HOW TO ACE AN ALGORITHMS INTERVIEW

We wanted to make a guide to help candidates on Technical interviews. Algorithms are a big part of a computer science interview.

It usually starts like this:

Given X, figure out an efficient way to do Y.

First, make sure you understand the problem. Don't hesitate to ask questions. Specifically, if any of the problem requirements seem loosely defined or otherwise unclear, ask your interviewer to make things more concrete. There is no penalty for asking for clarifications, and you don't want to miss a key requirement or proceed on unfounded assumptions. This will also buy you time if your brain isn't kicking in right away. Nobody expects you to solve a problem in the first 30 seconds or even the first few minutes.

> Once you understand the problem, strategize how to answer—any solution whatsoever. As long as it's valid, it doesn't matter if your solution is trivial or ugly or extremely inefficient. What matters is that you've made progress. Creating a strategy to answer forces you to engage with the content of the problem, priming your brain for improvements you can make later, and gives you something in the bank, which will in turn give you confidence. If you can achieve a brute force solution to a problem, you've cleared a major hurdle to solving it in a more efficient way.

After creating a plan, it's time to execute your answer. When answering, do not rush. Use the structure of your response that you have planned to carefully exemplify and communicate your solution, whether it is verbal or written code.

Tips for answering when stuck:

Almost all of these can help on almost any problem

1. Start writing on the board. Candidates often getstuck while staring at a blank wall. It is more productive to stare at some examples of the problem as a lead, than to stare at nothing. If you canthink of a picture that might be relevant, draw it. If there's a medium- sized example you can work through, go for it. (time permitting) Or just write down some propositions that you know to be true. Anything is better than nothing.

2. Talk it through. Don't worry about sounding stupid. If it makes you feel better, tell your interviewer "I'm just going to talk out loud. Don't hold me to any of this." Sometimes people prefer to quietly contemplate a problem, but if you're stuck, talking is one way out of it. Sometimes you'll say something that clearly communicates to your interviewer that you understand what's going on. Even though you might not put much stock in it, your interviewer may interrupt you to tell you to pursue that line of thinking. However, make sure that thinking out loud does not become fishing for hints.

3. Think about algorithms and data structures, and work in a top-down method. Sometimes it's useful to mull over the particulars of the problem-at-hand and hope a solution stands out to you (this would be a bottom-up approach). But you can also think about different algorithms and ask whether each of them applies to the problem in front of you (a top-down approach). Changing your frame of reference in this way can often lead to immediate insight. Here are some algorithmic techniques that can help solve more than half the problems that are asked:

Algorithms:

- Sorting (plus searching / binary search)
- Divide-and-conquer
- Dynamic programming / memorization
- Greediness
- Recursion
- Algorithms associated with a specific data structure (which brings us to our fourth suggestion...)

Data Structures:

- Array
- Stack / Queue
- Hashset / Hashmap / Hashtable / Dictionary
- Tree / binary tree
- Heap
- Graph

You should be familiar with these algorithms and data structures inside and out (What algorithms tend to go along with each data structure?) By knowing and understanding these, sometimes the solution to a problem will pop into your mind as soon as you even think about using the right one.

4. Think about related problems you've seen before and how they were solved. Chances are, the problem you've been presented with is a problem that you've seen before, or at least very similar. Think about those solutions and how they can be adapted to specifics of the problem at hand. Don't get tripped up by the form that the problem is presented. Distil it down to the core task and see if matches something you've solved in the past.
5. Modify the problem by breaking it up into smaller problems. Try to solve a special case or simplified version of the problem. A reduction of the problem into a subset of the larger problem can give a base to start from and then work your way up to the full scope at hand. Looking at the problem as a composition of smaller problems may also be helpful.
6. Don't be afraid to backtrack. If you feel like a particular approach isn't working, it might be time to try a different approach. Of course you shouldn't give up too easily. But ifyou've spent a few minutes on an approach that isn't fruitful and doesn't feel promising, back up and try something else. Be willing to work backwards and abandonan unpromising approach.

Incidentally, trying out a few different approaches (rather than sticking with a single approach) tends to work well in interviews, because the problems we choose for an interview usually have many different solutions.

While You Are Coding

- Think out loud and spell out what's going on. Explain your thought process to your interviewer as you code. This helps you fully communicate your solution, and gives your interviewer an opportunity to correct misconceptions or otherwise provide high-level guidance. Even if you don't solve the problem efficiently enough, recognizing the issue and being able to explain your thought process counts.

- Openly discuss tradeoffs in algorithm and data structure decisions. Acknowledge weaknessesin your approach if you get stuck.

- Noticing bugs in your code and stating how they would be fixed is a plus.

- Avoid arrogance. If asked "can you further optimize your code?" run through an internal list of what you already did and determine if adjustments can be made.

- When writing code it's okay to write pseudo-code as an outline, then to develop final code on top for your final answer.

- Start simple and then expand. While you do want to think about the high-level design before you begin, it's good to come up with a simple solution first, and then build from there.

Remember: During technical interviews, hiring managers and engineers are interested in your general problem solving and coding abilities, more than your recall of library function names or obscure language syntax. If you can't remember exactly how to do something in your chosen language, make something up and just explain to your interviewer that you would look up the specifics in the documentation. Explain the shortcuts you took. If you skipped things for reasons of expedience that you would otherwise do in a "real world" scenario, please let them know what you did and why. For example, "If I were writing this for production use, I would check an invariant here." Since whiteboard coding is an artificial environment, this gives us a sense for how you'll treat code once you're actually on the job.

As an addendum, here are a few suggestions for books we like about the art of software construction:

- *Clean Code: A Handbook of Agile Software Craftsmanship* - Robert C. Martin
- *Code Complete: A Practical Handbook of Software Construction* - Steve McConnell
- *The Practice of Programming* - Brian Kernighan, Rob Pike
- *Design Patterns: Elements of Reusable Object-Oriented Software* - Erich Gamma, et al.
- *Effective Java* - Joshua Bloch
- Effective C++ - Scott Meyers