How To Code It?

v2017.12.8

Attila Egri-Nagy

Here is an adaptation of the classic problem solving method[1] to computer programming. It applies to coding exercise problems in (functional) programming languages with a REPL (read-eval-print-loop). Problem solving is a messy business, asking these questions may help to navigate through the process.

Understanding the problem

No point in writing code without knowing what we want to achieve.

What is the input? What is the output? What are the types of these data items? What is the connection between input and output?
Think about a few example cases. Use pen and paper, draw if necessary. Introduce suitable notation. Notation on the paper can be used for names in the code later.

Making a plan

Decomposing the problem into smaller and simpler tasks.

If you are a beginner, play in the REPL! Explore the data types of the input and output and use functions that work with them. First just play freely, try many related functions, even without any obvious connection to the problem. This is for refreshing memory, and for simple problems, using an existing function might turn out to be a solution.

Trying to solve a problem in one go is a sign of lacking a plan. Making a plan is identifying the steps needed, i.e. decomposing the problems into simpler ones. *What are the steps of data transformations? Can we combine existing functions to do work? Can we specialize a function to fit the task? Do we process collections? How to process an element? Do we need to extract the required data items from the input?* If needed, by wishful thinking, come up with imaginary function specifications that would make solving the problems easier. *If we had a function that computes x, and another that computes y, would it be easier to solve the problem?*
If the problem is too difficult, try solving a similar, but easier problem first. *Is there a more general problem? or a more special one?*

Carrying out the plan

Write code, test code, write code, test code…

Test each step, each function if possible. Make the write-code-test-code cycle as short as possible. Do not do deep nested calls without checking each function. Mistake in the first step may appear later, disguised as a totally different problem. *Did we get the first step right? The second? …What are the test cases? Can we reason about the correctness of the code?* When confused, go back to function definitions, examine their source code. Throw away code that does not work. Leftover code from dead end attempts and different half solutions turn the original problem into a harder one. *Do we use this line of the code?*

Looking back

Check, reflect and learn. Rewrite for improvement.

Can we check the solution? Does the program do more than what is needed? Is there a way to simplify this solution? Is there a different solution? Can we use the function or its method to solve other problems? Learning starts when the problem is solved. Refactor (rewrite) for simplicity, efficiency or wider applicability.

[1] *How to Solve It – A New Aspect of Mathematical Method* by George Pólya, Princeton University Press, 1945.

Trying to solve a programming problem can be a daunting experience. Not knowing how to start, or not understanding why the code is 'misbehaving', having no clue about the next step – these are intimidating and frustrating. On the other hand, programming is not magic, anyone with a bit of patience can learn it. The reward is great, the joy of seeing the right output for the first time on the screen is immense. Here are a few pieces of advice, that may (or may not) help in overcoming difficulties.

Where do ideas come from?

From previous experience.

There is the myth of pure genius creating great new ideas out of nothing. Even if this is a real skill, most of us don't have it. Luckily, superhuman abilities are not needed for programming. The number of previously solved problems is a reliable measure for proficiency in coding.

Have we solved a related or similar problem before? Can we use its result? or its method?

By *result* we mean the output of a function written for a previous problem. That function may be usable for the current one as well. Especially, if it was refactored for generality in the looking back phase of problem solving.

By *method* we mean the source code of previous solutions. Same type of algorithm might be applicable here with no a little modification. The solution might have a generic form. It might be recursive, or a folding.

Practice

It is not a big secret. If we keep doing something, we get better at it. Problem solving and coding are not exceptions.

What is less known is that solving a problem only once is often not enough. Performing musicians do not stop practising after they could play the whole piece without mistakes for the first time. They play it again, to reinforce the memory and to improve the playing style. Same for programming. You solve a problem, for the first time, probably with the help of someone or the Internet. It is not learning yet, it is only an opportunity for it. After the first solution, you should clear the screen and solve the problem again – this time alone. And again. Repeat this a few times, then you will see a better way to do it. Eventually, you get bored, and that is a good sign. You can move to a harder problem.

Reflection

It is always a good idea to be conscious about the steps we make. This is not to state the obvious that we can't write programs while sleeping. Rather, asking questions like *What did I do? Why did I do that? Is this taking me closer to the goal?* When progress has been made, *Is there a better way? Can this solution be used for something else?*

We tend do this naturally, but not to a large extent. It pays off to ask these questions explicitly. Maybe having a checklist of these questions handy when coding is a useful habit.

Explanations

When a piece of code does not work, the first step in fixing is to find an explanation of why it's not working. When the code does work, we still need to have an explanation why it does so. This is for checking whether we covered all valid inputs.

Pen & Paper

Not to be dismissed as an old technology. One of the most accessible form of external information representation. No need to switch between windows or scroll the page on screen. It helps to reduce the number of things we need to keep in mind at the same time.

Philosophically, it is just another type of computer. A computer can be defined as a device that stores, retrieves and transforms information. A sheet of paper can do all three. Storage: we can write on it. Retrieval: we can look at it. Transformation: the identity map.

Beginner's mind

Most problems, and delays in problem solving are due to the false belief of knowing. "I know how this function works." "I know what this call returns." These statements may or may not be true. When they are false, they prevent us from checking that and we try

to look for the problem somewhere else. The beginner's mindset helps. We admit that we don't know, so we can freely ask simple and basic questions, and check the 'obvious'.

Variations

When the code doesn't work, we go through it step-by-step. Again and again. If the problem persists, it pays off to change something in the revision. Maybe the order of step, or the input. Making a variation increases the chance of finding the mistake.

It's OK to be confused.

Really. Abstract thinking is a defining characteristic of human beings, but in programming we take this ability to extremes, so it does not come that easily.

Professional programmers also have the feeling of being lost frequently. For example, even if someone is a master of a language, still, there is a need to work with someone else's code, and exploring the unknown is not a straight process. Being professional means that you know how to recover quickly by going back to the point where you got lost.

Play in the REPL!

This is a repeated advice, but what does it mean exactly? Playing with what?

Playing in a sense of just trying things out, just reminding ourselves how data structures behave under certain functions.

What are the data structures mentioned in the problem? Characters and strings? Then let's just turn a bunch of characters into strings, or blow up a string into a sequence of characters.

The first goal is the mobilisation of existing knowledge, refreshing memory. Then, still in the REPL, we can start combining existing functions, getting closer to the required data transformation. If some useful combination comes up, after trying it with several inputs, we can just copy paste it to an editor. It becomes the body of a new function, by changing its 'moving parts' to formal parameters.

After solving many problems, the playing phase gets shorter. At some point the direction of the data flow between the REPL and the editor turns around: sending code written in the editor to the REPL for testing purposes.

'Trial and error' to be avoided

Playing in the REPL should not degrade to the trial and error method. Making random changes and hoping for the desired outcome do not involve informal reasoning, thus we will not gain understanding. Being desperate, we can try some arbitrary function calls, to see what happens. However, when looking at the result, it is not enough to ask 'Is this a good result?'. We should investigate why we get this particular result. We need to get an explanation.

Reading and understanding code

Reading source code is imagining what a computer would do when running that program. In a sense, the human brain can act like a computer. This process of 'mental computing' has advantages and limitations. We can do part of the computation instantly, just by imagining a resulting value. But when we actually trace the computaiton, our thinking is relatively slow and there is a limit of how many things we can think of at the same time. The limitations can be alleviated by using the REPL: we can check function calls reparately, fabricating input values whenever needed. Therefore, understanding source code is easier through experimentation in the REPL compared to mere reading.

Learn to throw away code

For small problems, it is a good idea to restart problem solving a couple of times. We are bound to make mistakes, and it is often difficult to understand what goes wrong. Debugging takes a long time. After erasing everything, we might have a good chance of not making the same error again.

If you keep code that is not really working, then it becomes part of the problem to be solved. In software engineering, the practise of throwing code away is very much needed too. Unfortunately, it is often impossible. This happens for instance when some not so good code is in use already. This the problem of *legacy code*. You may need to make

a lot of effort to code around a previous mistake, trying to neutralize it. You basically replace the original problem with a more difficult one.

Wishful thinking

Problems seldom can be solved in one go. Don't expect to be able to do that. No one can. The trick is to be able to break down the problem into smaller and easier ones. The most natural method is to make wishes.

> I have to go through a collection and find elements with a certain property. I wish I had a predicate function that can decide whether an element has that property or not.

There you go! Now you have a simpler problem, just write a predicate function.

Know when to stop

When doing problem seems too difficult even after a long period of time, it is often a good idea to take some rest or do something else. This is especially important if we consider that programming is not the best activity for the human body.

Switching to another activity may actually be the shortest path to the solution. If interested enough, the brain doesn't stop working, it continues subconsciously. So, when you revisit the problem, you may just simply see the solution.

Do only what is needed

Modern programming languages automate many things. For example, functional languages with `map`, `filter`, `reduce` take care of dealing with collections. You only need to create functions that deal with a single element of the collection.