Algorithms & Programming: It is the differences that are important when teaching programming

This article explains how computing science educators often confuse programming and algorithms before outlining the differences between the two and concrete steps we can take to give both their proper place. However, before we start it should be noted that we are not talking about formal algorithms which can be likened to mathematical formula and are often the best solution to a complex problem but algorithms that are part of the programming planning stage.

Why teachers conflate algorithms and programming

Teachers without a background in computing science can easily be forgiven for confusing algorithms and programming. The Encyclopaedia Britannica[1] describes an algorithm as

> *'a 'systematic procedure that produces – in a finite number of steps – the answer to a question or the solution of a problem'*

While Moore & Mertens in the Nature of Computation[2] describe it as

> *'a series of elementary computation steps which, if carried out, will produce the desired output'*

An everyday reading of these and many other similar definitions would suggest an overlap. Both use the same concepts of sequence, repetition and selection. Both can utilise data structures such as variables. Both consist of finite steps to a desired end. Little wonder that Waite in her recent research article[3] states that

> *'In discussion* (with primary K-5 teachers)*, the term code and algorithm were used interchangeably, or the teacher were very unsure of what an algorithm might be for a programming project.'*

Differences between algorithms and programming

Spraul[4] describes the difference between algorithms and programming as

> *'An algorithm is described "in English" rather than written in a computer programming language.'*

While Erwig[5] states that

> *'An algorithm cannot produce a computation by itself.'*

Tisdall[6] writes that

> *'An algorithm is implemented by coding it in a specific computer language, but the algorithm is the idea of the computation.'*

Kernighan[7] addresses the differences by writing

> *'The distinction between an algorithm and a program is like the difference between a blueprint and a building; one is an idealization and the other is the real thing.'*

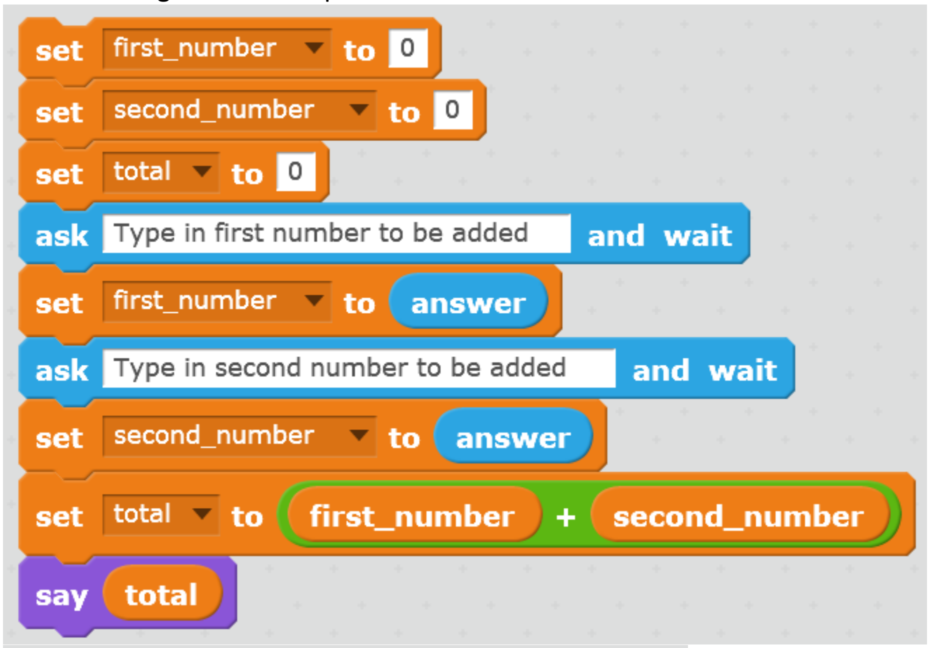
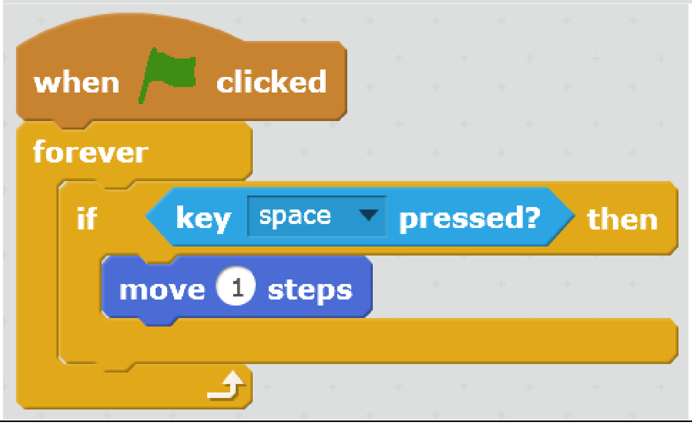
To summarise these and other voices. An algorithm is not written in code. So, if pupils are writing out code exactly as it will be programmed, they are planning with code not planning by writing an algorithm. Waite[3] believes that this confusion first arises when pupils design routes for their simple bee-bot/blue-bot/Roamer too robots which are called algorithms. These simple arrows are the same as the programming inputs on the top of the device which means they are planning with code rather than algorithmic plans.

Should we stop using the word algorithm at this point? This author believes that KS1 pupils using these symbols understand the planning blueprint part of what an algorithm is and that at this age with these devices it is not possible to create algorithms that are abstracted away from the concrete reality. Often in teaching we present a partial truth before pupils are ready for the whole truth and we might still identify this as an algorithm as it is a series of steps to achieve a desired result as long as we determine to develop pupils algorithmic understanding separate from programming as soon as this is possible.

How can we separate algorithm and programming successfully?

The key is understanding the layers of abstraction necessary to move from idea to working program.

Ideas level	This is where the overall idea is expressed. *I want to make a game program that moves my character around a maze collecting items as it goes. I want to make a program that adds two numbers together and shows the result. I want to make a program where two characters talk to each other.*
Planning Level Objects Initialisation Algorithm(s)	This is where we decide which characters and objects we want in our program and what they will do at the start (initialisation) and what algorithmic steps they will carry out to achieve our ideas. A game algorithm may consist of lots of small decomposed scripts to achieve parts of the game design where as an algorithm to add two numbers and show the result will probably be expressed in just one script. Where ever possible teachers should use language that is similar but different from the programming language chosen when modelling algorithms or teaching algorithmic concepts so that pupils realise that this is about thinking through their ideas algorithmically rather than writing out code. [8] For example *Assign 0 to first_number variable* *Assign 0 to variable called second_number* *Assign 0 to total variable* *Ask the user what number they would like to add first and assign their answer to a variable called first_number* *Ask the user what number they would like to add to the first number and assign their answer to second_number variable* *Add first_number to second_number and assign the result to total* *Say what is inside the total variable* Or *My hero character will start at the centre of the maze and will move by* *Always loop* *If space key is touched* *Walk a step* Both examples could be written in many ways and as long as they make sense and don't have logical errors teachers should accept and encourage a wide variety of algorithmic forms which can be easily read in pupils native language.

Code level	This is where the algorithm is turned into code and the objects are turned into real things on the computer.  set first_number to 0 set second_number to 0 set total to 0 ask Type in first number to be added and wait set first_number to answer ask Type in second number to be added and wait set second_number to answer set total to first_number + second_number say total  when clicked forever if key space pressed? then move 1 steps
Execution level[3]	This is where the code is run. Will it work? Will anything need to be changed? Are there any bugs?

These levels are not designed to be just progressed through from top to bottom. A pupil might write only a small part of the algorithm before coding it and they might go back and adapt their idea and build on their algorithm after considering their planning, coding or execution. Teachers might choose to assess a project through the planning level rather than the code and execution level which will improve the status of this level as there is evidence that pupils dislike planning. They might choose to model or part model a level so that pupils have useful examples that they can adapt and work from. Pupils may be encouraged to plan in pairs but program individually or vice versa to encourage discussion or assess individual learning. Teachers might provide scaffolding at any level where they deem it useful for groups, individuals or the whole class. This author agrees with Armoni[8] when she writes that

> *'Instructors (teachers) should be aware of which abstraction level they work on and should clearly distinguish between the levels. It is important that students are aware of which level they work at, and when they change it.'*

Benefits of developing an algorithm/planning level independently to the code level

By developing the ideas and planning levels separate to the code and execution level pupils can port their algorithmic thinking ideas between different programming languages and other algorithmic unplugged outcomes. Their thinking is independent of any programming language and therefore more portable. I provide some evidence towards this in my article Does writing algorithms improve pupils chances of porting their knowledge?

Phil Bagge January 2019

[1] Encyclopaedia Britannica (2015)

[2] Christopher Moore & Stephen Mertens, The Nature of Computation Oxford, 2011

[3] Jane Waite, Abstraction in Action: K-5 teachers' uses of levels of abstraction, particularly the design level, in teaching programming, 2017 Jane names the levels as shown above but calls the last level running the code.

[4]V. Anton Spraul, Computer Science Made Simple: Learn how hardware and software work, Crown/Archetype, 2010

[5]Martin Erwig, Once Upon an Algorithm: How Stories Explain Computing, MIT Press, 2017

[6]James Tisdall, Beginning Perl for Bioinformatics: An Introduction to Perl for Biologists, O'Reilly Media, 2001

[7]Brian W. Kernighan, Understanding the Digital World: What You Need to Know about Computers, Princeton University Press, 2017

[8]Armoni, Teaching Abstract Thinking in Introduction to Computer Science for 7th Graders, 2013