Gmail rishav ghosh <rishavghosh605@gmail.com>

Don't do another interview until you master these 10 techniques!

1 message

Sam Gavis-Hughson <sam@byte-by-byte.com> Tue, May 21, 2019 at 9:33 PM
Reply-To: reply-bytebybyte.activehosted.43.127.13780@s4.asa1.acemsd3.com
To: Rishav Ghosh <rishavghosh605@gmail.com>

Hey Rishav -

Yesterday I told you about the #1 thing that people screw up in their interviews: They don't know what to do if they get stuck.

In fact, I've found through working with dozens of students that many people don't even know how to approach a new problem. Rather, they simply go into the interview and hope that a solution magically appears to them.

This is NOT a good strategy.

If your interviewer knows what they're doing, they will specifically ask you a question where the solution is nonobvious. That doesn't mean you have to be a genius to be able to solve it, but you do have to have a structured approach.

When you see a problem that you don't recognize, what is your plan? Here are 10 strategies to keep in your back pocket in case you get stuck during your interview.

Find a brute force solution

There's probably one reason above all others that causes candidates to get stuck finding a solution: They try to immediately find an optimal solution.

I've discussed starting with brute force solutions in the past, but it bears repeating. It is very easy to go down different rabbit holes when trying to find an optimal solution and finding a brute force solution always gives you a grounded place to come back to. Even though it may not be a great solution, if you can't find a better solution, it gives you something to fall back on.

The brute force solution also gives you a jumping-off point to optimize your code. With a brute force solution, it is often easy to see places in which you are doing repeated work or otherwise slowing down the execution.

After finding the brute force solution, you can use Gayle Laackmann McDowell's BUD optimization, which stands for bottlenecks, unnecessary work, and duplicated work. By identifying places in your code that are slowing everything else down, you can find ways in which to optimize the code.

For example, if you're looping through an array and for each element, you need to look up that element in another array, you have an `O(N * M)` time complexity, if `N` and `M` are the lengths of the two arrays. However, if you put the second array into a set, you get a lookup time of `O(1)`, so your time complexity improves to `O(N)`.

Finding a brute force solution is incredibly valuable, because it helps you understand the problem better and ensures that you have a solution.

Fully understand the problem

It is shocking to me how many people will start trying to solve an interview problem without really understanding what they're supposed to do.

In fact, when evaluating new coaching clients, I always ask them to simply print a linked list in reverse order. This should be a very simple task, but many people miss the core directive, which is to **print** the list. I often find people trying to return a reversed list or other such things. However, that is NOT what I am asking and can be significantly more difficult.

When problems are more complicated or confusing, I often find that people will spend a few minutes trying to understand the problem, but eventually give up and figure they'll just start solving it. That's a critical mistake, though. How can you solve a problem when you don't know what you're solving?

There are a couple things that you can do if you're struggling to understand a coding interview question. The first is to simply ask more questions of your interviewer. Ask them to show you an example or clarify any specific points on which you're unclear. Don't feel bad about asking a lot of questions. It's their job to make you understand what to do.

You can also work through a few examples. Look at what the inputs and outputs are. What is the function signature going to be? Explicitly defining these parameters can be very helpful in thinking about the problem. For example, if you're writing a recursive function, it's way easier to figure out the base case when you know what the return type should be.

Work through the problem by hand

Not sure how to solve the problem in a coding interview? Well can you solve the problem yourself without writing any code? If you understand the problem thoroughly, you should be able to answer it. And if you can solve it by hand, you can solve it with code.

Let's consider the example of determining whether a string is a palindrome. Say I give you the string "acaramanamaraca". Is it a palindrome? Go ahead and decide.

How did you solve it? Did you compare the first and last letters, then the second and the second-to-last and so on? Did you reverse the string and then compare them? There are several different ways that you can determine if a string is a palindrome by hand.

And now that you've done it by hand, can you write code to do the same thing? In the two example solutions that I described, I basically explained an algorithm that you can use to solve this problem by hand. And that means it's easy to write code to solve it.

The key with this technique is just to be as detailed and specific as possible when solving it by hand. If you say "oh I just know it's a palindrome", then that's not very helpful. How do you know? If you're struggling to think about it, pick a larger input to solve by hand. Something that you can't do in your head.

For example, if I asked you to multiply two numbers, it would be silly to use the example of `3*4`, since presumably you have that memorized from doing multiplication tables in elementary school. A far better example would be `14235 * 2512`. With this, it is very unlikely that you'll be able to do it in your head so you have to write it out. This will help you identify the "algorithm" that you're using.

Brainstorm different data structures and algorithms

Sometimes the easiest way to approach a coding interview question is to take different common solutions and see if any of them apply. We can do exactly this in coding interviews by brainstorming different data structures and algorithms and seeing if any of them are a good fit for our problem.

Let's say that we have a problem that involves strings. For example, maybe we want to all the strings in an array that have a certain prefix. Let's brainstorm:

- Just keep the strings in the array. We iterate through the array and just check the prefix of each string.
- Add all the strings to a hashmap with mappings for each possible prefix. That might work but there are a lot of possibilities.
- We could sort the array of strings. That might make things easier.
- Maybe we could do some sort of tree structure?
- Speaking of trees, what if we used a trie?

From this basic brainstorm, we've come up with multiple valid possibilities. In this case, the problem that we're looking at does have many possible solutions, so that's to be expected. Even if we only had one way to solve the problem, though, this may help elucidate ways in which we can approach it.

The key is that no idea is too dumb when you're brainstorming. You're doing this in front of your interviewer, so the temptation is to try and sound smart, but you can very easily shoot yourself in the foot by doing this. If you filter yourself while brainstorming, you may miss key insights.

Put everything down on the whiteboard and then see if anything jumps out at you. Can you make any clear connections? This technique is great for sparking ideas.

Consider all the information you're given

Lots of times, our interviewers give us clues as to how we should think about a problem. By giving us details about how the input is formatted or any restrictions on the input, they are subtly hinting at how to view the problem.

For example, let's say that we wanted to find a value in a sorted matrix. If we know that the rows and columns of the matrix are in sorted order, that gives us a big clue for how we can solve the problem.

Without that information, we're fairly limited in how we can approach this problem. Our best bet would be to just iterate over every value in the matrix to see if the value we're searching for exists. However, we have a big clue, which is that our matrix is sorted.

Knowing the matrix is sorted, we have algorithms to help us deal with that. There's binary search and we can also consider other properties of our matrix. For example, since both rows and columns are sorted, if we pick an arbitrary cell `[i, j]`, any cell `< i` and `< j` is going to have a value less than cell `[i, j]`.

From the information given, if you're stuck, you can simply take some time to extrapolate everything you can. If the array is sorted, what does that mean? If the input is in tree format, how can we use that to our advantage? Look for these clues.

Simplify the problem

My favorite types of problems to ask as an interviewer are problems that have a lot of moving pieces. These force the interviewee to consider not only each part of the problem, but how all of the different components interact with each other.

As an interviewer, this is great because I get a better idea of an interviewee's mental model. As an interviewee, however, it can be difficult. Whenever you see a problem that has a lot of different stuff going on, a good starting point is to consider a simpler problem.

Let's say that we want to find the path through a matrix with the greatest product. This is a moderately difficult problem, but it is complicated even more if you consider that values in your matrix can be negative. Since multiplying negatives cancel each other out, this problem gets pretty tricky pretty fast.

But here's the thing. The solution is not fundamentally different than solving the problem for only positive numbers. If you get the solution for solely positive values, then it is relatively easy to extend the solution to include negative numbers as well.

Similar to first working through a brute force solution, solving a simpler version of the problem gives you a really good starting point from which to build out your solution. It is often simply a matter of handling additional input values and adding some conditionals. Once you have the framework, this is easy to do.

Break down the problem into subproblems

In addition to simplifying problems, you can also just ignore parts of the problem when you're initially coming up with a solution. This is my favorite technique when people do it right.

When you are asked a problem and are trying to come up with a solution, ask yourself the following question: "Is there any function that, if I had access to it, would make it significantly easier to solve this problem?"

Oftentimes, you will find that there is an obvious function that dramatically simplifies the problem. Consider the example of printing a linked list in reverse order. There should be two obvious functions that would simplify this problem: reverseLinkedList() and printLinkedList(). If we already had these functions, our solution becomes trivial.

```
void printReversedLinkedList(Node linkedList) {
    linkedList = reverseLinkedList(linkedList);
    printLinkedList(linkedList);
    // optionally, we can reverse our linked list again to
    // return it to the original state
}
```

Our code really can't get much simpler than that. And now that we have our top-level function laid out, we can focus on implementing each of these functions individually. This won't necessarily be trivial – reversing a linked list is a bit tricky – but it will be a hell of a lot easier than what we were doing before.

The key with this technique is that by breaking down the problem into smaller components, it is much easier to see how to solve the problem. You can start by just assuming that you have a function that does X. Now you only have to worry about doing all of the other things. This abstraction makes it much easier to reason about the problem.

And this technique also has an ancillary benefit. If you don't finish writing all of the code, it is still clear that you know what you're doing. If you choose to use a helper function, you should implement that last. By implementing the core code first, it is clear you know what's up, and if you run out of time to implement the helper function, chances are that was something that would be pretty easy to do anyway. This is a great way to compensate for lack of time/speed in an interview.

Take a step back

Going down a rabbit hole is one of the easiest ways to get stuck in a coding interview. However, it's not always obvious in the moment that you've even gone down a rabbit hole in the first place. That's why I recommend that whenever you find yourself starting to get stuck in a coding interview, you take a moment to step back and look at the big picture.

Let's say that we realize that we need a function that stores value mappings. For example we have something like int -> string. Our initial solution might be to create a HashMap to store these values. That would seem to be the logical thing.

However, we later discover that all of the integer keys are each of the values from 0-n and we are going to want to be able to iterate over the keys in order. Now our HashMap doesn't seem so great, right?

However, I can't tell you how many people I see who will just take what they already have and try and force it to work. They're trying to put a square peg in a round hole. You can do it, but it's a lot of work.

First they have to get a list of all of the keys in the HashMap, then they have to sort it, and then they have to iterate over it. And that's not to mention the extra work of putting everything into a HashMap in the first place.

If they just used an array, it would be so much easier.

However, this is not always obvious. We tend to think about problems in an iterative way, building up to a solution. That means that if we don't stop to think carefully about what we're doing, we may end up doing a lot of unnecessary work.

Taking a step back is simply taking a moment to think about what we're doing at a high level and confirming that it is, in fact, the best way that we could be accomplishing that task. You may be surprised how often you don't realize that there is a vastly more efficient approach.

Collaborate with your interviewer

Many people see interviewing as a battle. It's you versus the interviewer. You're trying to get the job and they're trying to take it away from you. You want to succeed and they want you to fail.

This is one of the most destructive things that you can think in your interview. This is completely false and it can change the tone of the interview to be combative, which isn't fun for anyone.

The truth is that your interviewer wants you to succeed. Their goal is simply to make sure that you meet the standards that the company has. If they're being hard on you it's because they're trying to make absolutely sure.

When you're stuck on a problem, you can get feedback from your interviewer. The more you talk out loud in your interview, the more they will already know what's going on, too, so the more they will be able to help you.

One of my favorite phrases to use in an interview is simply, "does that sound good to you?" When I come up with a solution and I'm not sure if it's the best, I'll ask that question, which gives my interviewer two options.

1. If I've found a decent solution, they can say yes and suggest that I start coding it up. This way I don't waste any unnecessary time trying to come up with an even better solution.
2. If they're not happy with my solution, they can say no and suggest that I think about it a little more. This is helpful too because, without really giving a hint at all, they are saying that I'm not quite there yet.

Granted, not all interviewers are going to be quite so helpful, but there is no harm in asking and it can make your life a lot easier.

Ask for help

I've included this last because it's definitely a last resort. Most interviewers will mark you down for needing a hint in your interview. However, if you're stuck and not making progress, this is better than just twiddling your thumbs. And it doesn't necessarily disqualify you from getting an offer.

When you're really stuck and not able to proceed, you can just be upfront with your interviewer. Tell them that you're a little stuck and ask if they can give you a hint. Chances are, they'll offer you up something that will help you keep moving in the right direction.

The worst thing that you can do in your interview is to give up rather than asking for help. In the real world, there are plenty of opportunities to get help from others, but no company wants to have a quitter on their team. If you just give up when faced with a hard problem, that's definitely not what they want.

You're not going to use all of these strategies in every interview. They're simply not necessary.

Sometimes you'll see a really easy problem and you won't need to use any of them. Sometimes you'll get a really hard problem and have to combine several strategies.

Regardless of the specific interview question, though, knowing these strategies will make you bulletproof against even the toughest questions.

So I want you to do something right now. **Pick one of these strategies and try it out for yourself right now.** We went over a ton of stuff today so start with just picking one. I guarantee you'll see a big difference.

Best,
Sam
p.s. Do you get stuck on recursive problems? Next week I'll be sharing more info on my recursion masterclass *Coding Interview Mastery: Recursion*, so keep an eye out for that!