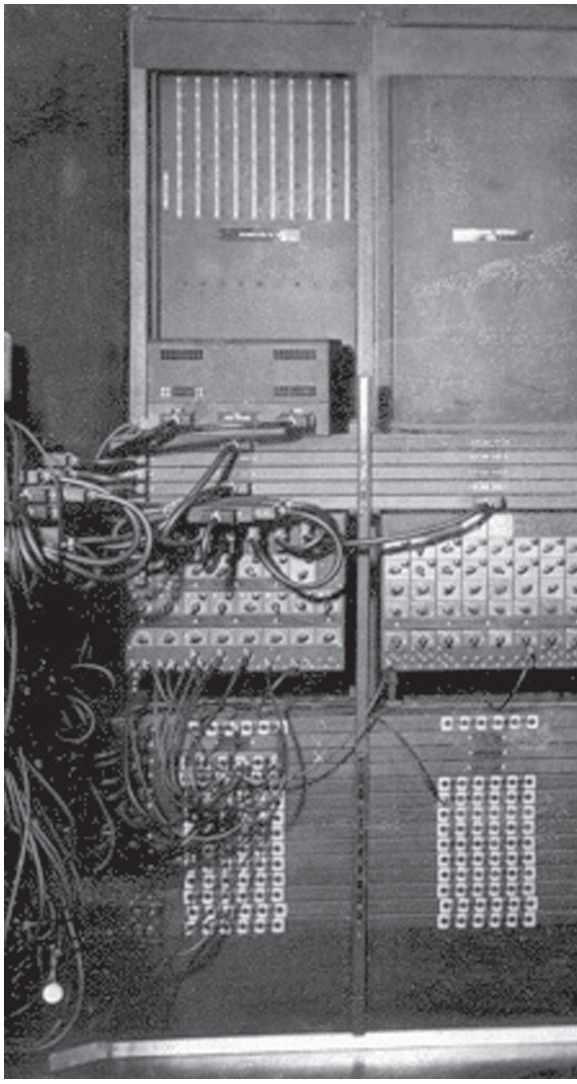
The real ghost in the machine

Martin Erwig introduces the algorithm, the sets of instructions that decide how we live



Algorithms are everywhere, and they affect everyone. As well as the obvious places – computers, smart phones, cars and appliances – they also occur outside of machines. A recipe is an example of such a machine-transcending algorithm. So are pieces of music, evacuation plans, the instructions on how to assemble furniture, card tricks and many other activities. If general interest in science and technology is not a sufficient reason to learn about them, maybe their omnipresence will persuade you.

Some may think that algorithms are complicated and only for the initiated few, an impression reinforced by stereotypes that plague computer science in the public imagination – think of Dennis from *Jurassic Park* or the Warlock from *Die Hard 4*. In films like these, computer science is a black art performed by nerdy guys. While this picture may contain an element of truth, it gives the false impression that computing is only for nerds and nothing could be further from the truth.

Algorithms were with us long before modern technology started to amplify their impact. They are simply methods for solving problems, and the need for solving problems is as old as humanity. Algorithms are independent of machines and can often be described in natural language or even pictures, which offers an opportunity for making algorithms accessible to a wider audience. In particular, we don't need to master a programming language to engage with this subject – just as we can explore music and learn an instrument without first having learnt music notation.

To understand algorithms, how they function, and why they have so much impact, it is helpful to know about the relationship between algorithms and computation, the need for using representations and expressing algorithms in some language, the requirement for algorithms to work for varying inputs, the questions of algorithmic correctness and efficiency and the limitations of algorithms.

Since an algorithm is just a plan for solving problems, it needs to be executed by an agent, be it a machine or a human, to exert its effect. When executed, an algorithm takes as input a representation of a problem and produces a solution. This process is called *computation*.

The use of a representation is crucial for an algorithm's ability to solve problems. Consider, for example, the string '10', which when interpreted as a decimal number represents the number ten, but when interpreted as a binary number represents the number two.

The transformation that adds a '0' at the right end means 'multiplying by ten' in the decimal representation, since '100' is the decimal representation of the number one hundred, whereas it means 'multiplying by two' in the binary representation, since '100' is the binary representation of the number four.

The separation of the symbols that are manipulated by a computer from their meaning is what ultimately enables problem-solving through computation, since it allows the encoding of problems in a form that is amenable to formal symbol transformation.

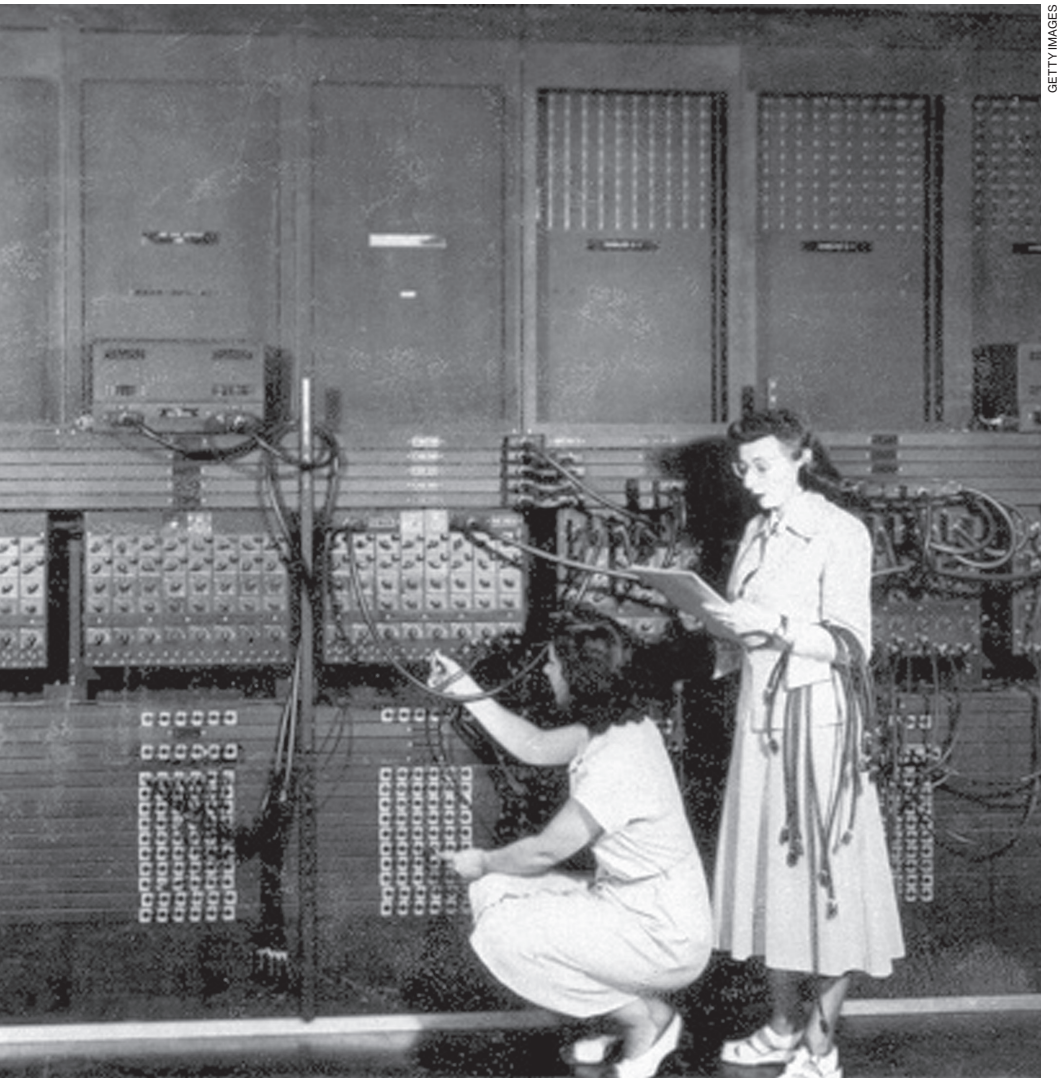
The separation of a computation from its description makes it possible to execute an algorithm by many different computers at different times and at different places. For this to work, however, an algorithm must be expressed in a language that is understood by the executing computer.

This separation of algorithm and computation makes algorithms independent of particular computers and allows them to endure over time and space. Whereas an individual computation is transient, an algorithm is eternal. This duality is similar to how genes represent proteins. Since genes can be copied, they can be put to use in many cells in parallel and at different times. It is imperative that an algorithm can solve a class of related problems and accept different inputs.

Consider, for example, an algorithm for sorting a deck of cards. It should work for any deck, no matter how many cards it contains and in what order.

A method that could multiply only two specific numbers or find the shortest path between two specific locations would be useless as an algorithm because the result could simply be remembered and stored, and there would be no need to ever execute the algorithm again.

Note that this requirement does not apply to algorithms that produce transient objects. Even if a chocolate cake recipe produces only one specific kind of cake, it is useful to execute it many times because the



GETTY IMAGES

ENIAC, the Electronic Numerical Integrator and Computer, developed at the University of Pennsylvania in 1946

cake gets eaten. This is different for abstract results such as numbers or paths, for which we can have our cake and eat it. Thus an algorithm must be able to process different inputs and solve different problems.

Two major challenges for algorithms are *correctness* and *efficiency*. It is not easy to get an algorithm to work correctly.

Consider the following algorithm for helping Hansel and Gretel find their way back home from the forest: 'Locate the closest pebble until you are home.'

This method sounds reasonable, but observe what happens after they have located the first pebble. One would assume that they should move on to the second pebble. However, if we interpret the instruction accurately, Hansel and Gretel should not move at all, since the closest pebble is the pebble they are currently at. The method takes for granted that previous pebbles won't be located again, but that is not specified in the algorithm. An algorithm must be unambiguous and must not rely on smart interpretations by the computer that executes it. This algorithm is not correct, because it does not terminate.

Algorithms require time to execute and space to store data. The runtime of algorithms is not measured in time but is given as a function that indicates how the number of required steps grows for a change in the size of its input. For example, the runtime of a linear algorithm grows proportionally with the size of the input, which is quite good. The runtime of a quadratic algorithm grows with the square of the input size, which is still fast enough in many cases but can be too slow in others.

On the other hand, algorithms with exponential runtime are practically unusable because their runtime doubles whenever the size of the input grows by a small constant. To get a sense of how bad exponential runtime is, consider the following fact. Linear, quadratic and exponential algorithms complete their work instantaneously for inputs of size 30. For inputs of size 100 the runtime of linear and quadratic algorithms is still not noticeable while an exponential algorithm would take about nine times the age of our universe to complete.

Exponential runtime is one major limitation of algorithms because it renders such algorithms unusable for all but small inputs.

For many optimization problems only exponential algorithms are known, which means that these problems can only be solved through approximations. It gets worse: many problems cannot be solved by algorithms at all. For example, the question of whether an algorithm terminates cannot be answered by an algorithm.

Algorithms are ubiquitous, and it is impossible to avoid them. Once developed, an algorithm is an intellectual contribution that can continually provide benefit, but only if it is correct and efficient.

A basic understanding of algorithms empowers us to better judge their risks and opportunities. For example, concerns about the loss of privacy in the face of Google and Facebook should not be attributed to an algorithm but rather raise questions about what data users disclose and what data those companies accumulate.

On the other hand, algorithms are increasingly employed in making decisions that have a huge impact on people's lives, such as approving loans or determining the length of prison sentences.

Recently, Ben Shneiderman, of the University of Maryland, has argued for the creation of a National Algorithm Safety Board to hold algorithms accountable and provide transparency to the public.

To understand reports and recommendations issued by such an agency one needs to have a basic understanding of algorithms. Knowledge of algorithms, their features as well as limitation, is necessary to appreciate their impact on the world.

Martin Erwig is Professor of Computer Science at Oregon State University and author of the book 'Once Upon an Algorithm: How Stories Explain Computing'