**Task 3**

**Solve a problem:**

1. **Read the problem at least three times:**

You can’t solve a problem you don’t understand. There is a difference between the problem and the problem you think you are solving. It’s easy to start reading the first few lines in a problem and assume the rest of it because it’s similar to something you’ve seen in the past

1. **Understand the problem.**

When given a coding problem in an interview, it’s tempting to rush into coding. This is hard to avoid, especially if you have a time limit.

However, try to resist this urge. Make sure you actually understand the problem before you get started with solving it.

This first step is vital as we often don’t take the time to fully understand the problem. When you don’t fully understand the problem, you’ll have a much harder time solving it.

* **What are the inputs?**

What kinds of inputs will go into this problem? In this example, the inputs are the arguments that our function will take.

Just from reading the problem description so far, we know that the inputs will be numbers. But to be more specific about what the inputs will be, we can ask:

Will the inputs always be just two numbers? What should happen if our function receives as input three numbers?

* **What are the outputs?**

What will this function return? In this case, the output will be one number that is the result of the two number inputs. Make sure you understand what your outputs will be.

* **What the program show if user doesn't enter the right or desired input?**
* **Create simple examples, then create more complex ones.**

1. **Devise a plan.**

* Next, devise a plan for how you’ll solve the problem. As you devise a plan, write it out in pseudocode.
* **Pseudocode**

It is a plain language description of the steps in an algorithm. In other words, your pseudocode is your step-by-step plan for how to solve the problem.

Now you have your step-by-step plan to solve the problem.

For more complex problems, professor Evans notes, “Consider systematically how a human solves the problem.” That is, forget about how your code might solve the problem for a moment, and think about how you would solve it as a human. This can help you see the steps more clearly.

1. **Carry out the plan.**

The next step in the problem-solving strategy is to solve the problem. Using your pseudocode as your guide, write out your actual code.

Professor Evans suggests focusing on a simple, mechanical solution. The easier and simpler your solution is, the more likely you can program it correctly.

First, just get out your simple, mechanical solution.

**What if you can’t solve the entire problem? What if there's a part of it you still don't know how to solve?**

Colt Steele gives great advice here: If you can’t solve part of the problem, ignore that hard part that’s tripping you up. Instead, focus on everything else that you can start writing.

Temporarily ignore that difficult part of the problem you don’t quite understand and write out the other parts. Once this is done, come back to the harder part.

This allows you to get at least some of the problem finished. And often, you’ll realize how to tackle that harder part of the problem once you come back to it.

1. **Look back.**

Once your solution is working, take the time to reflect on it and figure out how to make improvements. This might be the time you refactor your solution into a more efficient one.

As you look at your work, here are some questions Colt Steele suggests you ask yourself to figure out how you can improve your solution:

* **Can you derive the result differently? What other approaches are there that are viable?**
* **Can you understand it at a glance? Does it make sense?**
* **Can you use the result or method for some other problem?**
* **Can you improve the performance of your solution?**
* **Can you think of other ways to refactor?**
* **How have other people solved this problem?**

**search on google(different ways, techniques etc etc):**

If you want to find the right answer to a problem you have, you need to know how to ask the right question. For most people this is the most difficult thing to do – asking the right question without beating around the bush.

**Tricks:**

* Use the Keyword related: to find similar websites.
* Use the Keyword search: to search within a website.
* Use the Keyword ... to search within a time frame.

Developer 1992...1999

* Use the Keyword - to exclude a word.
* Use the keyword \* to replace missing words.
* If you're looking for a definition of a word, use the keyword define: to find a definition.
* You can search for an exact phrase using double quotation marks " "
* Use site: to search for a particular website or content.

**building logic:**

**7 Tips for Enhancing Logic Building Skills**

* **Practice writing a lot of code**

Practising is vital when trying to improve logic building skills. Solving problems every day and rigorously challenging oneself is a great way to become capable of building superior logic. Programmers must practise working with different algorithms extensively and thus make way for a deeper understanding of program logic.

Building logic over and over again allows developers to build logic more effectively over time. Programmers must keep on vigorously coding or writing solutions to problems and hone their application of logic on a daily basis. To be a good programmer, one needs to write hours and hours of code, practising the same problems and algorithms over and over again.

* **Check solutions by other people**

Checking for solutions is probably the most important method of learning how to handle programming challenges. Especially when it comes to logic, solutions provided in communities and by other programmers help explain how the logic is built and why.

The “why” is very important here as building superior logic is not possible without a clear understanding of how a program’s logic functions and affects the programming. One must not shy away from looking at resources, books or GitHub to search for solutions when stuck at a certain problem.

* **Use a pen and paper to work out solutions:**

It is a great idea to pen down solutions, models and strategies. Humans tend to remember things they write down much more efficiently. Also, jotting down strategies and solutions is a great way to understand how they function and can be implemented.

Using computers to program logic is great, but working on logic using pen and paper allows much more advanced insight. Breaking down solutions and thinking about them is much easier when programmers write them down.

* **Keep learning new things:**

Programmers must expand their knowledge and logic building skills by tackling new topics and challenges. Delving into unknown territory is highly suggested and one can always look up solutions from other places if he or she feels confused or stuck.

Even if one is not completely confident about the current topic he or she is at, it is advised that he or she practises as much as possible and then move on to other topics, eventually coming back to the older topics later on.

* **Be consistent:**

Programmers must be consistent. This is applicable to programming in general and not just logic building. One must practise building solutions to problems daily. This does not just mean randomly selecting problems and trying to solve them but requires a consistent step-by-step path where every topic is covered extensively and dozens of problems of the same nature are practised. One must be consistent with the field of interest as well as the topics of interest.

A good study plan also involves using selective resources and not jumping from resources or study materials randomly. This is a waste of time and also affects the continuity of the process. Jumping from course to course confuses programmers and one must be really careful when choosing the paths to follow as it is advised to see them through or follow them till the end.

Learning from a single source or solving problems from a single set increases productivity and allows programmers to progress effectively. Being consistent also means that programmers should not keep any gap between practising. Programming for a few days and then taking the next few days off is a bad idea.

* **Face problems head-on:**

When practising solutions or dealing with challenges, programmers will definitely encounter confusing topics or problems which are very complex. One must not give up or lose confidence in himself or herself when facing this kind of situation.

Programmers are meant to face difficulties when tackling advanced programs but this is supposed to be taken as a learning opportunity instead of a hurdle in one’s path. Facing problems as they come is the way to get better at building logic architectures.

* **Don’t lose motivation**

Facing problems when building logic should not be a demotivating factor. Logic is important and key to programming effectively. So, no matter what, one has to get at building logic regardless of the reason.

Acquiring logic building skills is vital to a successful career as a programmer, thus requiring programmers to get better at it by default. This must not be seen as a personal shortcoming but a requirement that must be fulfilled. Without enough experience or practice, no one is supposed to be great at logic building skills.

Learning from others and their solutions while practising daily is the only way any programmer can get better at building logic. Programming productively daily is required out of everyone and is a determining factor in logic building skills and a developer’s total skills.

**start a solution:**

“The biggest mistake I see new programmers make is focusing on learning syntax instead of learning how to solve problems.” — V. Anton Spraul

1. **Understand**

***“If you can’t explain something in simple terms, you don’t understand it.” — Richard Feynman***

1. **Plan**

Don’t dive right into solving without a plan (and somehow hope you can muddle your way through). Plan your solution!

Nothing can help you if you can’t write down the exact steps.

In programming, this means don’t start hacking straight away. Give your brain time to analyze the problem and process the information.

1. **Divide**

* Pay attention. This is the most important step of all.
* Do not try to solve one big problem. You will cry.
* Instead, break it into sub-problems. These sub-problems are much easier to solve.
* Then, solve each sub-problem one by one. Begin with the simplest. Simplest means you know the answer (or are closer to that answer).
* Once you solved every sub-problem, connect the dots.

1. **Stuck?**

By now, you’re probably sitting there thinking “Hey Richard... That’s cool and all, but what if I’m stuck and can’t even solve a sub-problem??”

First off, take a deep breath. Second, that’s fair.

Don’t worry though, friend. This happens to everyone!

The difference is the best programmers/problem-solvers are more curious about bugs/errors than irritated.

1. **Practice**

Don’t expect to be great after just one week. If you want to be a good problem-solver, solve a lot of problems!

Practice. Practice. Practice. It’ll only be a matter of time before you recognize that “this problem could easily be solved with <insert concept here>.”