code.energy

How to Code it

Douglas Teixeira

11-14 minutes

Problem solving is a hard, strenuous activity for which, most of the time, there are no fixed rules. Thus, when a book comes out claiming to provide some sort of script for problem solving, it will inevitably draw a lot of attention. It was so with Polya's "How to Solve It", published in 1945. But the book became a huge success–more than one million copies sold world wide–not only because of the topic it addresses, but especially because of
its simple, clear rules for problem solving.

Pólya wrote one of the most famous math books of all times.

In the book, Polya divides problem solving into four phases. First, understand the problem in order to clearly see what you have to find and what you already have at hand. Second, understand how the various parts of the problem are connected, and devise a plan for carrying out computations to find the unknown quantity–the answer to the problem. Third, carefully execute the plan you devised, paying attention to each step of the execution. Fourth, take a step back, analyze your solution, see what worked and what didn't, and what could be generalized or reused to solve other problems.

There is a close relationship between problem solving in Mathematics and Computer Science. Polya's four phases were created for numerical problems, but they can certainly help to solve computational

problems too. Let's investigate these four phases, and gain important insights into what helps programmers solve problems more effectively.

Rule #1: Understanding the problem

Polya's first commandment is: "You have to understand the problem." The first thing you have to do is ask yourself three basic questions: "What is the unknown? What is the data? What are the conditions?"

A thorough understanding of the problem is necessary in any domain. As you have likely experienced, trying to solve a problem you do not understand is extremely difficult and a tad counterintuitive. Yet, we do it all the time.

For a programmer, the urge to jump at the keyboard and start typing a solution to a problem they have yet to fully grasp is too tempting. "I'll just explore the problem a bit", is what we usually think in these moments. And there's nothing wrong with a little exploration. But more often than not, our little explorations turn into flimsy solutions to problems we haven't taken the time to fully comprehend.

Pen and paper are still the best friends of a computer programmer, despite the amount of time we spend at our keyboards. Again, writing code to explore a possible solution is great. But there should be much more back and forth trips between the keyboard and a sheet of paper than usually happens.

A lack of understanding can be detrimental in other scenarios too. Donald Knuth once said that "premature optimization is the root of all evil", in the sense that programmers spend far too much time thinking (or worrying) about the speed of noncritical parts of their programs. If we as programmers better understood our programs (and the problems they solve), we would be better equipped to debug, maintain, and even optimize them. A thorough understanding of a problem is necessary not only to write the program that solves it, but also in every other part of the program's lifecycle.

Rule #2: Devising a Plan

Polya's second commandment is: "Find the connection between the data and the unknown. You may be obliged to consider auxiliary problems if an immediate connection cannot be found. You should eventually obtain a plan for the solution."

So the goal in this phase is to devise a plan for the solution, to think about the steps that can lead you to find the unknown. If we think of it in terms of algorithmic problems, this is the phase in which we design the algorithm to solve the problem.

“An algorithm is a procedure, a recipe, a process, a routine, a method, a set of rules or directions for getting a specific output from a specific input. The distinguishing feature of an algorithm is that all vagueness must be eliminated; the rules must describe operations that are so simple and so well defined that they can be executed by a machine. Furthermore, an algorithm must always terminate after a finite number of steps.”–Donald Knuth

Books upon books have been written on the design of algorithms,and there are several strategies (or paradigms) for this. Algorithms in the same paradigm have the same overall behavior. For instance, all divide-and-conquer algorithms divide the original problem into smaller ones, solve these smaller problems, and then combine their solution to obtain the solution for the original problem. Let’s suppose you have an array of numbers that you want to sort. A divide-and-conquer algorithm would split them into sub-arrays, sort these arrays, and then combine the smaller sorted arrays in order to sort the original one. At first glance, the advantages of this strategy may not be obvious, but some of the fastest sorting algorithms today are divide-and-conquer algorithms.

Sometimes, as you’re designing an algorithm and finding the connection between the data and the unknown, you discover which type of algorithm should be used to solve the problem. This makes it much easier to find the steps that lead to the solution. That’s because your algorithm will employ the same overall behavior from other well-known algorithms of the same type.

But it doesn’t stop there. Designing an algorithm is much more than simply saying it will be a divide-and-conquer, or greedy, or dynamic programming algorithm. There are many details that one needs to be aware of, and several pitfalls that one should avoid. For instance, your plan to reach the solution will be a sequence of steps. It is your responsibility to ensure none of these steps disobey any constraints of the problem.

At this point, you’ve come a long way toward obtaining a plan for reaching the solution. But there’s still a way to go. After designing an algorithm, you should have a general idea of how long it will take to execute as a function of its input. In practical terms, this means conducting a time complexity analysis of your algorithm.

As with the design of algorithms, there are a myriad of books on algorithmic complexity analysis. In short, what you first have to figure out is whether the running time of your algorithm grows linearly, quadratically, or worse, as a function of the size of the input. This will allow you to determine, roughly, how much time it will take for you to get the answer to the problem.

There’s an easy way to become acquainted with all the must-know strategies to create algorithms that all expert programmers know and love. And to get a working understanding of complexity analysis! With

Computer Science Distilled, you'll find this and all other essential computer science tips and tricks that every programmer should know–in an easy language beginners understand. Check it out!

Sometimes, you'll also want to have a certain degree of confidence that your algorithm is correct–that it will always find the correct solution for every possible input. To mathematically prove the correctness of an algorithm is hard. But there are techniques to help you convince yourself that the algorithm is correct, even without proving it.

Some of the most commonly used techniques that give you a sense of the correctness of your algorithm are preconditions, postconditions, and invariants. Preconditions are things that must be true before your algorithm (or a section of it) starts running. For instance, a precondition to a binary search is that the data being searched is sorted. Postconditions are conditions that have to be met when a given portion of your algorithm finishes. In the case of a sorting algorithm function, a postcondition might be that the input array is sorted at the end of the function. An invariant is a condition that must always be met throughout the execution of the algorithm. For example, an invariant in a sorting algorithm might be: "all elements up to the section of the array delimited by variable x are sorted".

You can check preconditions, postconditions, and ensure invariants by using assertion. For instance, the statement `assert( n >= 0 )` will do nothing if n is zero or positive, but it will raise an error if n is negative. You can add a check throughout your algorithm, if "n cannot be negative" is an invariant.

Rule #3: Carrying out the Plan

After all the work you've put into designing an algorithm, next comes the easy part, relatively speaking. As Polya puts it: "To devise a plan, to conceive the idea of the solution is not easy. It takes so much to succeed; formerly acquired knowledge, good mental habits, concentration upon the purpose, and one more thing: good luck. To carry out the plan is much easier; what we need is mainly patience."

In Computer Science, as well as a lot of patience, to carry out the plan one needs to know how to implement the proposed solution in a given programming language. In other words, this stage is about turning an algorithm into a program. According to Donald Knuth, "the word algorithm denotes an abstract method for computing some output from some input, while a program is an embodiment of a computation method in some language."

The question then becomes: given a specific algorithm, what are the skills one needs to have and the steps one needs to follow to implement the algorithm in a programming language of choice?

You need, of course, to know a programming language. It might sound obvious, but knowing a programming language–and knowing it well–is an invaluable skill. Get to know the syntax of a

programming language, get to know the libraries available to it, and how to use these libraries. Learn what their methods expect as input and what they output. Gradually, you'll be able to write programs without looking online at what parameters you need to pass to the library functions you are using, which saves a lot of time. By mastering a programming language you will be able to turn algorithms into programs much more easily, faster, and elegantly. Besides, you will be able to prevent and identify errors in your programs more effectively.

Programming is an error-prone activity, and one needs to be prepared to identify and correct bugs throughout the whole process. Your algorithm might be flawless, but bugs can creep in when you try to implement an algorithm in a programming language.

The best way to identify bugs in your code is to test it extensively. Test each function you write, test how the functions interact with one another, test how the whole program behaves when you weave your functions together. Test not only the normal, expected inputs, but also the corner cases as well as how your program behaves when given unexpected inputs.

Rule #4: Looking Back

Looking back, you ensured a thorough understanding of the problem, then you designed an algorithm to solve it, and finally you implemented your algorithm.

> You've finally made it! You've got an answer to your problem.

Now, examine your solution, remember all that you have done. Meditate on what worked and what didn't. Evaluate possible ways to make your solution more general, by thinking about how it might be applied to other problems. You can also investigate ways to make your program more elegant, simpler, or faster.

Maybe there’s a detail in the program’s constraints that you can leverage to speed up your program–or to simplify your code. It’s easier to find these types of tricks now that your program is working. Taking a moment to look back and trying to improve often leads to important breakthroughs. And it improves your problem solving skills in general.