Algorithms Problem-Solving Guide

(This template has been adapted from the template developed by Stanford IDEAL Research Lab)

Blue - Examples
Purple - To dos
Red - Note

Part 1: Planning

1. **Read and Interpret**: What is the objective of the problem?
    1. What are the parts of the problem that have the question in it? What information is relevant for solving the problem?
    2. How can you simplify the problem statement?
    For example, if it is a "story problem," can you abstract out an algorithmic problem statement?
    3. What assumptions and constraints are there?
    For example, for designing an algorithm, does the problem tell you what the running time or space complexity should be? For proving a statement, are there requirements on the type of proof you should produce?
    4. What resources do you have access to?
    For example, are there assumptions given in the problem that you can take advantage of? For designing an algorithm, does the problem statement suggest that you use a particular data structure, sub-routine, or approach?

2. **Identify the type of problem:** What are you being asked to do in the problem (*select one or more)*
    1. Design an algorithm and present it either in pseudo-code or with a high-level description
    2. Prove the correctness of an algorithm
    3. Analyze the time complexity of an algorithm
    4. Prove a claim, theorem, lemma, etc.

    For CS161 HW and Exam problems, we will always have a "We are expecting" block at the end of the problem to clarify what we are looking for. Unfortunately, real life isn't always like this. But in either real life or in CS161, you can always ask if you are not sure!!

3. **Work out an example**: Based on the type of problem you identified in part 2, go through examples!
    1. **Designing an algorithm**: To make sure you understand the algorithmic problem, choose a specific small version of the input, and see what the output should be. Do this for a few different inputs. (It's okay if you don't have a fast algorithm to do this yet – for a small example, you can work out the output via an easy but slow algorithm, like brute force). If the expressions are proving too unwieldy, try using a substitution (*for example, if you have terms that are all functions of log n, you might want to substitute log n = y)*. While

you are doing this for a few examples, try to understand what the "easy but slow" algorithm is, and how slow is it?

2. **Proving the correctness of an algorithm**: Before proving the correctness of an algorithm, you should make sure you understand what the algorithm is doing. To do this, pick a small specific example input (or a few of them), and run through the algorithm by hand. While you are doing it, think about why it's working to get intuition for your proof.
3. **Analyzing the time complexity of an algorithm**: As with proving the correctness, you should first make sure you understand what the algorithm is doing, so work through an example run on a small input!
4. **Proving a claim/theorem/lemma**: Before proving something, you should understand what it is you are trying to prove. Usually the thing you are trying to prove will have the form "Suppose X. Then Y." Pick a small example where X holds, and try to convince yourself that Y also holds in this case.

4. **Similar Problems:** Based on the type of problem you identified in part 2, are there similar problems that you have seen before in lecture, past homeworks, the textbook, etc.? Might these examples be useful for inspiration (e.g., you may be able to tweak them), or might they be useful tools to solve the problem (e.g., you could use an algorithm we've seen in class as a black box)?

   Usually there's no way to know this for sure up front! Right now we're just gathering ideas.

   1. **Designing an algorithm:** Think about:
      1. Algorithmic techniques that we've seen so far in the course (*for example, divide-and-conquer; greedy algorithms; dynamic programming*).
      2. Tasks that we've seen so far (*for example, sorting; shortest path in a graph; matching*).
   2. **Proving something:** Think about:
      1. Proof techniques we've seen (*for example, proof by contradiction; proof by induction*).
      2. Similar proofs we've seen in class (*for example, if you are trying to prove the correctness of a divide-and-conquer algorithm, go back and look at the proof of correctness for MergeSort that we saw in class*).
      3. Statements that we've proved in class that might be helpful.
   3. **Analyzing the time complexity of an algorithm:** Think about:
      1. Techniques we've seen so far (*for example, analysis of recurrence relations and the master theorem, if your algorithm is recursive*).
      2. Examples of such analysis *(for example, if you are trying to analyze the time complexity of a divide-and-conquer algorithm, go back and look at some of the analyses we did in Lectures 2/3/4*).

5. **Information Needed**: Based on identified relevant concepts and or similar problems, what other information do you need to know to solve the problem?

For example, perhaps you may want to go back and refresh your understanding of a technique or an algorithm you've already seen.

6. **Smaller problems:** If this problem seems too tricky, try to identify an easier problem that you still don't know how to solve.
   For example, if the problem asks you to solve something for general n, try solving it for a special case where n=5. Or try solving it under an additional assumption (e.g., "I will assume that all the elements in the array are distinct.") Try Part 2 below for these smaller problems, to get an idea about how to approach the big problem.

Part 2: Execution

1. **Solution Plan:** Hopefully as part of Part 1 (or running Part 2 already with smaller/simpler problems), you've gathered a bunch of ideas about how to approach the problem. For each approach you have in mind, create an outline for your solution.
   1. **Designing an algorithm.** *For example, if you are going to try divide-and-conquer, how are you going to break up a big problem into smaller problems? How are you going to combine the solutions of those smaller problems to solve the big problem? If you are going to try DP, what are the subproblems that you will consider, and how do solutions to the smaller ones help you solve the bigger ones? If you are going to try a greedy algorithm, what is the quantity you are greedily maximizing at each step?*
   2. **Proving the correctness of an algorithm.** For example, if you are going to try a proof by induction, what is the base case? What is the inductive hypothesis? What would you need to show to establish the inductive step?
   3. **Analyzing the time complexity of an algorithm.** *For example, if you are going to try to use the Master Theorem to analyze the running time of an algorithm, you'll need to write down a recurrence relation: how are you going to get that recurrence relation out of the algorithm description?*
   4. **Proving a claim/lemma/theorem.** *The same example with induction holds above. More generally, if you have an idea for a proof outline (perhaps inspired by another proof we've seen, or your own intuition), write down that outline. What smaller claims/lemmas/theorems would you need to prove to fill it in?*

2. **(Try to) solve, and iterate!** Try going through the steps of your solution plan above. If it just works, great! But if not, try to figure out why not.
   1. Is it something that can be fixed by a small tweak to the same solution plan?
      For example, maybe it feels like divide-and-conquer will work, but you're just not figuring out how to combine the sub-problems back together correctly – in that case, repeat the whole process again with the smaller problem "how do I combine the sub-problems correctly?"
   2. Is it something that seems like an insurmountable problem with this approach?
      For example, maybe you are supposed to get an algorithm that runs in time O(log n), but your solution plan inevitably has to look at all n input items. In that case, scrap this solution plan and try a different one!

3. **Getting (Un)stuck**: After you've tried the "Try to solve" step for a while, with a few different solution ideas, it might be time to step back. First, make sure you really understand why your current approaches aren't working. Perhaps it's because the approach just won't work; or perhaps it will work but you are missing a key piece of information or a key concept. Try to articulate the challenge you have as best you can. For help articulating the challenge, you can:
    1. Try to explain your approach to a classmate or a CA (in OH or on Ed), and see if together you can identify the sticking point.
    2. Work through a few more examples to see where your current solution plan isn't working.

    Once you understand why your current approaches aren't working, often that will inspire some new approaches (and you can return to Part 2, Step 2 with those approaches); or it will tell you that you need to understand a particular concept better (and you can Part 1 to understand that concept). But if you are all out of approaches, you need some new ones! To get some:
    3. Brainstorm with your HW group.
    4. If your HW group is stuck, brainstorm with other CS161-ers – you may be able to find some at OH or at the Homework Parties!
    5. Return to Part 1 and see if there are any examples or course content you may be able to draw on for more inspiration.
    6. If you are still stuck, tell a CA what you've tried (in OH or on Ed), and they may be able to point you in the right direction. (But please don't ask the CAs to tell you the solutions! Also if you are going to post what you tried on Ed, please make it a private post to avoid spoiling the problem for others.)

Part 3: Answer Checking

1. **Write it up carefully!** Often trying to write up solutions carefully is the best way to spot mistakes. A good strategy is to put your notes aside and try to write up the solution from scratch – that way you're less likely to repeat any mistakes you may have made in Parts 1-2.
2. **Did you answer the question?** Go back to your answers at the beginning of Part 1 and make sure that you've checked all the boxes.
3. **Does your solution make sense?** Does your solution set off any red flags?
    - **Does your solution seem too good to be true?** *For example, if your solution implies a comparison-based sorting algorithm that runs in time O(n) [which we will show in Week 3 is impossible], maybe go back and double-check your work.*
    - **Does your solution use all of the assumptions in the problem?** Usually the assumptions are there for a reason. If you didn't use all of the assumptions, there might be something fishy going on. Similarly, for proofs, you should check if your proof proves the assertion even if you remove the assumptions. Is there a counterexample which shows the assumptions are actually necessary? That would imply your proof has a hole that needs fixing!

4. **Work through a few more examples** (if the problem requires you to do the following 2):

- **Designing an algorithm**. Run your algorithm (either by hand, or by coding it up) on a few examples. Do you get the right answer? If you do it out by hand, is the algorithm doing what you thought it would?
- **Proving something**. Walk through the proof for a particular example. Are all the claims that show up correct?

5. **If appropriate, share!** If you are working in a group, ideally the whole group is doing this same process in parallel. Come back together as a group and make sure you all got the same answers![1]

Part 4: Final Reflection

1. **Reflect**: What did you find challenging about solving this problem and what did you learn in solving this problem or from asking for help to solve this problem? In particular, what have you added to your arsenal for doing Part 1 next time?
2. **Do you have any remaining questions?** If yes, ask a classmate or CA to help you answer them!

---

[1] Divide and conquer (aka, "Person 1 does problem (a), Person 2 does problem (b), etc")) can be really tempting for group HW. After all, it's so efficient! However, while it may be efficient for running time, it's not the best choice for either correctness or for everyone's learning! Instead, we recommend duplicating some work. *For example, if you do Parts 1 and 2 (brainstorming and problem-solving) as a team, but each do Part 3 (answer checking) on your own, then you can be extra sure your answer is correct, and also that everyone on the team understands it!* [As an aside, if you want to understand the most efficient way for a computer to duplicate work to check for errors, take CS250, Error Correcting Codes!]