CS161
Fall 2014

Homework Advice for Algorithms

Based on a handout by Tim Roughgarden, with edits by Keith Schwarz and Sophia Westwood

This handout contains information about the homework sets for CS161 – how to approach them, what we're looking for when grading, and so on. We hope that this will prove useful as you work on the homework.

General Advice

Here are some general pointers about how to approach the problems on the the homework:

- **Look over the problems early**. Algorithm design takes time, and even simple algorithms can be surprisingly tricky to develop. We suggest reading over all the problems as soon as the homework goes out. Starting the night before is a particularly bad idea in this class, where insights often come only after taking a break from a problem and returning to it later on.
- **Don't submit your first draft**. When you come up with an algorithm and a correctness / runtime proof for it, your first iteration will likely have some rough edges or unnecessary parts (similar to a first draft of an essay). We strongly advise against submitting your very first draft of your answers. 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.
- **Work on your own before discussing in a group**. Although you are allowed to discuss the assignments in groups of up to three, we strongly recommend against doing so until you have already thought in-depth about each of the problems on your own. We recommend spending perhaps an hour working through each problem on your own before consulting others. This time is crucial for developing your algorithmic problem-solving skills, a key part of the class (and of course necessary for the exams as well!). Just to reiterate, if you do collaborate you still must write up solutions individually and acknowledge anyone with whom you have discussed the problems.

Submission Instructions

When submitting homework sets, **please be sure to put each problem on its own page or pages, each with your SUNet user name.** This makes the assignment easier to divvy up across TAs.

Send an email with a PDF copy of your answers to the submission mailing list (psetCS161@gmail.com) **Please submit your assignment as a single PDF under 5MB.** Send from your Stanford email address with the **subject line** “yoursuid : HOMEWORK #hwid” – for example, “sophiasw : HOMEWORK #1”

Citation Policy

When writing up your solutions, you are free to use any result from lecture that you would like as long as you provide a citation. If you are uncertain about whether you may cite a result not covered in lecture, ask via a private post on Piazza.

If you work in a group, please make sure to cite all people you worked with. As mentioned before, you must write up your own solutions to each problem.

How We Grade and Writing Proofs

In your assignments, aim to be clear, precise, and concise. Understandability is an important factor in the grading, so be sure to present a logical argument using the proof-writing skills learned in CS103. If you feel like you could use a refresher on how to write proofs, or on the different types of proofs (direct proof, proof by contradiction, proof by induction, etc) you are encouraged to revisit the slides from the current offering of CS103 (cs103.stanford.edu) and even the SCPD videos recorded for the class this quarter (myvideosu.stanford.edu). It can also help to take results proved in the textbook, rewrite these proofs on your own, and then compare back with the textbook.

Of course, your answer must also be correct, with a working algorithm complete with a valid correctness proof and runtime analysis. If your algorithm is buggy, or your correctness proof is flawed, or your runtime analysis is incorrect, you will lose some of these points. You may also lose points if your algorithm is correct but has unnecessary special-cases or could easily be simplified. Similarly, we will deduct points for proofs that are unclear or are far more complicated than necessary.

Writing strong proofs is a key skill to develop to succeed in this course. We encourage you to approach the homework as an opportunity to spend time finessing your proof-writing skills. While many students are intimidated by writing proofs, it is a skill you can develop and learn via practice and attention. If at any point you're curious whether one of your answers is sufficiently clean, you can always stop by office hours to have one of us look over it.

Describing Algorithms

When writing up an algorithm, you'll need to provide a description of how that algorithm works, often via a combination of English and psuedocode.

As you write up an algorithm, you need to present enough detail so that you can accurately analyze the algorithm's correctness and runtime, but not so much detail that the high-level idea isn't clear. It's perfectly fine to describe an algorithm in plain English as long as there's enough detail to recover how the algorithm works; in fact, we'd prefer this if possible.

You should try to avoid unnecessarily detailed pseudocode unless it's absolutely necessary for your analysis. Low-level detailed pseudocode is hard to understand, so if you do provide pseudocode be sure to provide lots of comments!

Proving Correctness

When designing algorithms, we require that you write a proof of correctness. This should be a rigorous mathematical proof, rather than a general idea about why the algorithm works.

In many cases, we will ask a problem in plain English without providing any mathematical notation. In that case, we suggest introducing symbols or terminology as appropriate when writing the proof. Just make sure that what you have is still easy to read!

Writing a correctness proof doesn't necessarily require going line-by-line through your algorithm and confirming that each step works. If the operation of the algorithm is clear, you can just prove that the general process it follows will work correctly without referring to why each particular step in the algorithm is correct.

Analyzing Runtime

When analyzing the runtime of an algorithm, you do not need to call down to the underlying definition of O, Ω, or Θ notation when proving upper or lower bounds. It's fine to do an informal analysis (e.g. multiplying work done across all iterations by the number of iterations, or accounting for the work done by various code blocks across the entire algorithm runtime). If you want to show that a function runs in time $\Theta(f(n))$, be sure to justify why your bound is tight. One option would be to prove that the function runs in time $O(f(n))$ and $\Omega(f(n))$. Another option would be to do a more precise accounting of the total work done by the function.

In some cases, we will ask you to use the formal definition of O, Ω, or Θ notation in a proof. In that case, be sure to use the formal definitions of these terms.

Sample Problem

To give you a sense for what a good homework solution might look like, here is a sample problems along with a solution that would earn full credit. We suggest taking some time to work through this problem so that you have a sense for how to solve it.

Sample Problem: Array Partitioning

You are given an array A and a predicate P. Design an algorithm that rearranges the elements of A so that all elements for which P is false appear before all elements for which P is true. Your algorithm should run in $O(n)$ time and use $O(1)$ additional space, where n is the size of the array.

Solution to Sample Problem

Our algorithm is as follows: maintain two indices r and w, both initially 0. Then, repeat the following:

- If $r = n$, the algorithm terminates.
- If $P(A[r])$ is false:
    - Swap $A[r]$ and $A[w]$
    - Set $w = w + 1$
- Set $r = r + 1$

Correctness:

To show correctness, we will prove that the following is always true at the top of the loop:

(★) P is false for all elements in $A[0 \ldots w-1]$ and true for all elements in $A[w, r-1]$

Assuming (★) is always true at the top of the loop, then when the loop terminates (that is, $r = n$), we have that P is false for all elements in $A[0 \dots w - 1]$ and true for all elements in $A[w, n - 1]$. Thus all elements for which P is false precede the elements for which P is true.

To show that (★) always holds, we proceed by induction. After 0 iterations of the loop, (★) is vacuously true because $A[0 \dots w{-}1] = A[0 \dots -1]$ is empty and $A[w, r{-}1] = A[0 \dots -1]$, which is also empty.

Now assume that after some number of iterations that (★) is true. We will prove that (★) is true after the next iteration. Let r_0 be the initial value of r on entry to the loop and w_0 be the initial value of w on entry to the loop. Now, either $P(A[r_0])$ is true or it is false. We consider these independently:

Case 1: $P(A[r_0])$ is true. By (★), we know P is false for all $A[0 \dots w_0 - 1]$ and true for all $A[w_0 \dots r_0 - 1]$. Since $P(A[r_0])$ is true, we know that P is true for all $A[w_0 \dots r_0]$. Since at the end of the loop $w = w_0$ and $r = r_0 + 1$, this means (★) still holds at the end of this iteration.

Case 2: $P(A[r_0])$ is false. By (★), we know P is false for all $A[0 \dots w_0 - 1]$ and true for all $A[w_0 \dots r_0 - 1]$. We then swap $A[r_0]$ and $A[w_0]$. Since $A[r_0]$ is false, this now means that P is false for all A$[0 \dots w_0]$. If $w_0 = r_0$, then it's vacuously true that P is true for all elements in the range $A[w_0 \dots r_0 - 1]$. Otherwise, if $w_0 \neq r_0$, then since we know that $A[w_0]$ was true on entry to the loop iteration, we now know that P is true for all elements in $A[w_0 + 1, r_0]$. Since at the end of this iteration $w = w_0 + 1$ and $r = r_0 + 1$, this means(★) still holds at the end of this iteration. ■

Runtime:

To show that this algorithm terminates in O(n) time, note that the loop runs n times, each time doing O(1) work. Therefore, the algorithm runs in O(n) time.

Space Usage:

Since the algorithm only needs to store r, w, and O(1) temporary variables beyond its parameters, it uses only O(1) additional space.