The confidences of a Coder: Finding bugs must rely on both hunch and rules

Inspiration and rules will help you become one of the most outstanding Coder of any working team, as long as you know how to promote both.

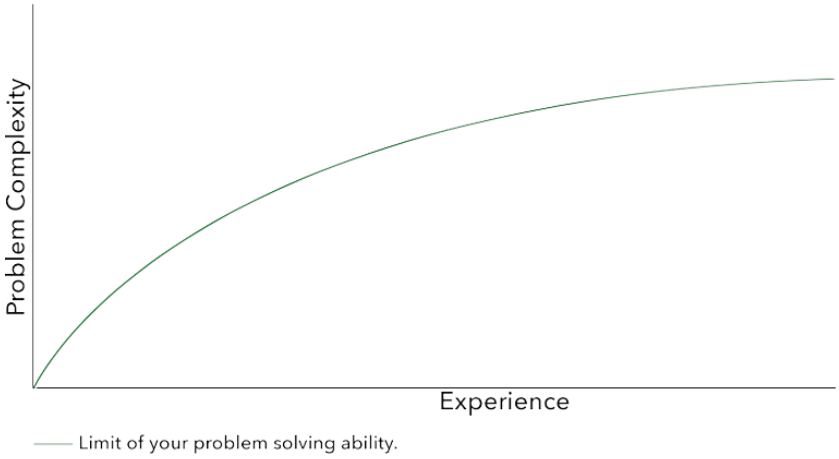
I started writing the first lines of code nearly 32 years ago - when I was 6 years old. I developed strong coding skills. I can understand all problems and immediately know how to solve them, just intuitively.

Over time, I earn my living by "walking code", I feel like a celebrity, a real "rock star coder". I discovered and fixed errors (fix bugs) faster than all colleagues. The team leader started to give me the most difficult tasks and the most difficult bugs. They call me "witch".

However, following intuition can't take me further. I have made great strides but then stopped and nothing stood out. The ability to write code also cannot push me forward.

Trouble believing in hunches

Unfortunately, intuition (or hunch) is a cognitive-based technique to solve problems that cannot be used to make a good evaluation standard. When you rely solely on intuition and aptitude, you will encounter a curve that looks like this:



The relationship between the complexity of the problem (Experience Complexity), Experience (Experience) and the limits of each person's ability to solve problems

Surely you can choose to accept the limit and only solve the problems below this line. Although it seems to make a "rock star coder" like you enjoy, it will start to limit your growth and career quickly. In addition, this choice is not interesting at all.

When I pushed myself further in my career, I began to feel difficult and could not rely on hunches anymore - I realized it was a chaotic direction. I am no longer a child who is eager to learn when in a new environment.

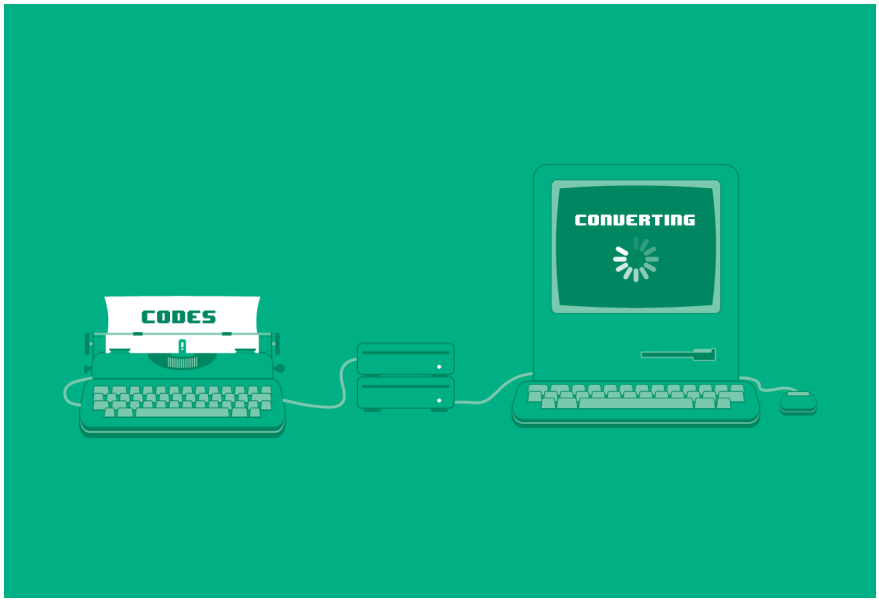
I know that in the end, I will face more intelligent and talented people than me (My illusion of greatness is true. I am not a genius).

But when I look around, I realize that some people beat me not by using "super intellect" or something that belongs to the "innate" code ability. They only have a secret "weapon" that I completely lack: **rules.**

This indicates that a persistent, methodical, repetitive approach to the cognitive and problem-solving process will eventually produce results that are much larger than anything called "innate" or gifted - that you have probably developed.

Equip yourself with problem-solving skills

No matter who you are, how passionate you are or the natural ability of Coding, in the end, you will encounter a certain limit. However, with some of the following techniques, I'm sure you'll improve your problem-solving skills greatly.



Suppose, you have a debugger. You ran it, "Google" tried the result and you could not detect the error.

Continue, assuming if the problem has been indicated by someone, you need to reproduce the error. If not, you need to create the error first. Then compare the context and environment where the error you are trying to repeat is. Begin eliminating all differences, one by one until you repeat the error successfully.

RTFM (Read The File Manual)

Quickly read the document, you are wrong!

Read it really - more than once if it feels necessary. Don't just skim through to find things you can copy - paste and then hopefully they work with your system.

The problem is that you want to read the answer quickly. You want to quickly get results. You are not willing to put your energy to work. So slow down, breathe, drink a cup of coffee and read the relevant documents carefully.

Without documentation, build it yourself and share it with others after you've solved the problem.

Check hypotheses

If you expect something to happen and as a result, it is because you have made a false hypothesis at some stage in the process. So try to test many hypotheses and prove the right one.

Let's start with the simplest hypotheses that can be tested quickly. Is the server really working? Is the network connected yet? Is everything spelled correctly? Are all semicolons and parentheses in place?

If you don't start with simple things, but in the end one of them is exactly what you need to find, you can be "crazy" because it took too much time. So, don't complicate matters.

Disassemble (disassembly) and Assemble (assembly)

Remove the components of the solution until it works again, then turn them back to their original positions to find the broken element.

This may seem bland and boring but this is one of the most effective and disciplined ways to detect the cause of errors in your code. However, be sure to make backups before starting.

If you fall into a situation where you don't know how to assemble the code back to its original order, this sign indicates you're facing a more serious problem: you don't understand the codebase you're working on. Now, the best way is to ask someone to explain or not, you have to spend the whole night learning about it and how this code works.

Remove variables (Variable)

Anything that changes from the first attempt to the next attempt should also be set to static while you are debugging.

This is how Test Driven Development (TDD) works. If you are using TDD, you should have some mock objects in the removal process.

Mock objects are "pretending" objects that can mimic the behavior of real objects in a controlled way. A particular programmer can create a mock to test the behavior of other objects in the same way that a car designer uses a doll in crash tests to stimulate his behavior. people under the impact of vehicle collisions.

If TDD is not implemented, you need to assume any variable to detect an error in the event that other factors remain unchanged.

Note : If you assume an object and the bug suddenly disappears, then it is more likely that the error is in the object you assumed.

Use "Saff Squeeze" technique

This is a well-known and popular technique by Kent Back, developed from the above two ideas.

Kent Back describes the following: *"To find out what is wrong, start testing at the system level and continue like that until you conduct a test with the smallest possibility that will appear wrong."* .

So instead of disassembling mock or code, simply add the functions you are checking into the test itself, then reduce the level of assertions (assertions) until the error disappears.

This is very helpful in helping you perform specific and smaller tests.

After fixing the error, repeat it and correct it again

Never ignore an error until you fully understand how to fix it. You should repeat the error and continue editing.

Don't feel stressed because if you fix a bug and you're not sure exactly why or how you fixed it, the error will likely appear at another time you didn't expect.

What is aptitude?

You've learned important debugging techniques already and the problem is always using them first instead of relying on your aptitude? No, absolutely not!

If you have a strong hunch about the problem and can test it very quickly, do this first. If the error is below the blue line in the chart above, the ability to rely on intuition will help you find the fastest solution.

Once you have tried it based on hunch, but find the wrong error, apply the rules that I mentioned.

Inspiration and rules will help you become one of the most outstanding Coder of any working team, as long as you know how to promote both.