MARCH 9, 2018 / #CODING

# How to make progress while studying for coding interviews

By Sam Gavis-Hughson

interviews all the time.

You put in consistent work. You've gone through hundreds of practice problems and block out an hour every day to study. But you're just not progressing.

Even after all this prep work, you still feel like every problem is a new challenge that you don't quite know how to solve. You can solve the problems most of the time, but it doesn't feel like it's getting that much easier.

When I work with 1:1 coaching clients, this is exactly the situation that I often find them in. And in almost all of the cases, once we identify the thing (or things) holding them back, they experience major breakthroughs. Resolving these issues has gotten my clients jobs at Amazon, Bloomberg, Uber, and more!

So what exactly is it that is holding you back? What is preventing you from making the progress you want? In this article, I will show you the ten most common problems that people struggle with. Fair warning, though: it can be really hard to identify these issues in yourself. If you really want to break through your issues, I recommend working with a coach.

## 1. Develop a strong foundation

I have talked about this issue before, but one of the most important keys to success in coding interviews is having a strong foundation of Computer Science fundamentals.

FAST Method. However, none of these techniques are valuable if you don't have strong fundamentals under your belt.

For example, the FAST Method is designed to help students with dynamic programming. The first step of the FAST Method is to find an initial brute force recursive solution. This is something that you need to be able to do on your own for the FAST Method to be of any use to you. Even if you understand the methodology, it won't help you if you don't know how to get that initial solution.

If you have strong fundamentals, then recursion is something that shouldn't be difficult for you. It is a topic that comes up frequently enough that you should be able to whip it out whenever you want.

Such is the case with all other fundamental data structures and algorithms. If you don't know how to implement a linked list, then it doesn't matter how many tricks you learn for solving linked list problems.

If you are lacking a formal Computer Science background, or simply find that the recommended problem solving techniques for coding interviews aren't really helping you, then your first step should be to dedicate yourself to learning all of the fundamental data structures and algorithms.

The best way to do that is to take either the MIT (Python) or Princeton (Java) data structures and algorithms course. Buy the book, do the assignments, and take the exams. If you put in the work, you can easily complete the course in 3 months or less, and you will

## 2. Get more coding experience

Have you ever tried to do something for the first time? When I was first learning how to play guitar, it would take me 30 seconds to get the fingering for a basic chord. In theory, I could play any song if I had enough time and had the chord charts in front of me, but it wouldn't sound very good.

Sometimes I work with students who are in a similar position with their coding. They simply have not reached a level of proficiency yet where they can easily write up the code in their interview.

In theory, the coding component of the interview should be the easy part. As long as you've practiced on a whiteboard, the hard part of your interview should be developing a solution in the first place before you ever write a line of code. If this is not true, then you probably need to improve your coding skills.

There are also some groups that are more affected by this issue than others. Primarily, those who have less experience coding. I often see bootcamp grads, Masters students who switched into Computer Science for their Masters from a different field, or long term managers who simply haven't coded in a while struggling the most with coding.

The key here is simply to get more practice coding, and ideally do so in an environment where you are getting good feedback on your code. One of the best ways to do this is by contributing to open source projects. Not only is this a great way to show off your experience, but you will get the benefit of thorough code reviews

great resource for those looking to get started. They share tutorials on how to contribute, and have compiled a list of potential projects that would be good for beginners.

## 3. Strategically approach each interview question

When going into battle, any general who expects to succeed has a detailed plan. If you want to succeed in your interview, you need to have a detailed plan as well.

The most common plan I see to solving interview problems looks something like the following:

1. Look at the problem
2. Think about the problem
3. Come up with a solution
4. Write the solution
5. Success

However, do you notice a problem here? Hopefully the first thing you asked was “How do I come up with a solution?” This plan totally lacks any sort of strategy for coming up with the solution. It assumes that the solution will just appear.

And that’s how most people think about interview problems. They memorize tons of solutions with the hope that one of them will be

This is risky at best.

A far better way to interview is to have a clear game plan for how you're going to approach each interview and each problem within the interview. Here is a rough outline of how you can approach an hour-long interview:

**[0:00–0:05]** Get settled and make sure that you fully understand the problem they're asking. Work through any example inputs provided.

**[0:05–0:10]** Figure out a brute force solution to the problem. No coding at this point, just talk through it and draw any pictures if you find that helpful. If you're stuck coming up with a brute force solution, try working through the problem by hand and translating your process for solving it into an algorithm.

[**0:10–0:15**] Optimize your solution. Take these 5 minutes to figure out the absolute best solution you can in this period of time. When comparing solutions, consider the time complexities.

**[0:15–0:35]** Code up your solution. Even if it's not optimal, it is better to have a complete, non-optimal solution than an incomplete, optimal solution.

**[0:35–0:50]** Test your code and fix any issues. This is incredibly important. It doesn't matter if your code isn't perfect the first time, but you had better be able to identify the errors.

**[0:50–1:00]** Questions for your interviewer.

the world is running out of time when you totally know how to solve the problem.

Second, these steps will help you ensure that you always get to a solution. By starting with the brute force solution and optimizing, you can almost guarantee that you will at least come up with **some** solution. Often times, the most optimal solution isn't necessary if you do well in the rest of the interview.

## 4. Consider different possible solutions

Did you know that most interview questions have more than one correct solution? I know, shocker, right? However, tons of people find a solution and then they just stop there without looking any further.

This always disappoints me. Often times, there is another solution that would have been even better and they were so close. Or maybe there were comparable solutions that had different tradeoffs.

For example, consider this problem:

- One solution has `O(n)` time complexity and `O(1)` space complexity
- Another solution has `O(log n)` time complexity and `O(log n)` space complexity

Which of these solutions is better?

better. However if memory is not an issue, then obviously we want to go with the second solution.

The key here is that while in some cases there may be one “best” solution, there are way more problems where you can make different trade-offs and you have to decide which ones to make. As an interviewer, I love to see candidates who weigh the different possibilities.

When you’re coming up with solutions, take a moment to think about other ways that you could solve the same problem. Could you make trade-offs that would improve the space usage in relation to time or vice versa?

Finally, you should always consider the space and time complexity of every solution you come up with. This gives you an objective way to evaluate which solutions are better than others and helps make a way more informed decision about which solution to choose. If you have multiple solutions that are comparable, discuss with your interviewer and decide collaboratively which one would be the better approach.

## 5. Start with the brute force solution

I already mentioned this in passing in tip 3, but one of the biggest mistakes that people make when trying to solve interview problems is that they immediately try to find the most optimal solution to the problem.

I'll tell you that finding a brute force solution is 1000% better than not finding a solution at all. And if you start by immediately trying to find the optimal solution, it is easy to get stuck and end up without a complete solution by the end of the interview.

While you don't actually have to code it up, I recommend at least mentioning briefly how you could solve the problem with a brute force solution before you go on to try to optimize your solution. This accomplishes two important things:

1. It gives you a fallback plan. If you try to optimize your solution and fail, you can stop after 5 or 10 minutes and just code up your brute force solution. You might still pass the interview. Not all problems have optimal solutions.
2. It helps you clarify the problem. Defining a brute force solution can help you to understand exactly what is involved in coming up with a solution to this problem. That is key. Understanding the problem in this deep way will make it easier to optimize.

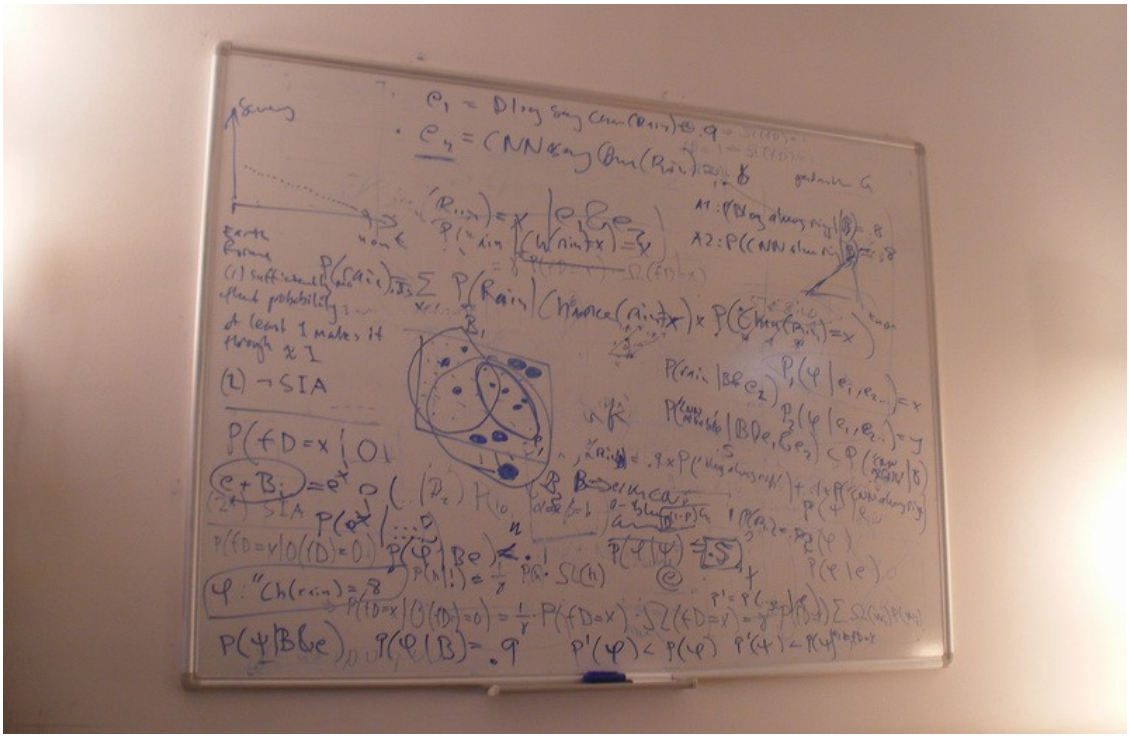
Attempting to immediately find an optimal solution may seem like the right approach because it will save valuable time. However, I find that the confusion that results from taking this approach often wastes a lot more time than you gain.

Starting with a brute force solution gives you clarity and a starting point to make everything else way easier.

the whiteboard that you're going to use is not going to have a copy-paste option. That means that you want to have a good outline of your code before you write it.

Often times, people dive right into writing code as soon as they get asked a problem in their interview. Now it's totally fine if you want to go ahead and define your method upfront, but that should be the extent of the code you write until you've fully worked out the solution. Writing any more code than that is a critical mistake for two reasons.

First, as I said, whiteboards don't have copy-paste functionality. That means that if you want to move lines of code around, you have to either erase and rewrite them or draw arrows going all over the place. You don't really want your whiteboard to look like this:



interviewer. It is easier for them to understand your solution and it's much easier for you to keep track of what is going on. And if you decide to just rewrite stuff instead, you'll be wasting a ton of valuable time.

The other issue with starting to code right off the bat is that it can lock you into a specific way of thinking about the problem. We'll talk about this more in point 7, but this can be incredibly harmful to your interview performance.

Imagine that you see a problem and a solution immediately comes to mind. You start coding it up, but you realize that it is no longer optimal. You are unlikely to want to erase everything and start over. And even if you do, you'll be locked into that mode of thinking.

There are problems where the optimal solution is completely unrelated to the brute force solution, and attempts to optimize the brute force solution will fail. If you wait to start coding, you avoid locking your mind into that one way of viewing the problem.

These concerns are why I always suggest that you fully understand the solution that you want to code up before you write the code. Draw pictures, write pseudo-code, do whatever you need to do to understand the solution. Once you start coding, it should be trivial because you already know exactly what to write.

## 7. Keep the big picture in mind

One of the biggest problems that I see for more experienced

This is a perfect example of losing the forest for the trees. The question they're trying to solve becomes one of "How do I write this loop correctly?" instead of "What purpose is this loop serving in the greater context of my code?"

A perfect example of this is a problem where you try to use the wrong data structure. Let's say that you're storing the values indexed from `1-N` and you decide you want to use a HashMap. You can insert `1 -> val` ue `1, 2 ->` value2, and so on.

But now when you want to iterate over them in order, it is going to be a pain because you have to get all the elements from the HashMap and sort them. However, if you took a step back and looked at what you're actually trying to do, you just want to store the value at each index and iterate through them. An array would be a much easier data structure to use.

Now you may be thinking, "I would never do something stupid like that," but trust me, it happens all the time. Your thought process is non-linear when you're solving problems, so you may have thought you needed a HashMap because of some other line of thinking that you have since abandoned.

This is why it is so critical to stop every so often, especially if you start doing something that seems challenging, and look back at the big picture of what you're trying to do. Whenever you're doing something that seems unnecessarily complicated, look at your end

## 8. Use abstraction to your advantage

I love asking complicated interview problems. If a problem involves several different components, you as the interviewer get such great insights into how a candidate manages their thinking when there is so much to deal with all at once.

The key to successfully solving these problems is to use abstraction. At its core, this means breaking out your code into smaller functions with more specific purposes.

Consider a simple example. Let's say that we wanted to print out a linked list in reverse order. After working through this problem we realize that there is an `O(n)` time and space solution to the problem using a stack (push each element onto the stack as we iterate over the list and then pop each item and print), but we can solve the problem in `O(n)` time and `O(1)` space by reversing the linked list.

Now it would be easy enough to reverse the linked list in our code, but what if we had a function to do that for us? That would make our lives way easier. We just call the function on the linked list, iterate over everything in the list and print it, and then reverse the list again so that we return our input to the original state.

With that logic, we can now isolate the process of reversing a linked list and think about how to do that effectively. While this problem is a very simple example, it is easy to see how this reduces the amount of complexity we have to think about at any given time.

With more complicated problems, I recommend you ask yourself the

assuming those functions already exist. Then you can go back afterwards and implement those functions, now that the rest of your code already works.

This has several advantages:

1. If you run out of time, you still have a basically working code. Abstraction allows you to focus on the overall structure without having to get bogged down in the minutiae. If you have extra time, you can worry about the minutiae, but even if you don't, it's clear to your interviewer that you know what's up.
2. Clarity of thinking and code. They say that a clear desk means a clear mind and the same is true with code. The better you organize your code and break it up into manageable components, the easier it is to think about.

I find that the more involved the problem is, the more valuable it can be to break things up into manageable components.

## 9. Test your code

When we are doing something new, we tend to forget a lot of the things that we already know. We assume that the things that we already know don't apply.

Consider this example: I've been learning how to play guitar in my spare time. I was struggling to make progress, so I asked my teacher for help, and he suggested that I write down some goals about what

prep and guitar practice.

In the same way, I find that many students forget to apply the best practices they know from real-world coding to their interviews. They assume that coding interviews are totally different, so the things they'd do normally don't apply.

One of the things people forget all of the time is to test their interview solution. But would you ever commit code in the real world without testing it thoroughly first?

You test your code because you want to make sure that it's correct and that it does what you think it should do. This is even more important in the stressful environment of an interview because you are more prone to make mistakes.

The key with testing your solutions is to actually go through the code line by line, tracking the values of each variable, and effectively "running" the code. If you just read through the code at a high level, you can very easily miss smaller issues with your code. I recorded a video that demonstrates exactly how to walk through and test your code.

I find that many students stress out about having their code be perfect the first time, and while this is a good aspiration, it rarely happens. It almost never happens in the real world so why would you expect it to happen in the more stressful environment of the interview? However, if you test your code thoroughly, you can fix any bugs and still end up with an A+ solution.

interviewing. You can get more comfortable with the experience and give yourself plenty of opportunities to succeed. An often-recommended strategy is to schedule a ton of interviews with the ones you're least excited about first. That way you get to practice interviewing where you don't really care about the outcome so that you're more prepared when the important interviews come around.

While I think this strategy has merits, there is one fatal flaw: Companies are notoriously bad at giving you any meaningful feedback.

"So," you might say, "Who cares? I can just judge my own performance."

Well yes, that's true, but it can be really hard to judge yourself. You don't know what criteria your interviewer is looking for (just getting an optimal solution to one problem may not cut it). And if you're struggling to succeed, chances are there's something that you're not seeing.

This is why mock interviews, and on top of that ideally working with a coach, are so important. Mock interviews give you the opportunity to get detailed feedback on your performance. The interviewer can also tell you if there are things that you didn't notice

If you're really serious about doing well in your interviews, I also recommend working with a coach on top of the mock interviews. Mock interviews are individual data points. They tell you that on a particular problem at a particular time, you did well or poorly.

performance.

Ultimately, mock interviews give you data points and coaches help you connect the dots. Getting this sort of feedback is the best possible way to accelerate your interview progress.

Time and again, I see people stall out in their interviews. And it's almost always for one of the reasons described above. If you're not making the kind of progress that you want, read this article closely. Identify your problem area(s) and work to correct it. In time, you will be able to refine your interviewing and start getting the calls you want to hear.

---