Episode 029: 7 Steps to Solving Code Challenges

Welcome back to La Vie En Code, a podcast dedicated to the self-educated web developer. I'm your host, Nicole Archambault. So hey, everyone, especially my new listeners. It's been a little while. If you're just tuning in, it's my first time back, on the podcast after about probably about a six or seven-week hiatus. I'm fine. I think everybody needs a little break once in a while. I am back in full force now. I know I've said that before, but I really think that I am at this point. It's kind of a big deal for me. Sitting in this chair again in front of the podcast, my life is turning into a bit of a wild story and I'm looking forward to sharing these last six months on a future episode, but it's definitely helped to form my identity as a person, a web developer and an entrepreneur, but I know that's kind of vague, but I am OK. The important part is I'm OK. I'm here with some more great information for you guys.

I have a really great topic to go into with you all today because it's such a big issue in programming overall. Folks who've been listening in for a while know that I'm very much focused on the experience of the self-taught web developer, but this is also a topic that seriously under-addressed even in a lot of computer science and coding boot camp programs. A traditional education or boot camp environment, they just don't touch it. The topic is problem solving now for the purposes of this episode in particular. I'm going to be referring to problem-solving in the code challenge context only and I won't be focusing too much on like debugging. I think that's probably good for a future episode and I have some specific tips actually for debugging that deserve their own airtime, but now this week, we're going to be focusing in on the types of algorithmic challenges that you're going to be finding on popular e-learning platforms like Treehouse and Free Code Camp, as well as in developing your own algorithms that work for your own projects and of course, you'll find these challenges in the form of take-home coding challenges to an even whiteboard challenges and job screening or interview setting, but in the past three years since I began learning to code in 2015. I've heard from so many students who were really intimidated by the prospect of solving a code challenge from scratch and that really resonates with me because it was a tough thing for me to overcome as well. Whether it is staring at a Free Code Camp code challenge or a project where you have to write code the passes tests in order to move on, or you just know that you need to write a function that does something for your own project. It can be stressful. I definitely struggled and I still struggle. That doesn't go away. To apply some of the approaches that we'll be talking about today, but what I learned is that when I do apply them, they always work, and it makes the process a lot easier and less stressful.

So today, I'm going to help you approach more complex, but still beginner level coding challenges with greater confidence using a little 7 step approach. That isn't unique to me. I've picked it up online and at one point I Googled how to solve problems early on in my education, fortunately, and this is kind of what has worked best for me. It's led to success. So, there's no code specific advice here. So no language and you can use this approach to solve lots of different types of problems even in your own life, but by the end of the episode, you'll basically have a new tool in your toolbox and be able to see complex

problems for what they really are, which is lots of smaller problems piled one on top of the other. So, you ready? Let's get into the meat and bones. All right, let's go.

Let's start first with a basic question that's often overlooked. What even is problem-solving? I guess it's kind of self-explanatory, but in the context of programming, what really are we going for? So problem-solving is the process of finding solutions to difficult or complex issues and specific to programming. You're going to use problem solving when you're facing a code challenge as well as in a multitude of other instances where you'll need to find a solution to a problem. Something is effectively not happening that needs to happen, and you're going to make it happen by issuing step by step commands in your language of choice, whatever it may be. Those step by step commands uses together in order to accomplish this goal are called algorithms. Algorithms, automate the process of accomplishing tasks and by extension, solving problems by making things happen. I'm going to be blunt with you guys, which is probably the best way to approach this. If you don't know how to solve a problem, no matter whether it be complex or basic, you'll very likely get overwhelmed and probably drop web development.

Everything in this industry is a problem that needs to be solved at the top level. Every application solves the problem for the end-user. Games are designed to meet the engagement needs of users and organizational or productivity apps meet the productive and the stress management needs of the end-user. Their life is made a little easier because of the app, but those are just some examples of problem-solving as a whole. We're going to focus on code challenges specifically, but that should give you some context of how prevalent problem solving is in this industry and how important it is. So if you're just starting your web development career or if you're still trying to figure out if you want to transition careers into web development, just know that your education isn't going to cruise for very long, especially if you don't learn how to manage these code challenges early on. Even if it's easy at first, it doesn't stay that way. We all know that if we've gone that route, but the bottom line is that problem solving is super important and will never go away. So you might as well learn it as early as possible.

Where are you going to encounter these problems? As I had said before, you'll find these throughout the course of your e-learning process, and some introduce code challenges that require high-level problem-solving skills. Free Code Camp is perhaps best known for its algorithm challenges, which I believe are part of the front end and the back-end certification tracks. You're going to need to write code or specific functions that output results that are expected. So, for instance, you write a function called multiply, which is intended to multiply two numbers together that the user inputs. So if you were to pass in the parameters two and three, it outputs six. Five and three, it would output fifteen and so on and so forth. The Free Code Camp platform expects that with this code challenge if you input two numbers that it specifies, you'll get a certain value as the output and once it passes those tests, then you can move on. As you can expect, there are lots of ways to solve any particular problem, which is why these types of tests have worked really well for when you're developing. Different new coders are going to have different approaches for sure. If you're not already, you'll be using lots of testing once you get into a more advanced developer role.

Treehouse, which I use for the majority of my education, also has similar built-In programming challenges, and they're mostly designed to apply the skills that you just

learned in their course videos. Each lesson is made up of videos and challenges. However, there's are often a lot simpler and they give far more helpful tips. They can even mention the specific areas that you need to fix. If you are not referencing something in an easy problem, it will draw your attention to this specific area where you might be getting confused, which is super helpful, obviously, but the code challenges are nice because they generally come right after you learn a particular topic. You'll learn a new topic and then you use it immediately after. The hope is that you can use it, you know that you're able to and that's what these code challenges are intended to test and if you struggle in this area, then this episode is definitely for you. Last thing before we actually get into the steps, because these are important topics to go over. One last question. Why are code challenges even so difficult?

Well, it can be overwhelming when you're asked to do something for the first time. Plain and simple. You think about that the first time you've ever done anything. It's going to be complex. and in these code challenges, virtually everything you're going to be experiencing is new in the sense that you have to actually use it. Once you've used it several times, it gets more comfortable to recall, but often you know the desired outcome. You know the problem that you want to solve, but the steps in between are the mystery that you need to solve first. There's lots of room for stumbling where it makes it difficult. You might not have the required skills yet or even know where to find them or actually even understand the problem, to begin with. I overcomplicate everything, too, and we'll talk about that. So by deductive reasoning, the process gets easier the more challenges you encounter. You'll only get better at solving code challenges by doing more code challenges. I, for instance, have a very complex mind and when I see problems, I see equal complexity and this over complicates my process and lots of wires get crossed.

Have you ever started working on a project and rather than just solving one small part at a time, solving one small problem at a time, you kind of go in trying to write the functions that, you know, that are kind of easy and doing all sorts of other scattered bullshit. That's pretty much the problem-solving approach that I had for the majority of the beginning of my education and how I started most of my code challenges for the first year and a half and sometimes it works out and oftentimes it doesn't. Usually depending on how much you know and how good you are with applying such a scattered approach. Your brain would kind of have to work very differently, but generally speaking, though, you need some actual steps to finish one algorithm and then move on to the next.

Until you build an actual solution and a program that runs as expected, but even if you get overwhelmed, there is hope. Rather than seeing just the bigger problem and getting overwhelmed or scared off, I've had great success in training my brain to see a whole slew of smaller problems. I've probably been told a ton of times over the course of my life how to solve a problem. I don't think I ever really understood anything in life until I started teaching myself web development. That might seem a little dramatic, but it's true. Web development, teaching myself taught me everything that I really need to know. You might identify with that. The experience of self-education is basically figuring out how to maximize your own education, while just about every day you're realizing that you kind of already knew a lot of these things. Someone told you at some point in your life that you can't and shouldn't overcomplicate things. Keep it simple, but if you're like me, keeping it simple might feel weird because of these things, these problems, they feel complex. Most students I work with are able to solve these problems if they just take a deep breath and go through the process step by step.

So, seven steps for effective problem-solving. Let's take a look at my method for approaching problems, I worked with as kind of result of being personally the absolute worst at approaching problems. I teach this process in my upcoming course 30 days to web development, which is a kind of a pre-learning prep course for aspiring and current self-taught web developers. Step one is to read and identify the problem, read the problem entirely at least twice, and then read it again to ensure that you understand it fully, but don't read it a fourth time that's overkill. It's possible, too, that you may not be provided with a question in actually written format. And if the problem is posed to you, verbally, ask the person posing it to write it down. Problems communicated verbally are generally also communicated poorly. It's more open for miscommunication and plus seeing a problem written in front of you will greatly help your brain make the needed neural connections required to solve the problem. So, if the problem is something you're facing while working on your own project, just stop and take a minute to write down what is wrong. What were you trying or wanting to do? What did you want or expect to happen? Then what actually happened? Don't worry about how to solve or fix the problem just yet. We'll get to that through this process but once you've read or identified the problem fully, you can move on to step two.

Step two is to break down the problem. We need to break down the problem into the smallest possible problems and the more complex that the top-level problem is, the further it can be broken down. Usually, I mind map them, just so I can visualize what the smaller problems are or just draw it out on a whiteboard. So, when all your problems are broken down, figure out which ones you want to tackle first and tied into the rest. We've broken down several problems now, so you should be able to work through this step and come to an approach and it does not matter if your approach is perfect. In fact, it probably won't be. It's going to be a little bit nerve-wracking and disastrous at first. Additionally, this is a time for you to identify all of your less obvious factors. What are your variables? What's your input? What's your required output? It's so simple to overlook, those really basic things about, you know, writing the code before you actually write it in. It can be tricky to answer those questions, but they are critical to answer before moving forward.

Failing to identify your variables and your input, for example, will leave you unable to complete the next step. Identifying your cases is also really important as you'll soon learn about control structures and how they direct the flow of the problem and this requires asking yourself some more questions. For example, if your function accepts parameters, what if the user inputs something other than what's expected? Is your code going to blow up? Once you've laid out those smaller problems, identified an approach that includes factoring in your variables and your input and output, you're ready to move on to step three.

Step 3 is to pseudocode your solution manually. You're going to pseudocode. Your solution manually in pseudocode is just solving problems, using plain communicative language, you know, whichever you speak instead of programming syntax. We're worried about what the code is doing as opposed to the syntax of how it's actually written and we're not worrying about, you know, errors being thrown in this step. Armed with the approach, you're going to write out, your individual steps in these steps will be condensed in the next step, but for right now, they just need to work. They need to get you to your end goal and provide as much detail as you need to These steps will be rough, though, and that's OK. They'll be like a rough draft. For example, if you wrote an algorithm to eat an apple, you could initially write out something like step 1 check to see if the apple even exists. That's very important and often overlooked. Step two, take one bite of an apple and then step three if there's more apple left, you take another bite and then step four would be

if there are no more bites, then stop. Run the program or exit the program, rather. If your problem requires you to consider something like chewing and swallowing in order to ensure that your mouth isn't full, for example, some more steps in the process for more detail, you may need to add additional steps to factor that in. Again, you'll be as detailed as the problem requires. So, once you have your steps written out in pseudocode, you can move on to step 4.

In step four is to automate your pseudocode. You're not going to wait. It might seem a little counterintuitive, but you're going to automate your pseudocode and it's an important step because it's going to tighten up your solution. Automating is a form of optimizing, but we're not going to wait until we actually write code. In this step, we're going to identify the steps themselves that can be condensed as well and if you are wondering why we do this before coding, I mean, don't you want to see how the code, you know, actually runs, see the code first and see what it does, but the key is that we don't want to wait until you write actual code to optimize it because it can be difficult to remember what exactly you're trying to optimize. That was a huge problem for me, but generally speaking, never write code before figuring out what you're actually trying to do with that code in just regular communicative language first. This step is also really important, especially in my opinion, because one of the main principles of good programming is that you don't want to repeat yourself.

You want to keep your algorithms code as “dry” as possible. In other words, don't repeat yourself. A wet code on the contrast has a lot of repetition. Which wastes valuable memory and it takes longer to return the output. To identify the places in your pseudocode where you can tighten things up and smooth out your procedure rather than repeating steps. We could add code that tells the computer to go back to this step and repeat it until some other piece of code is satisfied, some condition is met and yes, that counts as automating and you should begin looking for opportunities to automate whenever possible as early as possible. Once you've identified those areas to optimize in your code, you'll give it a shot, in your pseudocode, rather, but being able to identify patterns in it is big here and it'll help you out in the long run. So, with your optimized pseudocode in hand, you can continue on to step five.

In step five is to convert that pseudocode, the optimized pseudocode into programming code. So, we're going to take it from a communication language into a programing language and you're finally going to get to code your solution. By the time that you get to step 5, it means that you've already made it more than halfway to your final code solution. So, this step will be so much easier now that you've actually planned out your pseudocode. You can put it in comments and then flesh it out into your code, which is a great thing to do. That's the way that I approach a problem and, you know, many new web developers just dove right into this point from the beginning. They didn't go through steps 1 through 4 and usually, though, quickly realize that they're drowning, but you won't drown. You'll be swimming along because you've already done most of the hard work. After you've written code, actual programming code that passes the computer. So, no errors. It doesn't have to be perfect, but at least it passes. Then you can move on to step 6.

Step 6 is to test your solution and revise it. So, in the nature of tech, you'll get instant feedback from the computer telling you if something doesn't work, usually in the form of our friend's errors. In the beginning, you're going to get a lot of errors. A lot, a lot, and the good thing is errors are a wonderful learning tool and they'll be an integral part of your self-education. Trust me. But for this step, the goal is to try a few different inputs where we know what the output should be. Plug in a bunch of different values and check out and see

if your code still works. If the expected results aren't yielded, though, you're going to back up and fix it and you want to make sure that you're thinking about as many of those cases and edge cases that you need to consider. Make sure that the input is something that your program can handle and once you get consistent results, then you're going to move on to step 7.

This is the final step. Step 7 is to improve and optimize your solution. So now that you have code, actual programming code that works to solve your problem with a variety of different user inputs, you should look at where you can improve not only your process but your code itself. There may be better options for what you're trying to do in terms of using specific programing approaches or as we call them, in the industry design patterns but getting the problems solved in some way, shape or form is your first goal in this entire problem-solving process. When you get to the point where you have great code, first off, I want to see it and also make friends with other developers and ask them what they do differently. Developers, in particular, love, optimizing things and offering input based on their own approaches that they know. So, you'll probably learn a lot of different approaches and perspectives on the same problem.

Those are the exact seven steps that I use to begin and then go through a project. If I don't use them, then I'm bound to hit a wall somewhere. I mean, I'm not exempt from that. I've just learned that in order to have the smoothest possible ride and not only that but to get maximum enjoyment out of the coding process, it's just better to address all this stuff before I Leroy Jenkins into a project and wipe everybody, myself included. Bonus points if you get that reference, but if this process helps you or you just really enjoyed the episode, which I hope you did.

I would love to hear your feedback and I would also really appreciate it if you leave a happy little review on your podcast, Player of Choice. The La Vie En Code podcast is available on iTunes Google Play, SoundCloud, and Stitcher and actually to hopefully be up on Spotify soon as well. You can also follow my shenanigans in the world of coding on Twitter at Lavie_encode or on the Facebook page at Lavieencode.net/Facebook.

Next week I have an episode that should help a lot of people if you're familiar with the web developer roadmap. You know how confusing it can look at first and if you're not familiar with it. Don't worry. It's a visualized road map created by our fellow web developer, Kamran Ahmed. The roadmap itself is a flow chart of the common past that web developers take in the industry in the skills they learn within those subsets of web development. So next week, I'm going to dissect it for you folks in a way that simple and easy to understand, which I think everybody appreciates. So, you'll want to be sure to subscribe to the podcast. To be sure that you don't miss it and one last note. 30 Days of Web Development is very much still happening, and this is my prep course. As I mentioned before, it turns out making an online course is really hard, especially when you're working pretty much solo first off and, working through real-life stuff, too. As I had mentioned at the beginning of the episode. So it is very much still happening, though and you can check out the status, which I'm trying to keep updated at this point. I'm really gunning for June at this point. It's just hilarious to me that it has taken so long to get out, but you can check out the actual Web site and see what's in the course. So the curriculum is actually posted. You can see what topics are going to be covered over the four weeks of the course, but the Web site is up at 30 days to Web development.com. You can sign up for e-mail notifications there as well. When the code course actually does go live, you'll get notifications and I'm going to be offering a really good discount for people that are on the email list.

Thank you so much again for spending your valuable time here with me today. Until next time, my friends. Peace, love, and code.