All you need to know about Algorithm Complexity Evaluation

by Susanna Balashova - Monday, January 11, 2021

https://gonitsora.com/all-you-need-to-know-about-algorithm-complexity-evaluation/

Computers are highly complex, complicated machines capable of storing, processing, and analyzing vast quantities of data far faster than any single human ever could. While the human brain and nervous system are very impressive in their own right, they simply can't match up with the speeds and ruthless efficiency of these magical machines we've created.

However, in spite of their immense speeds and powers, computers aren't limitless in their abilities. They have set parameters and boundaries that simply can't be surpassed until stronger technology is developed.

We can understand the general 'power' of a computer by looking at its processing speed, which is measured in GHz. A 3GHz processor can tackle 3 billion basic instructions every single second, with a 'basic instruction' basically being something like adding numbers or assigning values in the system memory.

The code we can read, like Python or Java, gets transformed into code that the computer can read, before being executed, and just one small section of our human code can turn into many basic instructions for the computer to carry out.

So, when a computer is reading lots of code and handling multiple programs and processes all at once, it's important for machines to be efficient in terms of processing the data they receive and carrying out their functions. This is where algorithmic complexity comes into play.

What is algorithmic complexity?

Well, it might sound complicated, but algorithmic complexity is actually pretty simple. Algorithms (which are essentially lists of steps) for tasks are more complex when they have more steps involved in them. Basically, a longer algorithm will be more complex than a shorter one.

So how do we evaluate algorithmic complexity? Well, in theory, there are a few possible ways we could go about this. We could write an algorithm and then try to work out the number of steps it needs to run.

But how do we quantify what makes up a single step? Should it be a single line of code? If so, we could just look at the code data and count the number of lines to evaluate the complexity of the algorithm, but this could cause all kinds of problems, as code can be written differently and longer sections of code can be condensed quite easily from 10 lines to just 2 or 3 lines in some situations.

Fortunately, this isn't the right way to calculate algorithmic complexity. So what about the lines of machine code? Would we theoretically be able to count those and then work out the complexity of an algorithm?

Well, while it might be possible on paper, in practice it would be a terrible way to go about things. Different computer systems have different types of machine code they can read, and each one would be represented differently, resulting in anomalous results that give us no true conclusions or evaluations whatsoever.

So what does actually count as a 'step' in an algorithm?

Well, the usual method to calculate algorithmic complexity is to look at each 'code-block', which is a section of code that represents a single operation relevant to our input data. It can be one line of code or more, linearly running from start to finish without any loops.

This is a useful way to define algorithm steps, but we can go even further and eliminate the need to look at code entirely.

Experienced programmers are, in fact, able to take a look at code, effectively translate it back into algorithm form by looking at the individual blocks, and then see the algorithm in basic, simple English we can all understand.

At that point, it's a lot easier for programmers, or indeed anyone with thesis help, to take a look at the algorithm and work out just how complex it really is, based on what it involves.

Big O

Finally, we have to take a look at Big O notation, which basically tells us how good an algorithm is at scaling. For most people, Big O and algorithmic complexity have the same basic definition.

To calculate Big O, we follow the same process as above, but take the final expression and remove all non-dominant terms and constant coefficients, leading up to a final result that shows us exactly how well our chosen algorithm can scale.

Once we know this, figuring out the complexity of any algorithm is actually quite simple, and we also learn a lot more about each algorithm's functionality along the way, showing that algorithm complexity evaluation isn't so complex, after all.
