ALGORITHMIC THINKING

We now move from logical thinking on to algorithmic thinking. This section will introduce the basic properties of algorithms (sequence, iteration and selection), as well as the use of state.

In brief: Algorithms

Logic and algorithms are not the same. Rather, algorithms build on logic because, as part of their work, they make logical decisions. The other part of their work is 'stitching' those decisions together.[11]

Logic gives you a set of rules that allow you to reason about some aspect of the world. That world does not have to be static. Logic can deal with things that are dynamic and continually changing. This we saw with the Noughts and Crosses examples. The rules of the game were laid out as logical statements. Sometimes the propositions were true, sometimes false. By working out their truth values, we could come to conclusions about how the game would be in different situations.

But if we want to go further, if we want to build functioning systems based on rules, then logic alone isn't sufficient. We need something that can integrate all these rules and execute actions based on the outcomes of evaluating them. That something is algorithms, and they are the power behind all real-world computational systems.

Algorithm: a sequence of clearly defined steps that describe a process to follow a finite set of unambiguous instructions with clear start and end points.

Algorithms are a way of specifying a multi-step task, and are especially useful when we wish to explain to a third party (be it human or machine) how to carry out steps with extreme precision. As with logic, humans already have an intuitive understanding of algorithms. But, at the same time, a rich and precise science dictates exactly how algorithms work. Gaining a deeper understanding of this will improve your algorithmic thinking. This is important because a correct algorithm is the ultimate basis of any computer-based solution.

Intuition vs precision

Algorithms are tricky things. In a sense, they're intuitive concepts that have been with us for centuries, yet they only received a definition in the last hundred years or so (Knuth, 1997). It's easy to articulate them intuitively, but at the same time computer science's concept of an algorithm is multi-faceted and can take beginners some time to comprehend. Misunderstandings commonly arise (see Pea et al., 1987; Pane et al., 2001), so we'll focus on building a solid understanding.

Before going into the hard details, let's take as our starting point the simple definition in the box above. You've very likely dealt with algorithms in this sense already. Anyone who has followed a recipe while cooking has encountered something like this. Or anyone

who went on a treasure hunt as a child. Or anyone who has assembled furniture. An algorithm is a way of making our processes explicit so that you can communicate them to someone else in a way that allows them to carry out those same steps too. These are only analogies however. Strictly speaking, algorithms perform operations on data rather than cake ingredients or coffee tables.

The earliest pioneers of computer science examined what it meant to communicate ideas to a computer. They sought a means for us to take the ideas in our heads and put them into a form that computers can understand and compute on our behalf. They discovered that our existing, intuitive ideas of algorithms were insufficient, so they took those ideas and re-formed them into the kind of clear and unambiguous form that computers require. By doing so, they made algorithms our means for giving computers instructions.

As precise as they became, algorithms were also rendered more complex, so much so that a formal description of them can stretch over several paragraphs. In our case, the best way to approach a definition is to take each property of algorithms, one by one, and explain it. Along the way, we'll use an analogy to one of our intuitive ideas of an algorithm (in this case, following a recipe) to help us understand.

Defining algorithms

The definition of an algorithm is complex and involves several properties. This subsection describes those properties.

Collection of individual steps

The first property to mention just restates something said earlier: an algorithm is a **collection of individual steps**. A recipe fits this analogy quite simply, filled as it is with steps like: 'pre-heat the oven to 180 degrees Celsius' or 'add two tablespoons of sugar to the bowl'.

Definiteness

Following on from that property is **definiteness**, meaning that every step must be precisely defined. Each step in an algorithm can have one and only one meaning, otherwise it is ambiguous. Similarly, chefs have come to the same conclusion, which is why they produce recipes using precise measurements instead of writing things like 'some sugar' or 'cook it for a while'.

Sequential

Algorithms are also **sequential**. The steps that make up the process must be carried out in the order specified. Failing to do this means that the result of executing the algorithm is likely incorrect. Think back to the analogy. Dicing an onion and frying an onion are different steps. Dicing an onion **before** you fry it has a different outcome than the reverse. Similarly, multiplying a number by 2 then adding 5 to it yields a different result from adding 5 first **then** doubling it. Like a recipe, you must respect the sequence when running through an algorithm for it to have any meaningful result.

Detour: State in algorithms

It's worth going on a brief detour here to examine why sequence is so important to algorithms. It's all to do with **state**, by which I simply mean the current values of all

the things the algorithm is keeping track of. As a computer progresses through an algorithm, just as you progress through a recipe, the state of things can change. Clearly sequencing the steps of an algorithm ensures the that state always changes in the same way whenever the algorithm is executed.

State: the current configuration of all information kept track of by a program at any one instant in time.

There's an important implication to this: there is **no 'global view'** when it comes to algorithms. This means that, at each instant in time, the environment in which the algorithm is being run exists in some particular state. But by the time the next step is executed, something might have changed. The environment really exists as a series of snapshots, one for each step of the algorithm. The recipe analogy spells this out. At the start you might have butter, flour, milk, eggs and sugar. After each step, you take a photograph of the kitchen. The photos will show that, bit by bit, the state of the ingredients changes. Flour goes into a bowl; then the eggs join it; then the butter goes into the pan; and so on. There is no global view of the ingredients; just a series of snapshots.

For algorithms, this means that individual steps are executed one by one and only a single step can be under consideration at any one time. Once a step has been executed, the computer forgets all about it and moves on to the next.

This means that some way of 'remembering' things that happened in previous steps needs to be provided. If you, the algorithm's author, want the computer to remember the result of a step for later use, you have to explicitly tell the computer to do so. Algorithms provide for this by means of **variables**.

Despite sharing the same name, variables in programming do not correspond exactly to variables from mathematics. Mathematical variables are used in a function to represent quantities that vary. A variable in an algorithm is more like a scratchpad that's used as a placeholder for important information the computer should keep note of. It can also have its values updated throughout an algorithm's execution (this is called **assignment**). For example, if your algorithm counts the number of times a specific word appears in a news article, it would use a variable to keep track of the number as it reads through the text.

Controlling algorithm execution

In this section we'll look at two ways of controlling the execution of an algorithm.

Iteration

As well as recording information pertinent to a problem, variables can also be used to control the execution of an algorithm. At a basic level, they can be used in two ways. One of those is **iteration**, also known as looping. Iteration allows you to repeat a series

of steps over and over, without the need to write out each individual step manually. This can be very useful. Imagine having to write out the lyrics to a song like *99 Bottles of Beer*. If you don't know it, it goes like this:

99 bottles of beer on the wall,
99 bottles of beer.
Take one down, pass it around,
98 bottles of beer on the wall.

98 bottles of beer on the wall,
98 bottles of beer.
Take one down, pass it around,
97 bottles of beer on the wall.

And keeps repeating until the last verse:

1 bottle of beer on the wall,
1 bottle of beer.
Take it down, pass it around,
No more bottles of beer on the wall.

To save yourself a lot of typing, you could, instead, write just one verse along with instructions on how to repeat all the following verses. Something like:

X stands for the number of bottles. At the start, X is 99. Sing the verse:

X bottles of beer on the wall,
X bottles of beer.
Take one down, pass it around,
X-1 bottles of beer on the wall.

Subtract 1 from X. Repeat the verse if X is greater than 0, otherwise finish.

This is an example of iteration in action and it shows two things:

1. How a variable is used to control algorithm execution. The thing that changes from one loop to the next is encapsulated in a variable, in this case the number of remaining beer bottles. Everything else stays constant.
2. You must specify under what conditions the loop should terminate.

This last point brings us to the second method used for controlling algorithm execution.

Selection

In a loop, one way to control how many times the steps are repeated is simply to specify it; something like 'repeat the following steps 99 times'. But notice that the example above doesn't do that. Instead, it uses *selection* (aka a conditional), which is a way to test a variable's current value and make a decision based on it. In the '99 Bottles' song, the conditional comes at the end (repeat the verse **if X is greater than 0**). It's telling the computer to do something so long as the condition specified is currently true. So long as

a positive number of bottles remain, that condition is true and so the verse is repeated. But we know that the number is steadily decreasing and will eventually reach zero. At this point the condition becomes false (zero is not greater than zero), and so the verse is not repeated again.

Conditions can be used at any stage in an algorithm, not only to control loops. Wherever they are, they all serve the same purpose: creating a point at which the computer has to decide between performing a set of steps or not, and encapsulating the information needed to make that decision.

Example algorithm

This example illustrates some of the ideas just discussed. It is a partial and very simple algorithm for controlling a game of Noughts and Crosses. It's written using **pseudocode**, which is an informal means of writing an algorithm. It's aimed at a human audience rather than a computer, but it nevertheless closely follows the conventions of programming languages. This means you can easily translate your ideas into real program code by first writing them as pseudocode.

```
 1. game is started
 2. begin loop:
    3. prompt player to choose a square
    4. if chosen square is not occupied, then put player's symbol in
       that square
    5. check board to see if a row has been achieved
    6. if a row has been achieved, then game is won
    7. if a row has not been achieved and no squares are available,
       then game is drawn
    8. switch to other player
 9. exit loop if game is won or game is drawn
10. display message 'Game over'
```

Let's examine this algorithm:

- When this algorithm is executed, the computer goes through each line one at a time. (This is an example of **sequence**.)
- Line 1 initialises a variable, `game`, with a particular value, `started` (**state**).
- Line 2 sets up the starting point of a loop (**iteration**). All lines that are 'inside' this loop are indented to make it clearer what's inside and outside.
- Line 3 asks for input from the player. The choice is recorded.
- Line 4 makes a **selection** based on the player's choice. (Incidentally, this is a rather user-unfriendly algorithm. If a player accidentally chooses an occupied square, they get no opportunity to fix their mistake!)
- Lines 6 and 7 both make selections, altering the value of the `game` variable under certain conditions.

- Line 9 is the end of the loop. A choice is made whether or not to go through the loop again depending on the associated condition **at this point**. There are two possibilities:

    1. The `game` variable hasn't been altered yet, meaning it still has the value `started`. In this case, execution moves back to line 3.
    2. At some point, the game was won or drawn. In either case, the loop ends and execution continues onto line 10, whereupon the game finishes with a traditional 'Game over' message.

'GOTCHAS'

When first encountering logic and algorithms, most people tend to make the same mistakes or faulty assumptions. This section will point them out and explain how to avoid them.

The need for clarity and meticulousness

You might have noticed the emphasis on clarity, precision and meticulousness when working with logic and algorithms. Computer programs demand precision in their rules and instructions to such an extent that it might have the appearance of pedantry. But it really isn't. There are good reasons behind the need for clarity, which all boil down to a computer's nature.

A couple of facts about this are worth pointing out:

- A computer will do exactly as it is told. If it is told to do something impossible, it will crash.
- A computer has no innate intelligence of its own. It will not do anything that it has not been instructed to do.
- A computer has no common sense. It will not try to interpret your instructions in different ways. It will not make assumptions or fill in the obvious blanks in an incomplete algorithm.

In short, a computer will do only what you tell it to do. If you don't tell it to do something, it won't do it. If you tell a computer to do something stupid, like divide a number by zero, it will try to do it anyway. If a computer does something you didn't think you told it to do, that's probably because you misunderstood your own instructions to it.

Earlier, I drew an analogy between an algorithm and a recipe. However, the audience for a recipe is another person. Granted, a person can be the audience for an algorithm, but people are used to communicating using natural language. Even when it's very formal, that language can still be ambiguous and sometimes open to interpretation. Furthermore, the author may leave certain things up to a reader's common sense rather than labour a point.

You must be mindful of the (electronic) computer's limitations outlined above. Think of giving a recipe to an exceedingly literal-minded robot from a cheesy sci-fi film. Would it be the robot's fault if it read 'stir the milk, flour and eggs in a bowl', then tried to climb into the bowl itself? Or if it waited forever for the 'dough to rise' because the dough hadn't begun to levitate yet?

The point is, the goal of problem-solving using computational thinking is a solution that a computer can execute. That means an algorithm with precise, unambiguous meaning that requires neither creative intelligence nor common sense to understand fully.

And that often means splitting hairs and spelling things out in seemingly pedantic detail.

Incorrect use of logical operators

Our intuitive understanding of language leads to other 'gotchas'. Something as seemingly straightforward as our use of the words 'and', 'or', 'not' and 'then' can catch us out. For example, the statement:

> 'Everyone whose surname begins with A and B is assigned to Group 1.'

might seem logical to a human, but to a computer, which strictly follows the logical meaning of AND, it's illogical. That's because, as written, the statement tests two things: does a surname begin with A and does it begin with B? However, no surname can have different first letters at the same time. You can ask a computer to carry out this test as much as you like, but it will always fail.

Furthermore, in everyday language, people sometimes use 'and' as a sequencing term, as in:

> 'The player took their turn and the game was finished.'

Again, that makes no logical sense to a computer. A logical statement like this describes a state of affairs at an instant in time. A player cannot make a move in a game when the game is finished.

To avoid confusion, never play loose with the phrasing of logic. Always use the correct operators. Truth tables are extremely helpful when in doubt.

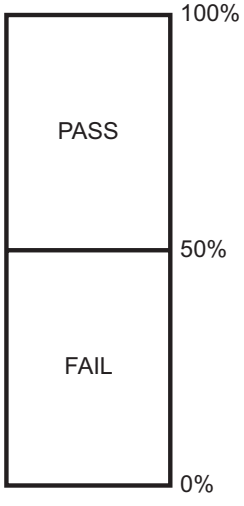
Logic and algorithms treat the 'if–then' form a little differently from each other. Consider 'if X, then Y'. Whereas a logical implication states how the truth values of X and Y are related, the intent of the algorithmic 'if–then' form is to make the execution of step Y dependent on X being true. If X isn't true, step Y isn't executed.

Missing certain eventualities

Another 'gotcha' related to the use of 'if–then' in algorithms is what to do when the condition is false. For example, here's an algorithm used to grade a student's work. It's

very simple. A student can earn one of two grades (pass or fail) and the student requires a score over 50 per cent to gain a pass (see Figure 2.6).

Figure 2.6 Grading levels for fail and pass



If score is greater than 50 per cent,
then mark student's grade as a pass.

Do you see a problem?

What happens in cases when the student scores 50 per cent or less? Common sense tells a human that the student should get a failing grade, because that's what the problem implies. But remember, computers have no common sense and won't do anything you don't tell them to do. That's why algorithms also include the 'if–then–else' form.[12] Using this, you could amend the algorithm to look like this:

if score is greater than 50 per cent,
then mark student's grade as a pass,
else mark student's grade as a fail.

When the condition in an 'if–then–else' is true, only the part after the 'then' is executed; when the condition is false, only the part after the 'else' is executed.

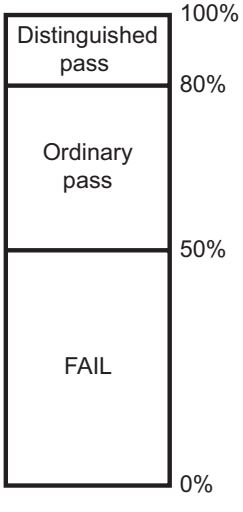
Complex conditionals

By itself, the NOT operator is simple to understand. For example, the grading algorithm could be rewritten as follows:

if score is not greater than 50 per cent,
then mark student's grade as a fail,
else mark student's grade as a pass.

No surprises there. However, difficulty can arise when a condition becomes more complex. Imagine an additional grade is introduced: students scoring over 80 per cent are awarded a distinguished pass (see Figure 2.7). A pass would only be awarded to scores over 50 and less than 80 per cent.

Figure 2.7 Grading levels for fail, ordinary pass and distinguished pass



To check for an ordinary passing score, you might begin an 'if–then' that tests two conditions like this:

```
if score is not less than 51 per cent or greater than 80 per cent,
then mark student's grade as an ordinary pass
...
```

A deliberate mistake lies in this little algorithm. All will be revealed shortly.

That might seem precise at first glance, but it could be interpreted in at least two different ways. Specifically, does the 'not' apply to both conditions or only to the first? Taking everyday language as a guide suggests it applies to both, just like saying 'the car is not big or red' means the car is neither big nor red. However, the realms of natural language and logic don't always align. This is a case where they don't and we'll see why shortly.

To avoid such confusion, symbolic logic provides very specific rules about how to interpret the various parts of logical statements. These rules are called the **order of precedence**. They dictate in what order each part of a logical expression should be evaluated. It all works just like in mathematics.

Table 2.8 Order of precedence for selected logical operators

Rank	Operator	Name
1	( )	Parentheses
2	¬	Not
3	>	Greater than
	<	Less than
4	Λ	Logical AND
5	V	Logical OR

When evaluating something like

$$9 - 3 \times 4$$

you get different answers depending on the order in which you carry out the subtraction and the multiplication. But strictly speaking, there's only one correct answer. That's because the rules of the order of precedence dictate in which order each part is evaluated; those with higher precedence are carried out first. To make the order explicit, you can put brackets around each expression. If we were to put them in explicitly, it would look like this:

First the multiplication

$$9 - (3 \times 4)$$

then the subtraction

$$(9 - (3 \times 4))$$

Technically, that means exactly the same thing as the original expression. Now we can see clearly that the multiplication must happen first and so the correct answer is -3. If you wanted to perform the subtraction first, that would be fine, but you would need to put your own brackets into the expression.[13] In this case, it would appear so:

$$(9 - 3) \times 4$$

This changes the meaning of the expression and its result (24).

Let's return to our student grading example. Before applying the rules, we should rewrite the expression using correct notation. Taking each condition one by one:

score is not less than 50 per cent
becomes
¬ score < 51

while
    greater than 80 per cent
becomes
    score > 80

Connecting them with the proper OR symbol gives us:

$$\text{if} \neg \textit{score} < 51 \text{V} \textit{score} > 80$$

To help us see the order in which things are evaluated, put the implicit brackets in place. Negation has the highest precedence of the operators here, so it gets brackets first:

$$\text{if} \neg (\textit{score} < 51) \text{ V } \textit{score} > 80$$

Comparisons have a higher precedence than disjunctions, so they get their brackets next (the left-hand comparison already has brackets thanks to the negation operator):

$$\text{if} \neg (\textit{score} < 51) \text{ V } (\textit{score} > 80)$$

Finally, the disjunction gets its brackets.

$$\text{if}(\neg (\textit{score} < 51) \text{ V } (\textit{score} > 80))$$

The expression is ready to be evaluated. To do that, we need to assign a value to 'score'. Let's arbitrarily choose 55.

Now that the brackets are in place, we can go from left to right, evaluating each piece as we go. NOT flips the truth value of the thing to its right, so we need to work out if that thing is true or not. 55 is not less than 51, therefore we substitute 'score < 51' with its truth value:

$$\text{if} \neg (\textit{false}) \text{ V } (\textit{score} > 80)$$

In other words:

$$\text{if } \textit{true} \text{ V } (\textit{score} > 80)$$

We've worked out the value of OR's left-hand condition. Now we need to evaluate the condition on its right. 55 is not greater than 80, therefore:

$$\text{if } \textit{true} \text{ V } \textit{false}$$

Consulting the truth table for OR, we find that when either or both conditions are true, then the whole expression is true. In this case, the student correctly gets an ordinary pass.

Let's try another score to make sure the expression works for distinguished passes too. We'll use 85 as the score. 85 is not less than 51, therefore:

$$\text{if} \neg (\textit{false}) \text{ V } (\textit{score} > 80)$$

In other words:

$$\text{if } \textit{true} \text{ V } (\textit{score} > 80)$$

And 85 is greater than 80, so:

$$\text{if } \textit{true} \text{ V } \textit{true}$$

Again, we've evaluated the expression to find that it is true. But this is not what we wanted. Look again at the expression in the context of the algorithm:

> if score is not less than 51% or greater than 80%,
> then mark student's grade as an ordinary pass,
> ...

We've demonstrated that, according to the algorithm in its present state, the student gets an ordinary pass even though they scored over 80!

The problem is that our natural language way of expressing our intentions doesn't quite agree with the rules of logic. To correct our original algorithm, we would have to put our own explicit brackets into the expression to 'override' the usual orders of precedence, like so:

$$\text{if}\neg(\textit{score} < 51 \text{ V } \textit{score} > 80)$$

This time, we evaluate **everything** inside the brackets first before evaluating the negation. Now, for a score of 85, the procedure looks like this:

$$\begin{aligned}
&\text{if}\neg(\textit{score} < 51 \text{ V } \textit{score} > 80)\\
&\quad \text{if}\neg(\textit{true} \text{ V } \textit{score} > 80)\\
&\quad\quad \text{if}\neg(\textit{true} \text{ V } \textit{true})\\
&\quad\quad\quad \text{if}\neg(\textit{true})\\
&\quad\quad\quad\quad \text{if } \textit{false}
\end{aligned}$$

The student will therefore not be assigned the wrong grade, which is good because it's always embarrassing when that happens.

To avoid mistakes like this, it's a good idea to use the correct logical notation from the very start, even when you're just beginning to sketch out your ideas.

SUMMARY

Logic and systematic reasoning are key underpinnings of CT because computer programs are essentially an automation of our reasoning processes. Therefore, writing solutions requires a good understanding of logic and the ability to use proper notation over natural language. This chapter has covered the basics of inductive and deductive reasoning, as well as the importance of Boolean and symbolic forms of logic to computing.

Logic is an essential part of algorithms, which are the among the bedrock of computing. They are the core of any program and thus play a part in every computer-based solution. This chapter introduced the basic building blocks of every algorithm and the notion of state.

It's important in the case of both logic and algorithms to respect the need for a systematic approach. Although both can make intuitive sense on some level, it's also easy to make mistakes when considering them that way. The use of correct logical notation will help you to avoid making common mistakes.

EXERCISES

EXERCISE 1

Mark the following statements as true or false:

A. An inductive logical argument makes conclusions that are probable rather than certain.
B. So long as a deductive argument has true premises, then its conclusion is certain.
C. In Boolean logic, a logical expression can have a maximum of two propositions.
D. Logical AND has a higher precedence than logical OR.
E. $x = x + 1$ would be a valid expression in an algorithm.

EXERCISE 2

A recipe database stores the recipes in Table 2.9.

Users can search for recipes by entering search terms, which the database matches to tags and cooking times. Consider the following search terms and decide which recipes will be returned:

A. cooking time less than 20 minutes and not vegetarian;
B. includes chicken or turkey but not garlic;
C. doesn't include nuts.

Table 2.9 Recipes in the database

Name	Tags	Cooking time
Broiled chicken salad	Chicken, lettuce, gorgonzola cheese, lemon juice	15 mins
Holiday turkey	Turkey, rice, onion, walnuts, garlic	60 mins
Three-spice chicken	Chicken, ginger, cinnamon, garlic, green beans	30 mins
Lentil salad	Lentils, onion, peppers, walnuts, lettuce	20 mins
Garlic dip	Garlic, lemon juice, chickpeas, chicken broth	5 mins

EXERCISE 3

Here are some rules of thumb to apply when deciding which supermarket queue to join (assuming you want to spend as little time queuing as possible):

A. People in a queue who are carrying their items by hand take about one minute to be processed.
B. People carrying a basket take about two minutes to be processed.
C. People pushing a trolley take about five minutes to be processed for a half-empty trolley; ten minutes for a full trolley.
D. People in a self-service checkout are processed in about 80 per cent of the time of a normal checkout.

Express these rules of thumb as logical statements. Each statement should make a conclusion about the estimated queuing time.

EXERCISE 4

Take the logical statements from the previous question and incorporate them into an algorithm. The algorithm should take a queue as input, and should output the total estimated queueing time.

EXERCISE 5

Based on the algorithm for singing *99 Bottles of Beer*, write a pseudocode algorithm for singing *The 12 Days of Christmas*.