General Advice for Technical Questions

Interviews are supposed to be difficult. If you don't get every – or any – answer immediately, that's ok! In fact, in my experience, maybe only 10 people out of the 120+ that I've interviewed have gotten the question right instantly.

So when you get a hard question, don't panic. Just start talking aloud about how you would solve it.

And, one more thing: you're not done until the interviewer says that you're done! What I mean here is that when you come up with an algorithm, start thinking about the problems accompanying it. When you write code, start trying to find bugs. If you're anything like the other 110 candidates that I've interviewed, you probably made some mistakes.

Five Steps to a Technical Questions

A technical interview question can be solved utilizing a five step approach:

1. Ask your interviewer questions to resolve ambiguity.
2. Design an Algorithm
3. Write pseudo-code first, but make sure to tell your interviewer that you're writing pseudo-code! Otherwise, he/she may think that you're never planning to write "real" code, and many interviewers will hold that against you.
4. Write your code, not too slow and not too fast.
5. Test your code and *carefully* fix any mistakes.

Step 1: Ask Questions

Technical problems are more ambiguous than they might appear, so make sure to ask questions to resolve anything that might be unclear or ambiguous. You may eventually wind up with a very different – or much easier – problem than you had initially thought. In fact, many interviewers (especially at Microsoft) will specifically test to see if you ask good questions.

Good questions might be things like: What are the data types? How much data is there? What assumptions do you need to solve the problem? Who is the user?

Example: "Design an algorithm to sort a list."

» *Question: What sort of list? An array? A linked list?*

» Answer: An array.

» *Question: What does the array hold? Numbers? Characters? Strings?*

» Answer: Numbers.

» *Question: And are the numbers integers?*

» Answer: Yes.

» *Question: Where did the numbers come from? Are they IDs? Values of something?*

» Answer: They are the ages of customers.

» *Question: And how many customers are there?*

» Answer: About a million.

We now have a pretty different problem: sort an array containing a million integers between 0 and 130. How do we solve this? Just create an array with 130 elements and count the number of ages at each value.

Step 2: Design an Algorithm

Designing an algorithm can be tough, but our five approaches to algorithms can help you out (see pg 34). While you're designing your algorithm, don't forget to think about:

» What are the space and time complexities?

» What happens if there is a lot of data?

» Does your design cause other issues? (i.e., if you're creating a modified version of a binary search tree, did your design impact the time for insert / find / delete?)

» If there are other issues, did you make the right trade-offs?

» If they gave you specific data (e.g., mentioned that the data is ages, or in sorted order), have you leveraged that information? There's probably a reason that you're given it.

Step 3: Pseudo-Code

Writing pseudo-code first can help you outline your thoughts clearly and reduce the number of mistakes you commit. But, make sure to tell your interviewer that you're writing pseudo-code first and that you'll follow it up with "real" code. Many candidates will write pseudo-code in order to 'escape' writing real code, and you certainly don't want to be confused with those candidates.

Step 4: Code

You don't need to rush through your code; in fact, this will most likely hurt you. Just go at a nice, slow methodical pace. Also, remember this advice:

» Use Data Structures Generously: Where relevant, use a good data structure or define your own. For example, if you're asked a problem involving finding the minimum age for a group of people, consider defining a data structure to represent a Person. This

shows your interviewer that you care about good object oriented design.

» Don't Crowd Your Coding: This is a minor thing, but it can really help. When you're writing code on a whiteboard, start in the upper left hand corner – not in the middle. This will give you plenty of space to write your answer.

Step 5: Test

Yes, you need to test your code! Consider testing for:

» Extreme cases: 0, negative, null, maximums, etc

» User error: What happens if the user passes in null or a negative value?

» General cases: Test the normal case.

If the algorithm is complicated or highly numerical (bit shifting, arithmetic, etc), consider testing while you're writing the code rather than just at the end.

Also, when you find mistakes (which you will), carefully think through *why* the bug is occuring. One of the worst things I saw while interviewing was candidates who recognized a mistake and tried making "random" changes to fix the error.

For example, imagine a candidate writes a function that returns a number. When he tests his code with the number '5' he notices that it returns 0 when it should be returning 1. So, he changes the last line from "return ans" to "return ans+1," without thinking through why this would resolve the issue. Not only does this look bad, but it also sends the candidate on an endless string of bugs and bug fixes.

When you notice problems in your code, really think deeply about why your code failed before fixing the mistake.