

CS256 PROBLEM SET ADVICE

Based on handouts by Tim Roughgarden and Keith Schwarz

General Advice

Start early. Algorithm design takes time, and even simple algorithms 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 play around with them over the course of the week. In my experience, sometimes just having the problem in the back of your head for a while helps, even if you aren't actively working on it.

Collaborate, seek help but work on your own first. Although you are encouraged to collaborate, work in groups, and seek help from TAs, I strongly recommend not doing so until you have tried each of the problems and given them some independent thought. The homework problems tend to have solutions that are not particularly complicated but which 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.

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 or unnecessary parts. Taking the time to clean up your proofs and clarify your algorithm will both cement your understanding of the material and help your overall assignment grade. You will also notice gaps in logic, or bugs that you did not 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.

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 runtime, but not so much detail that the high-level idea isn't clear. Low-level detailed pseudocode is often hard to understand (and is insufficient on its own). Pseudocode should only be used if it is absolutely crucial to your analysis.

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 will learn many techniques that can be used to analyze running time.
- **Space.** While it is good practice to include an analysis of the space complexity of the algorithm (and we will 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 make up for a proof. We will learn various techniques for proving correctness of different types of algorithms throughout the course—and you already have learned some (e.g. induction) in CS136.

Grading

Correctness. This is the primary component of your grade. If you turn in a working algorithm with a valid correctness proof and runtime analysis, you'll get full points here. If your algorithm is buggy, or your correctness proof is flawed, or your runtime analysis is incorrect, you will lose some of these points.

Clarity. While correctness is important, one of the goals of this course is to learn how to write clear and elegant algorithms and proofs. If your solution is difficult to understand or is unnecessarily complicated, you stand to lose points. On the other hand, if you have a simple algorithm with a short but elegant proof of correctness and runtime bounds, you stand to gain bonus points.