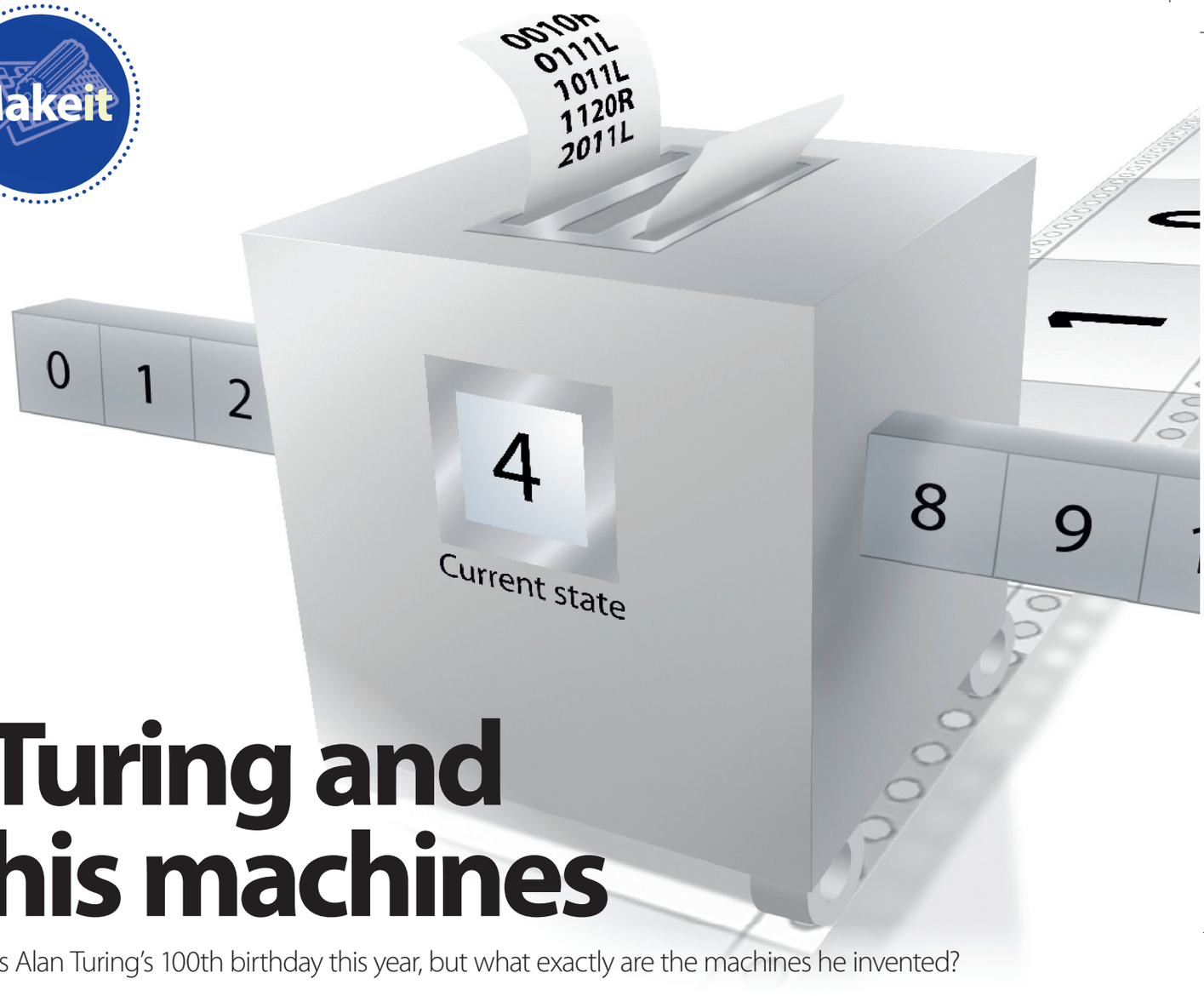


Turing and his machines

It's Alan Turing's 100th birthday this year, but what exactly are the machines he invented?

What's covered

▶ ALAN TURING'S LIFE AND LEGACY

We've been bombarded with news reports and articles on Alan Turing this year, the 100th anniversary of his birth, but what exactly is he famous for? What are Turing Machines, how do they work, and are they really the predecessors of the computers we use every day?

If you consider the greats of computing there would be two names at the top of the list: John von Neumann and Alan Turing. Both were mathematicians who became engrossed in the applied mathematics that is computer science and worked at a time when there were no computers that we'd recognise today.

Of the two, von Neumann is the one who's considered the founder of modern computers. He also designed the instruction set (the opcodes that performed the various operations like addition, testing values, jumping to instructions, and so on) for ENIAC – opcodes that are the basis of all current sequential programming languages.

Whereas von Neumann can be said to have had his feet firmly planted in the practical world of computer science, Turing was more embedded in an academic philosophical world. He described ideas and concepts with no thought of practical implementation, but that would help him prove theorems and conclusions in the world of mathematics that he inhabited.

Early life

Alan Turing was born in Maida Vale in London on 23 June 1912. From an early age he showed great mathematical promise – so much so that, when he attended Sherbourne School, his headmaster complained that his education would be deficient since he wasn't spending enough time with the classics.

From Sherbourne, Turing went to King's College, Cambridge in 1931, where he studied mathematics. He was awarded a first class honours degree in 1934. A year after graduating he was made a fellow of the same college, based on an original proof of the central limit theorem. In 1936, he published the first of the papers that would define him and his works: *On Computable Numbers*. We'll return to this paper and its fundamental invention, the Turing Machine, in a moment.

From 1936 to 1938, he studied under Alonzo Church, the inventor of the lambda calculus (now very much used in functional programming languages), in Princeton, New Jersey, earning his PhD. During this time he started to become interested in cryptography – work that would draw him to working at Bletchley Park during World War II, with particular focus on the German Enigma cryptography machine. He

▼ **State table for binary counting Turing Machine**

add	0	1	right	return
add	1	0	left	add
add	blank	1	right	return
return	blank	<no change>	no	start
return	0	<no change>	right	return
return	1	<no change>	right	return

Spotlight on...
Turing Machines and real computers

As we've seen, Turing Machines are abstract hypothetical beasts. No modern computer has ever been built as a Turing Machine, although many modern computers have been programmed to simulate such a machine. Even Lego Mindstorms kits are enough to simulate a Turing Machine.

Turing showed that, although a Turing Machine is simple to describe, it's powerful and capable enough to execute any computation a real computer can. The biggest difference is that, by definition, Turing Machines have infinite storage at their disposal with their tape. Real computers do not, although in practice useful Turing Machines do not need infinite tape at all.

Anything a Turing Machine can compute, a real computer can as well within memory limitations (and vice versa). This is usually expressed that the language being used to program the computer is Turing-complete. To be Turing-complete is a fairly low bar. In essence, the programming language can be used to simulate any other computing device (that is, it can be used to program a Universal Turing Machine). Generally, for an imperative programming language to be described as Turing-complete, it must have variables and conditional branching, which are enough to simulate a Turing Machine state table.

was instrumental in analysing the patterns produced by the Enigma Machine statistically and devised techniques based on those results that would help in cracking encoded messages. So advanced were the papers that resulted from these analyses, they remained secret until 2012.

After Bletchley Park

After the war, Turing worked on his own concepts for a computer that stored its own programs but, because of his top secret work at Bletchley Park, those papers were not widely known or acted on. In the late 40s, Turing moved to the University of Manchester, where he explored what could be meant by machine or artificial intelligence (again at a time when computers were very few and far between).

It was this work that led to the 1950 paper *Computing Machinery and Intelligence*. In this paper, Turing addressed the problem of defining whether a machine could be considered to think or not. The principal issue with this problem is the word 'think'. What is thinking and how do we recognise it? Can we construct an empirical test that conclusively proves that thinking is going on? From this discussion, he evolved the idea that became known as the Turing Test.

He approached the solution by considering a party game called the Imitation Game. In this game, a man and a woman go into separate rooms, so the rest of the guests don't know which is which. The guests send the subjects written questions and get typewritten answers back. From the series of questions and answers from each subject, the guests try to determine which is the man and which the woman. Of course, the subjects can try to muddy the waters by pretending to be each other.

From this basis, Turing then considered what would happen if the man or the woman were replaced with some kind of machine intelligence. Would the guests be able to guess the machine versus human more accurately than they would the man versus the woman in the original game? Turing's insight then was to rephrase the 'Can machines think?' question into a more general 'Can machines imitate how humans think?' This is the original Turing Test, although over the years it has morphed into a situation where we try to guess whether someone we're corresponding with is machine or human.

After this work, Turing moved on to mathematical biology as a field of study, but in 1952 he was charged with gross indecency for being homosexual. After being found guilty, Turing opted for a hormonal treatment to try to cure him of his homosexuality rather than be imprisoned. This course of treatment lasted for a year and rendered him impotent. The guilty verdict meant that he was stripped of his security clearance at GCHQ, which ended his work on cryptography. In June 1954, just over two weeks before his 42nd birthday, he was found dead from cyanide poisoning.

Since then, Turing's fame has grown, both from his work at Bletchley Park for GCCS during the war and from the two most popular of his many papers: the one that introduced the Turing Test, and the earlier one that introduced the Turing Machine. We've briefly seen the Turing Test, but what exactly is the Turing Machine?

The Turing Machine

Back in 1928, the German mathematician David Hilbert was trying to formulate a rigorous logical foundation for all of mathematics. In other words, given a small set of axioms (an axiom is a proposition that can be accepted as being true without proof), a mathematician could then build an edifice of theorems and their proofs that would cover all of mathematics. He believed this endeavour was feasible, but there were three big problems standing in his way, and this is how he stated them in 1928: was mathematics complete, was mathematics consistent and, finally, was mathematics decidable? He really wanted them to be true to help in his efforts to build mathematics as a self-consistent model. Kurt Godel comprehensively demolished Hilbert's first two aspirations with his incompleteness theorems in 1930 and the third – the Entscheidungsproblem or 'decision problem' – remained unsolved. So he posed it as a challenge for his fellow mathematicians: can you find an algorithm that, when given a set of axioms and an input statement in first-order logic, can prove the input statement is completely valid.

Both Alonzo Church and Alan Turing attacked this problem and both proved it false – Church by using his lambda calculus and some other results from Stephen Kleene (who is perhaps best remembered as the inventor of regular languages and expressions) and Turing by inventing a 'virtual' computer, one that existed only in the mind, the Turing Machine.

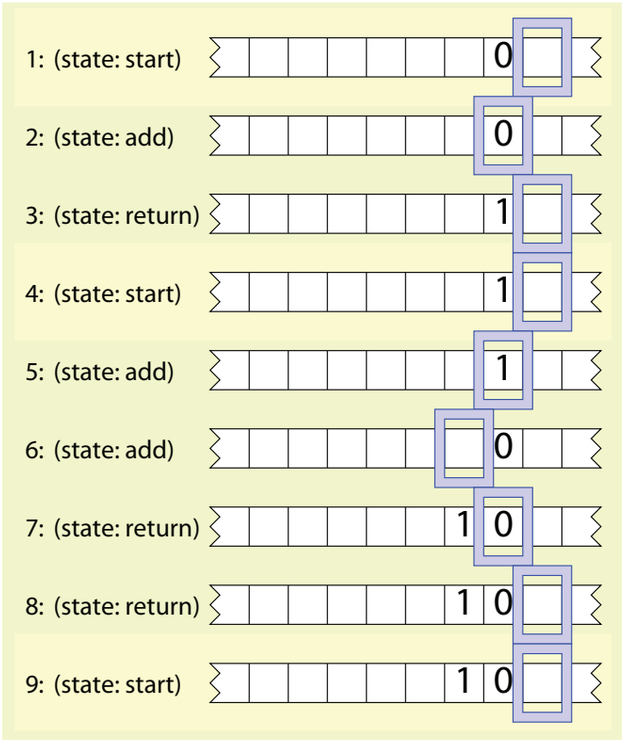
Turing approached the Entscheidungsproblem by ▶

Stopping the binary counter

The Turing Machine that counts in binary does not stop. It will continue to count ad nauseam. How then should we make it halt?

I can think of two ways. First: have the tape pre-loaded with a sequence of 0s instead of just the one. If we have four 0s in a row, we can make the machine stop after showing 17 binary numbers. Consider the add state: if the current cell is blank, we've reached the end of the initial list of 0s. Set that cell to 1, then invent a new 'returnthenstop' state that will do exactly what the return state describes, but when it reaches the blank on the right will set the new state to halt.

Second is to use an extra symbol and place it on the input tape to the left of the initial 0. Make the add state check for this symbol: if reached, output a 1 to the current cell, then use a new 'returnthenstop' state to work as described above.



▲ **The first few steps of a binary-counting Turing Machine**

Church-Turing thesis

One of the problems during the first half of the 20th century was properly defining the notion of computability. This is, in essence, the ability to devise an algorithm to solve a given problem efficiently. Efficiently here means that, on a fast enough computer, the algorithm will produce answers in a reasonable amount of time for given inputs.

The answer to a Travelling Salesman problem (one salesman has to visit N cities so that his cumulative travel is the shortest distance possible) can, for example, be shown to be extremely hard to compute if the number of cities is large, no matter how fast the computer.

The Church-Turing thesis shows that if some algorithm exists to efficiently calculate some value, the same calculation can be done with a Turing Machine, a lambda function, or by a recursively definable function. Computability doesn't depend on the expression of the algorithm using a particular process: if an algorithm is computable, it is computable in all ways.

defining what an algorithm is. He defined it as a mechanical process (do step A, do step B, repeat step C until something became true, and so on). Since a mechanical process is presumably something that can be done with a machine, Turing set about inventing a general machine that could be used to implement any such mechanical process or algorithm (not that Turing had any desire to build one).

The machine he invented consists of a device that operates on an infinite tape (or rather a tape that can be assumed to never end: there's always enough tape for the algorithm being processed). The tape is divided into cells, and in each cell there is a single symbol (with 'blank' being such a symbol). The device has a read/write head that can read the symbol in the current cell, write a symbol in the current cell, and move the tape one cell to either the right or left. Furthermore, the device has a special register – a memory location – that stores the current state of the machine, and there is a state table or transition table that defines what the machine does next. The state table takes as input the current state of the machine and the symbol under the head and defines what happens next: erasing or overwriting the current cell with another symbol, advancing the tape right or left (or leaving it where it is), and defining a possible new state. There is at least one state that stops or halts the machine and the computation.

Turing devised this hypothetical machine in such a way that its definition was obviously bounded. The symbols are assumed to come from a finite alphabet, the number of states is assumed finite as well, and the machine as a whole is well-defined (it can never get into a state that's unknown or undefined).

As you can appreciate, this is just a rather sophisticated state machine. Let's explore with an example: counting in binary. We'll start off with a tape with a single 0 and the head is positioned on the blank cell to its right. The state is 'start'. With the start state, no matter what is underneath the head, we move the head left one cell and set the state to 'add' because we're about to add one to the current binary number. In the add state, we look at the cell underneath. There are three possible values. If it is 0, overwrite it with 1, and set the state to 'return'. If it is 1, overwrite it with 0, move the head left one cell and keep the state as 'add'. If the cell is blank, set it to 1, and set the state to 'return'. Finally in the return state, if the cell underneath is either 0 or 1, move right one cell. If it's blank, don't move and set the state to 'start' again. In essence the return state is looking for the blank cell just to the right of the current binary number. Every time the machine hits the start state we can see the next successive binary number on the tape to the left of the head. Table 1 shows the state table and Figure 1 the Turing Machine in action.

The halting problem

Now that you've had a look at this particular example, I'm sure you can see that there's a big problem with it. Basically, it never stops. I didn't code into the state table any way to stop the binary counting and, with this simple example, it's easy to see that this is the case.

In fact, this is one of the main problems Turing considered with his hypothetical machine: given a Turing Machine and its initial input tape, is it possible by analysing the state table and the machine's symbol alphabet to determine whether the machine will halt or not? If you like, is there an algorithm you can use on a Turing Machine to determine, given some arbitrary input tape, that the machine will eventually stop? In computer science, this is known as the halting problem. It is a compliment to Turing's genius that he was able to prove that it's not possible to prove whether some arbitrary Turing Machine will halt or not given some input.

In fact, Turing went even further in discussing this problem. Since a Turing Machine is fully determined by its state table, alphabet, and input value, we can encode this information for a particular Turing Machine on a tape and give it to another Turing Machine. This second machine could read the input tape and execute the state table encoded on it and compute whatever the original machine was designed to compute. Let's call the encoded state table and input value a program and its data, and we can see we've hypothesised some arbitrary computer (admittedly with an infinite tape or storage) executing a program on some data. Will it halt or not? The answer is, we can't tell.

This second Turing Machine is what is now known as the Universal Turing Machine. It takes the definition of another machine and executes as if it were that other machine.

But let's suppose we have the algorithm Hilbert wanted. In other words, given some input statement, this algorithm could be followed mechanically (a Turing Machine) and output Yes or No as to the validity of the input statement. One such statement would be 'Here's some arbitrary Turing Machine, will it halt on every input?' Turing proved that the answer is no – no algorithm can decide whether a given Turing Machine will halt. In fact, the algorithm cannot even decide if it itself will halt or not in trying to decide if it will halt or not.

In his short life, Alan Turing provided some solid yet abstract ways to think about the world – ideas that are still widely known today. It's a shame he didn't live longer to improve them. ■