CS 61A
Fall 2018

Introduction

Discussion 0: August 22, 2018

1 Lost on the Moon

Your spaceship has just crashed on the moon. You were scheduled to rendezvous with a mother ship 200 miles away on the lighted surface of the moon, but the rough landing has ruined your ship and destroyed all the equipment on board except for the 15 items listed below.

Your crew's survival depends on reaching the mother ship, so you must choose the most critical items available for the 200-mile trip. Your task is to rank the 15 items in terms of their importance for survival. Place a number 1 by the most important item, number 2 by the second most important, and so on, through number 15, the least important.

Item	Your Rank (1)	Group's Rank (2)	NASA's Rank (3)	$\lvert(3)-(1)\rvert$	$\lvert(3)-(2)\rvert$
Box of matches					
Food concentrate					
50 feet of nylon rope					
Parachute silk					
Solar-powered portable heating unit					
Two .45 caliber pistols					
One case of dehydrated milk					
Two 100-pound tanks of oxygen					
Stellar map (of the moon's constellations)					
Self-inflating life raft					
Magnetic compass					
5 gallons of water					
Signal flares					
First-aid kit containing injection needles					
Solar-powered FM receiver-transmitter					
Total					

Secrets to Success in CS 61A

CS 61A is definitely a challenge, but we all want you to learn and succeed, so here are a few tips that might help:

- Ask questions. When you encounter something you don't know, *ask*. That is what we are here for. This is not to say you should raise your hand impulsively, but you are going to see a lot of challenging stuff in this class, and you can always come to us for help.
- Study in groups. Again, this class is not trivial; you might feel overwhelmed going at it alone. Work with someone, either on homework, on lab, or for midterms, as long as you don't violate the cheating policy!
- Go to office hours. Office hours give you time with the instructor or TAs by themselves, and you will be able to get some (nearly) one-on-one instruction to clear up confusion. You are *not* intruding; the instructors and TAs *like* to teach! Remember, if you cannot make office hours, you can always make separate appointments with us!
- Do (or at least attempt seriously) all the homework. We do not give many homework problems, but those we do give are challenging, time-consuming, and rewarding. The fact that homework is graded on effort does not imply that you should ignore it: it will be one of your primary sources of preparation and understanding.
- Do all the lab exercises. Most of them are simple and take no more than an hour or two. This is a great time to get acquainted with new material. If you do not finish, work on it at home, and come to office hours if you need more guidance!
- Optional lab questions are "optional" in the sense that they are extra practice, not that they are material that's out of scope. Make sure you do them if you have time!
- Do the readings before lecture! There is a reason why they are assigned. And it is not because we are evil; that is only partially true.
- When preparing for the midterms and final, do past exam questions! Lecture, lab, and discussion provide a great introduction to the material, but the only way to learn how to solve exam-level problems is to do exam-level problems.
- Most importantly, *have fun!*

Solving Problems

Based on original research by Loksa, Ko, et al.

Note that this advice may not make sense until you start solving problems on your own. As you work through your first assignments, try referencing this guide.

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.

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.