Algorithms and Data Structures

By Grady Fitzpatrick

Algorithms

What is an Algorithm?

An algorithm is, at the most general level, a set of instructions followed in order to achieve some aim. For something like cooking, the method could be seen as the algorithm. In computing, algorithms can be considered to be the high level view of the action, such as simply specifying the concept of how the aim is achieved, or at the low level, where each individual instruction to the computer is specified, or anywhere in between.

For example, an algorithm that only works on sorted data can be made to work on unsorted data by simply sorting the data before performing the normal operations, in such a case, the step "sort the data" is all that would be necessary to add, because how the data is sorted is not important.

What isn't an Algorithm?

Algorithms are just the sets of actions, rather than the whole set. For example, in a recipe, the ingredients are necessary to carry out the method, but the ingredients aren't part of the algorithm itself.

In terms of computer algorithms, things like Binary Search Trees, Dictionaries and Linked Lists aren't algorithms, even though they mandate certain behaviours, they contain much more than the algorithm part. Something like Binary Search on a Binary Search Tree would be an algorithm, as it defines the set of instructions to find something in the tree (which is the aim). Even though all operations on the binary search tree, such as its creation, look up, storage, deletion, and sorting could be considered algorithms, however the arrangement of bytes does not affect the algorithm.

Why does it matter?

Knowing what are algorithms and what are data structures allows programs to be better separated into behaviour and structure, which tends to lead to better code.

What is efficiency?

Efficiency is essentially how well resources are allocated. In computing we primarily deal with two types of efficiency, time efficiency, that is, how long a piece of code takes to execute, and space efficiency, how much space a particular algorithm needs to perform its actions.

One of the difficulties we immediately run into with efficiency is "how do we measure it?" This issue relates to a key aim of the subject to determine what algorithm is most appropriate to

use in each situation. In computing it seems like this would be an easy question to answer, just run the algorithm and see how much space it takes up and how long it takes to run, unfortunately, this only tells us how this algorithm ran on our computer, with our particular data set, it would also be much more useful to have a more general measure that we could use to determine how fast the code we will write will run compared to other choices without having to run the code, or, even better, without even having to write the code.

Accuracy

Accuracy is how close to the correct answer, or best outcome of the algorithm, the algorithm performs. This could be getting 95% of predictions correct or giving us the correct answer to an equation we give the algorithm. You could say we'd like all of our programs to give us perfectly accurate outputs all the time. That is, the input we give the algorithm will give the output that perfectly matches that input.

Why might we accept a program that isn't accurate?

When we write algorithms, we want them to run quickly; however what happens if we want our algorithms to run faster than we think they theoretically can? In those cases, we'll often accept some inaccuracy in our output, as we'd often rather a mostly correct answer than no answer at all. There exist some problems in computing where for some input we have fairly good approximations to correct answers, but the methods to find the exact answers could take so long that we wouldn't be alive to see them finish being computed. So in these cases we might accept less than perfect answers. Also, we may want our algorithm to make assumptions when some data is missing. If you want to find out more about accuracy, you might find the subject Knowledge Technologies interesting.

Correctness

Correctness is whether following an algorithm actually always achieves our aims. It is important in order to make sure our program always works no matter what we give it. Though it's fairly straight forward to understand what a correct program is, actually writing one and making sure what you actually write *is* a correct program is a significantly harder problem, by using debugging tools and techniques, bugs and problems with achieving aims can be avoided.

Why might we accept a program that isn't correct?

When we're writing a program, it may be useful to build certain parts of the program at a time and then test for certain inputs and outputs using some scaffolding, because the scaffolding only makes it appear that the program is working, the program can't entirely be considered correct since it does not achieve the aims of the algorithm for all inputs, however allowing a program to be not correct while building it can ensure that the parts come together without issue.

Staging

Building on the ideas mentioned about allowing programs to be not correct, we may build our program in stages, where the first stage of the program has some functionality that is a subset of the final functionality and the program is measured against the aim of being able to perform

that subset, thoroughly checking that the program is correct against that stage's criteria, and after the program has been expanded to encompass all stages, the program is able to perform the original aim.

An example of staging would be where we had a program which allowed operations on a list, and ultimately we wanted to be able to create, add to, delete from, search, update, save and load a list, we might start with creating the list as the first stage, followed by saving, loading, adding to, deleting from, etc. After all these stages have been written and thoroughly tested, we end up with a (very likely) correct list program.