to be exhaustive.

- All variables and equations should be in math mode—one should write $O(n)$ rather than $O(n)$, and $n < m$ rather than $n < m$. Both inline math mode (using $\$...\$$) and display math mode (using $\[...\]$) are acceptable.
- Whitespace and indentation should be done with correct LaTeX usage. The command \backslash should only be used to force a line break when necessary, not to end a paragraph (when a blank line would do).
- Text should always fit on the page, as otherwise it is impossible to read. Here is an example of text not fitting on the page.
- Math mode should not be used except to typeset math. To italicize text, use $\textit{}$.
- Environments should be used correctly—in particular, solutions should be within the designated solution environment.

One quick sanity check to see if the above requirements are being followed is to check for LaTeX errors during compilation—oftentimes, a LaTeX error indicates that you are doing something wrong.

Acknowledgment

This handout is based on similar handouts by Tim Roughgarden, Keith Schwarz, and Shikha Singh. The rubric is adapted from Tom Murtagh and the LaTeX guide is from Sam McCauley.