Studying Guide

Exam Studying Guide

> Courtesy of Josh Hug, with slight modifications for CS 61A.

Studying for an exam is about gaining a level of familiarity with the material such that you can solve interesting problems that aren't just repetitions of things you've already seen. Some general tips:

1. Start early and spread things out over time. Sleep does some sort of magic where things sink in better. More shorter study days is way preferably to a few short ones. Indeed, one of the reasons we have homework with deadlines is to exploit this fact.
2. Be as active in your studying as possible. Working through problems and developing cheat sheets are "active". Rewatching lectures and rereading something you've read is not.
3. The often missing component: Reflect on your own problem solving process. This is much easier through discussion with others, particularly other students (see suggested workflow below).

For those who feel behind

If there are some lectures that you feel very shaky on, watch the video and/or do the reading ASAP. Then, immediately afterwards, try to work through some of the problems on the mentoring and discussion handouts. Each handout is designed to ramp up in difficulty the further you get through the worksheet. Try to answer at least the first few questions in each section. As you work, add the seemingly most important items to your cheat sheet.

Once you feel up to date

Once you know the basics, doing the readings, reading the lectures, and rewatching the lectures is unlikely to do much good (in my opinion).

A better approach is to work through problems from old exams, as well as problems from mentoring and discussion handouts.

An even better approach is to follow up your attempt to solve a problem with careful reflection on your solution process. This is much easier to do with other people around.

My suggested approach for studying for the exam is as follows:

1. Form a small study group.
2. Meet regularly, agreeing on a set of problems that everyone should attempt before each meeting.
3. At each meeting, have each person lead the group through their own solution to one of the problems. Everyone else should interject with their thoughts. It can be rather enlightening to see what shortcuts or inefficiencies you'll uncover. Repeat until you've covered all problems and everyone has had a chance to talk.

I did this for my obscenely difficult graduate probability and statistics class in grad school, and I really wished at the time that I'd discovered this process sooner. Your mileage may vary.

Ideally, this group would meet throughout the semester, and not just in the days leading up to an exam.

Solving Problems

> Based on original research by Loksa, Ko, et al. (http://dx.doi.org/10.1145/2858036.2858252)

Reinterpret the problem prompt

Read *and reinterpret* the question. Usually, we begin with a description of the problem to be solved. What's important is not just reading the problem, but thinking critically about the implications of the details in the problems and clear up any ambiguities. When we jump into coding directly without first thinking through the problem and posing questions for ourselves, we often run into scenarios where we get stuck and need to ask ourselves, "What should I put here?" or "What is the right loop end condition?" This increases the cognitive load by requiring us to context-switch and remove ourselves from the problem while we answer a side question.

A couple concrete starting questions to ask yourself on any problem include:

- What is the **domain** (input) and **range** (output) of the program?

- Restate the intended behavior of the program in your own words.
- How will the values in this program change as the program executes?

Verify your understanding by studying the doctests. In computer science, the mental representation for a problem is often closely related to its solution.

Big hints are always given away in the doctest! The doctests inform us about the shape and format of the solution. If we look closely enough for the patterns in the doctest, we'll often expose details in the structure of how the problem is meant to be solved.

Although they provide many hints, the doctests are not exhaustive and they usually don't show the most important cases. Develop examples that cover at least the following situations:

- What's the smallest or simplest possible input I could give to this function?
- Is there a similar small input that is *invalid* for this problem? How is it related to or different from the earlier case?
- Can we come up with any larger inputs to the program that are related to or rely on smaller cases? The idea is to come up with some of the subproblems we might have to solve with recursion or other techniques.

Search for analogous problems

Does this problem look similar to something you've seen before? Armed with your experience from homework, lab, and discussion, develop a general idea of how to solve the problem.

Once we've identified a similar problem, we can then extract the general strategy for solving the problem. While details are useful, copy-and-pasting the solution from the analogous problem usually won't get us very far. Instead, verbalize the code and reinterpret it in English by asking, "What's the purpose of including this code?"

Adapting previous solutions

Implement a solution by applying the problem solving techniques you've learned alongside your experience with analogous problems. With recursion, for example, it helps to try to follow the steps of finding a base case, identifying the recursive calls, and then combining the results. However, the particular implementation in code will depend upon the specific details of the problem.

This is where our rigorous understanding of the problem will come in handy. We found an analogous problem that has a similar, but not exactly the same behavior, so we have a **general approach** in mind. We know the domain, range, and behavior in the correct program. Using *problem-solving techniques* learned in class, *apply the general approach* to the particular problem to come up with a rough draft that is a step towards the solution.

It might not be fully correct, but that's fine and completely normal; refining mental representations of the problem takes time and practice.

Evaluating solutions

Analyze and test the resulting implementation. We'd like to answer two central questions:

1. Is my approach on the right track? If not, maybe we should consider another analogous problem.
2. If my approach is in the right direction, **let's evaluate** and verify the correctness of the solution.

To improve our code, we just need to ask ourselves the right questions. What input would break the program? Think like Python: run through the code step-by-step until there's a problem. We have examples of what the output should look like, so make sure the actual result matches expectations.

If the results aren't consistent, let's try to identify why and make adjustments by asking more specific questions. Where is the root of problem? Let's trace back through the code to find the source of the problem. Then, once we've found the problem, let's try the same approach of searching for analogous problems, except on this one, particular subproblem.

CS 61A (/)

Weekly Schedule (/weekly.html)

Office Hours (/office-hours.html)

Staff (/staff.html)

Resources (/resources.html)

Studying Guide (/articles/studying.html)

Debugging Guide (/articles/debugging.html)

Composition Guide (/articles/composition.html)

Policies (/articles/about.html)

Assignments (/articles/about.html#assignments)

Exams (/articles/about.html#exams)

Grading (/articles/about.html#grading)