DIGITAL TECHNOLOGY PAST AND PRESENT

BBC MAKE IT DIGITAL

The Open University

To accompany the **BBC Make it Digital** season

DIGITAL TECHNOLOGY PAST AND PRESENT

Digital technologies have changed the way we work, shop, socialise, and are entertained. Behind them lie clever algorithms – step by step procedures that detect where we are and suggest a route, predict the weather, recognise our fingerprint, and do many other things. Algorithms pre-date computers by over 2000 years. For example, simple arithmetic procedures, like long division, are algorithms.

In this pack you will see how computation changed over time, from early calculating devices to modern computers, who are some of the women and men behind major breakthroughs, what are the fundamental concepts and the limitations of algorithms. You will also see how to translate algorithms into code, so that computers can execute them much faster and more precisely than humans ever could.

LEARNING WITH THE OPEN UNIVERSITY

The Open University (OU) has the UK's largest academic community with over 200,000 students, and with around 190 qualifications available in a range of fascinating and challenging subjects, you're sure to be inspired. We call our flexible study method 'Supported Open Learning' – it's different to other learning methods because it combines one-to-one support with flexibility, allowing you to fit study around your life. With us, you don't have to put your life on hold to get the qualification you need. Around 70 per cent of our students fit study around their job and busy, changing lives.

Beginning to study

Our Access modules have been specially designed to help you find out what it's like to study with the OU, get a taste for the subjects we offer, develop your study skills, build your confidence and prepare you for further study. You may even be able to study for free. To find out more visit **www.openuniversity.co.uk/ug-access**

Affordable education

Studying with the OU is more affordable than you might think. Depending on where you live we have a range of options to help make study more affordable. For example, if you have a household income (personal income if you live in Scotland) of less than £25,000 you may be eligible to study an Access module for free. For more information on this and all of the funding options available to you visit **www.openuniversity.co.uk/affordable**

If you want to take your interest in mathematics and computing further you may be interested in the following qualifications:

BSc (Hons) Mathematics (Q31)

Mathematics is indispensable to modern life. It enables us to predict the growth of markets, model airflow in a jet engine, calculate accurate drug doses and create 3D computer graphics. This degree will take your understanding of the concepts and theories of mathematics – and how they are applied in the real world – to an advanced level, and enhance your career prospects in a huge array of fields.

BSc (Hons) Computing and IT (Q63)

Computing and IT skills have become fundamental to the way we live, work, socialise and play. This degree course opens up the world of technology and an array of exciting career opportunities. It will help you to become a confident user and manager of information technologies, to administer and manage network or database systems, and to develop new software solutions to meet specific market or organisational needs.

For more information on these qualifications and the subjects you can study visit

www.openuniversity.co.uk/courses

Supporting you all the way

Whether you're at home, at work or on the move, your tutor, study advisers and other students are as close as you need them to be – online, on email, on the phone and face to face.

Whenever you log on, our forums are alive with people, and the opportunity to socialise doesn't stop there. Our students regularly get together, either to attend a tutorial or as part of a local study group.

Find out more

To discover more about studying at The Open University:

- visit www.openuniversity.co.uk/courses
- request a prospectus at www.openuniversity.co.uk/prospectus
- call our Student Registration & Enquiry Service on 0300 303 5303
- email us from our website at www.openuniversity.co.uk/contact

For information about The Open University's broadcasts and associated learning visit our website www.open.edu/openlearn/whats-on

The Open University has a wealth of free online information and resources about computing. To find out more visit www.open.edu/openlearn/makeitdigital

The Open University has a wide range of learning materials for sale, including self-study workbooks, DVDs, videos and software. For more information visit www.ouw.co.uk

Grateful acknowledgement is made to the following sources:

© AFP Photo/Eric Feferberg/Getty Images; © Alain BUU/Gamma-Rapho/Getty Images; © Ancient Art & Architecture Collection Ltd/Alamy; © Ann Ronan Pictures/Print Collector/Getty Images; © Austrian National Library; © Brinkstock/Alamy; © Chris Howes/Wild Places Photography/Alamy; © Chronicle/Alamy; © Classic Image/Alamy; © Cynthia Johnson/The LIFE Images Collection/Getty Images; © European Pressphoto Agency b.v./Alamy; © Everett Collection Historical/Alamy; © Fine Art Images/Heritage Images/Getty Images; © Geoffrey Kidd/Alamy; © GL Archive/Alamy; © GraphicaArtis/Getty Images; © Heritage Image Partnership/Alamy; © Hulton Archive/Getty Images; © Image Source Plus/Alamy; © INTERFOTO/Alamy; © iStockphoto.com/123dartist; © iStockphoto.com/adempercem; © iStockphoto.com/agsandrew; © iStockphoto.com/alengo; © iStockphoto.com/ayvengo; © iStockphoto.com/Erik Khalitov; © iStockphoto.com/nzphotonz, © iStockphoto.com/Pali Rao; © iStockphoto.com/simarik; © iStockphoto.com/Sirgunhik; © iStockphoto.com/Thomas Eye Design; © John Lund/Getty Images; © kubala/Alamy; © Lyroky/Alamy; © Maryam Mirzakhani/Courtesy of Stanford University; © Nick Higham/Alamy; © Nigel Tout/www.vintagecalculators.com; © Nils Jorgensen/REX; © Pictorial Press Ltd/Alamy; © Rex, © SSPL/Getty Images; © World History Archive/Alamy; © www.sliderulemuseum.com/Rose Vintage Instruments Ohio.

Every effort has been made to contact copyright holders. If any have been inadvertently overlooked the publishers will be pleased to make the necessary arrangements at the first opportunity.

Published in 2015 by The Open University, Walton Hall, Milton Keynes, MK7 6AA, to accompany the BBC Make it Digital season, Autumn 2015.

Broadcast Commissioner for the OU: Dr Caroline Ogilvie
Media Fellow for the OU: Mike Richards
Open University Make it Digital pack: Authors: Dr June Barrow-Green, Allan Jones, Chris Dobbyn, Dr Michel Wermelinger
Graphic Designer: Glen Darby
Broadcast Project Manager: David Bloomfield

Printed in the United Kingdom by Belmont Press.

The Open University is incorporated by Royal Charter (RC 000391), an exempt charity in England & Wales, and a charity registered in Scotland (SC 038302). The Open University is authorised and regulated by the Financial Conduct Authority.

MIX
Paper from responsible sources
FSC® C015185
FSC www.fsc.org

SUP 047583

HISTORY OF COMPUTERS

BBC MAKE IT DIGITAL

The Open University

Some key developments in the history of computers, from the 19th century to the present

COMPUTERS: THE IDEA

The idea of a computer was around long before one was made. Why?

The essential ideas relating to computers had been grasped during the 19th century by Charles Babbage, Ada Lovelace and others. But the creation of practical devices depended on electronics that came over a century later. With ever more sophisticated electronics and communications technology came faster, smaller and more versatile computers. As computers became ubiquitous and cheap, ways of merging them with communications technology led to innovations that could not have been foreseen.

LEARN MORE ABOUT coding and computers with The Open University. ***TU100 My digital life*** takes you on a journey from the origins of information technology through to the familiar computers of today, and on to tomorrow's radical technologies.

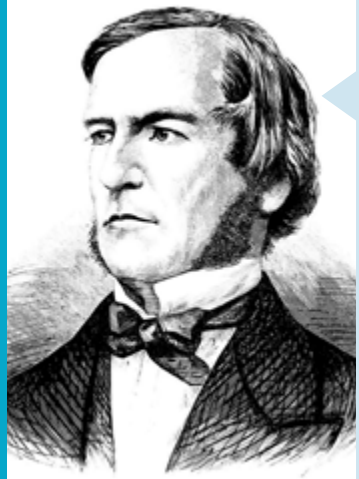
HISTORY OF COMPUTERS

BABBAGE'S ANALYTICAL ENGINE

1837

In 1837 the British mathematician Charles Babbage (1791–1871) outlined a design for a programmable, mechanical general-purpose computer called an Analytical Engine. Punched cards were to be used for programs and data. Apart from a small section, it was not constructed.

1837 1847



BOOLEAN ALGEBRA

1847

In 1847 the British mathematician George Boole (1815–1864) published *The Mathematical Analysis of Logic*. In this he used algebraic symbols to represent factual statements. Through algebraic manipulation, useful simplifications of the statements could be made. His algebraic symbols could take the values 0 and 1, representing false and true. Boolean algebra is widely used in digital electronics.

COMPUTABLE NUMBERS
1936

In 1936 the mathematicians Alan Turing (pictured) and Alonzo Church separately showed that there could not be a rule-based method (or algorithm) for establishing whether certain types of mathematical problem are solvable. As part of his solution to this long-standing puzzle, Turing conceived of a hypothetical programmable computing machine.

VON NEUMANN MODEL
1945

In 1945 the mathematician John von Neumann in the USA proposed a type of stored-program electronic computer in which program instructions were held in 'live' memory during operation, unlike earlier designs (such as Colossus) which were programmed by setting switches or configuring wired connections.

1936 1943 1945 1947

COLOSSUS
1943

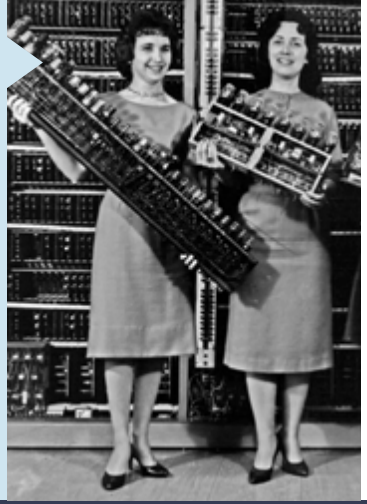
Colossus, the first programmable electronic digital computer, began working in December 1943 at Bletchley Park as part of the UK's wartime code-breaking operation. It was designed by the engineer Tommy Flowers, and partly funded by him as his superiors doubted its feasibility. Several were made, but they remained secret until the 1970s.

TRANSISTOR
1947

In 1947, the first transistors were developed at Bell Laboratories in the USA. They were miniature, low-power solid-state electronics devices for amplification and for switching. Within a few years they superseded the power-hungry and unreliable valves used as switching elements in the first computers.

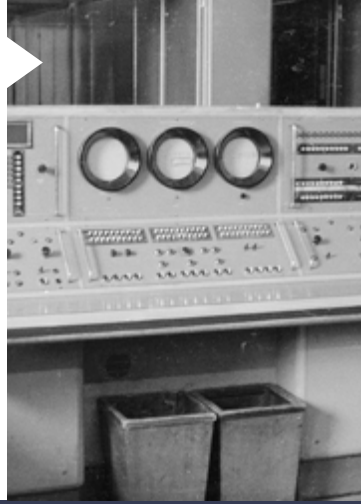
FIRST WORKING COMPUTERS
1949

The first stored-program digital electronic computers became operational in the period 1949–51 in several locations, and sometimes initially as prototypes to test feasibility. Examples include the EDSAC in Cambridge, the ACE at London's National Physical Laboratory, the Manchester University/Ferranti computers and the EDVAC in the USA.

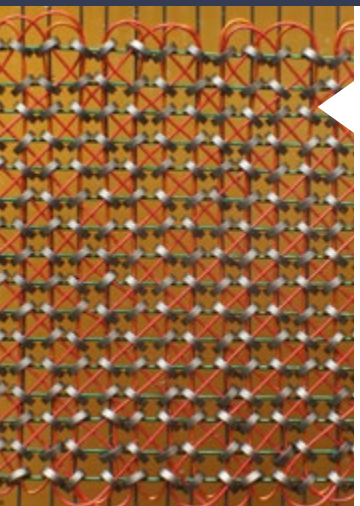


LEO COMPUTER
1951

The Lyons Electronic Office (LEO) was used for stock control and payroll in the Lyons chain of UK tea shops. It was the first computer used for a business application. LEO was based on Cambridge University's EDSAC computer. Computers based on LEO were used until 1981.

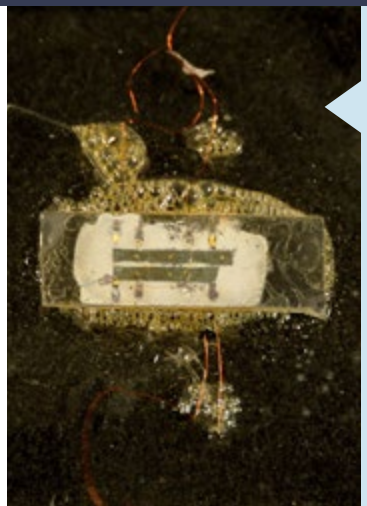


1949 1951 1951 1958



MAGNETIC CORE MEMORY
1951

Magnetic core memory, consisting of thousands of small rings of magnetic material, transformed the reliability of the random-access memory of computers, replacing temperamental technologies based on cathode-ray tubes or ultrasonic waves in mercury tubes. Magnetic core memory was dominant until the adoption of integrated-circuit memory in the 1960s.

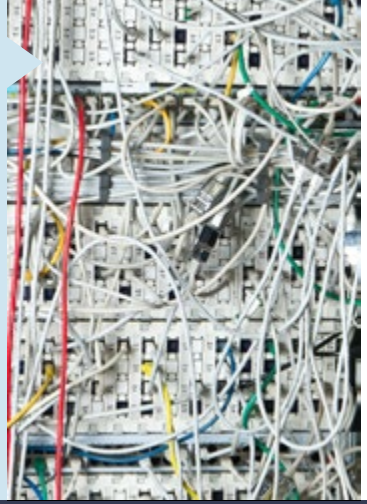


INTEGRATED CIRCUIT
1958

An integrated circuit contains all the components of a transistorised electronic circuit on a single, tiny piece of silicon. Integrated circuits are mass produced and individually very cheap. Typically they contain billions of components, and are widely used in electronic devices of all kinds, especially computers.

TCP/IP
1975

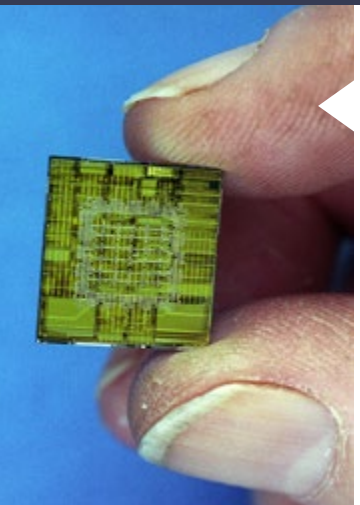
Transmission Control Protocol and Internet Protocol (TCP/IP) was developed to enable data communication between technically incompatible networks. It was developed in connection with the ARPANET project in the USA, and in 1975 a successful two-network test transfer took place between Stanford in California and London. TCP/IP is the basis of the internet.



PERSONAL COMPUTERS
1981

IBM model number 5150 was a desktop microcomputer that came to be known as a 'personal computer' (or PC). Its smallness and cheapness, and the existence of 'IBM compatible' accessories and software, led to its adoption in environments where computers had not been widely used. Apple computers soon followed.

1975 1981 1981 1989

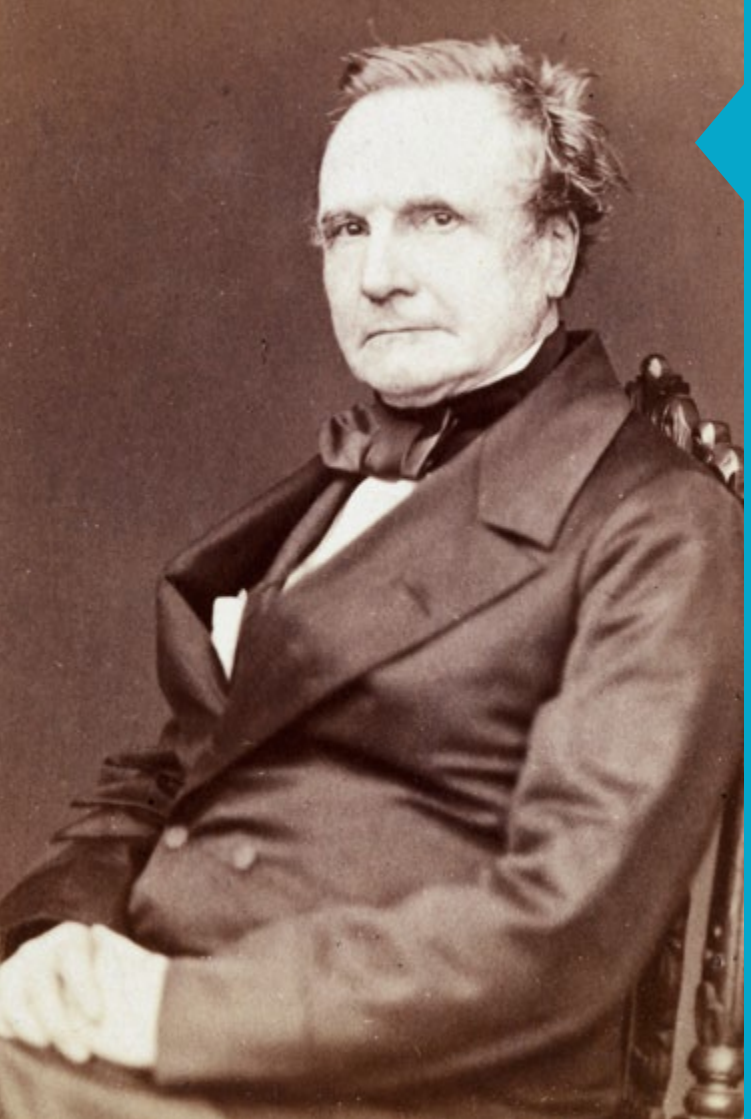


RISC COMPUTING
1981

Reduced Instruction Set Computing (RISC) uses a small set of processor instructions and highly optimised processors. It can outperform the more usual Complex Instruction Set Computing (CISC) used in conventional computers despite its limitations. It is widely used in mobile phones and tablets because of its power efficiency.

WORLD WIDE WEB
1989

In 1989 the British computer scientist Tim Berners-Lee, working at CERN, developed the World Wide Web. It facilitates the organisation, linking and display of information contained on dispersed computers joined by a local network or by the internet.



CHARLES BABBAGE

26 DECEMBER 1791 – 18 OCTOBER 1871

Charles Babbage was a British mathematician, engineer and inventor. He is now mainly remembered for designing two computational machines based entirely on mechanical processes, although he was active in many non-computational projects.

Babbage's first computational machine was the 'difference engine', conceived around 1822. It was a mechanised calculator rather than a computer, and was intended to help in the creation of mathematical tables such as logarithms. Such tables were normally compiled manually from human calculations and were notoriously inaccurate. Babbage's difference engine was never completed in his lifetime, although a modern version based on his later designs was made by the Science Museum in London between 1989 and 1991, and found to work as intended.

Babbage's second computational machine, the 'analytical engine', was described in 1837. It was closer to the modern idea of a computer than the difference engine was. 'Programs' were to be entered using punched cards of a kind already used in Jacquard looms in weaving. Babbage's design enabled loops and conditional branching to be followed during execution of a program. These are features of modern computer programs.

Funding problems and disagreements between Babbage and his engineer meant that only a small part of the analytical engine was made. Nevertheless, Babbage and Ada Lovelace described the programming of such a machine, which would have used a low-level language akin to a modern assembly language rather than a high-level language such as COBOL or Fortran.

ALAN TURING

23 JUNE 1912 – 7 JUNE 1954

The British mathematician and cryptographer Alan Turing envisaged a programmable computer during the 1930s as part of his solution to a problem posed by an earlier mathematician, David Hilbert, concerning the solvability of unsolved mathematical problems.

Hilbert had wondered whether a rule-based procedure (more precisely, an algorithmic procedure) could exist for determining whether mathematical problems were solvable. Turing, as part of his proof that no such procedure could exist, imagined a programmable general purpose 'computer' (not a term he used). The machine he envisaged was impractical, but this did not matter as his proof did not require that such a machine be built. However, he soon became interested in the practical realisation of his idea.

As a code-breaker at Bletchley Park during the Second World War, Turing was not directly involved with the Colossus computer designed and constructed by Tommy Flowers although Turing knew and admired Flowers's work. However, shortly after the war Turing designed the ACE computer for the National Physical Laboratory in London. Delays in the project led to his resignation from it, and his transfer to Manchester University, where he used the recently completed ACE computer until his early death in 1954.

Turing's contribution to computing was mostly theoretical. He also recognised the philosophical and ethical problems computers raised. Unlike many of his colleagues, he saw no reason in principle to distinguish between human intelligence and machine intelligence, although he did not pretend that the computer programs of his own time produced convincingly intelligent behaviour.

TIM BERNERS-LEE

8 JUNE 1955 – PRESENT DAY

The British computer scientist Tim Berners-Lee developed the World Wide Web. He developed his idea during his two periods of work at CERN, the European Organization for Nuclear Research in Geneva, Switzerland.

In his first period at CERN, in 1980, Berners-Lee sought to improve the organisation of information in an institution where staff turnover was high, where many projects were being pursued and updated simultaneously, and where computer use was centralised but open to multiple users. He developed a form of hypertext (which already existed) so that parts of documents on the system could link to parts of other documents on the same system. He considered this direct linking a better form of organisation than using hierarchies, trees or keywords.

During his second period at CERN from 1984, Berners-Lee developed his hypertext system to work between networked computers on different, and possibly widely separated networks. For this he married his hypertext system with internet protocol, so that addresses of remote computers could be embedded in hypertext links. Users could then move seamlessly between links in documents held on different networks in different locations. Berners-Lee also created the first internet browser.

A measure of Berners-Lee's success is that many users mistakenly think the World Wide Web is the same as the internet. He declined to patent his invention or to require royalties, believing the World Wide Web should be freely available.

WOMEN IN MATHEMATICS AND COMPUTING

BBC MAKE IT DIGITAL

The Open University

Women who dared to go against the flow, and their achievements in mathematics

THE STORY SO FAR

Ever wondered why stories about mathematicians always seem to be about men? Is it because men are better at mathematics than women? Absolutely not.

It's because until very recently society dictated that it wasn't very respectable for women to be mathematicians. Unfair as it was, it was very difficult for a woman to make herself heard and to be accepted by other mathematicians. It just wasn't the done thing in polite society. But there were a few women who dared to go against the flow.

LEARN MORE ABOUT mathematics with The Open University. ***MU123 Discovering mathematics*** provides a gentle start to the study of mathematics. It will help you to integrate mathematical ideas into your everyday thinking.

WOMEN IN MATHEMATICS AND COMPUTING

HYPATIA
c.300 – 415

Hypatia is the only woman mathematician of significance whose name has come down to us from Antiquity. Although none of her work survives, she is reported to have written commentaries on Apollonius's *Conics*, Ptolemy's *Almagest*, Diophantus's *Arithmetic* and Archimedes' *Measurement of the Circle*. For her prominent identification with learning, she was hacked to death by a Christian mob.

c.300 **1706**

ÉMILIE DU CHÂTELET
1706 – 1749

The intellectual and scientist Gabrielle Émilie Le Tonnelier de Breteuil, marquise du Châtelet, knew and was respected by many of the leading French mathematicians of the day. She collaborated scientifically with Voltaire, who was one of her lovers and her long-time companion, and her masterful translation into French of Newton's fiercely difficult *Principia Mathematica*, which included a detailed commentary of her own, remains unsurpassed.

MARIA AGNESI
1718 – 1799

Maria Gaetana Agnesi was the first woman mathematician of the modern period. By the age of eleven she had mastered several languages, including French, Latin, Greek, German, Spanish and Hebrew. Her mathematical fame rests on her two-volume *Instituzione Analitiche ad Uso della Gioventù Italiana* (1748–9), one of the earliest treatments of the calculus.

MARY SOMERVILLE
1780 – 1872

Mary Somerville was a Scottish science writer whose *Mechanism of the Heavens* (1831), a popularisation and translation of Laplace's celebrated but somewhat impenetrable *Mécanique Céleste*, made her famous. Laplace, who praised her interpretation, claimed she was one of the few people who understood his work. She was the first woman to have experimental results published by the Royal Society.

1718 **1776** **1780** **1815**

SOPHIE GERMAIN
1776 – 1831

As a woman Sophie Germain was barred from studying at the École Polytechnique so she adopted a male pseudonym and corresponded with the lecturers. In 1816 she became the first woman to win a Paris Academy Prize with her work on elasticity. Her best work was in number theory where she made important contributions to the proof of Fermat's Last Theorem.

ADA LOVELACE
1815 – 1852

Ada Lovelace was the daughter of the famous poet Lord Byron, though she never met her father. She was taught mathematics by Mary Somerville and by Augustus De Morgan. Through her family and friends she met several influential mathematicians and scientists, one of whom was Charles Babbage.

FLORENCE NIGHTINGALE
1820 – 1910

The founder of modern nursing, Florence Nightingale, was a pioneer of applied statistics. Her studies of the 1850s on mortality statistics, arising from her experience in the Crimean War, including her introduction of the polar-area diagram, led eventually to reforms in ward hygiene and hospital design.

EMMY NOETHER
1882 – 1935

Emmy Noether was one of the most talented and creative women mathematicians of the 20th century and maybe of all time. One of the founders of modern abstract algebra, she was dismissed from the University of Göttingen by the Nazis in 1933, and died following surgery in the USA at the height of her creative powers.

1820 1850 1882 1900

SOFIA KOVALEVSKAYA
1850 – 1891

In 1868 Sofia Kovalevskaya engaged in a 'fictitious' marriage so she could leave Russia to study mathematics in Germany. With her appointment at the University of Stockholm in 1883, she became the first professional female mathematician. In 1888 her work on the mathematics of a rotating body won the Prix Bordin of the Paris Academy.

MARY CARTWRIGHT
1900 – 1998

A student of GH Hardy, Mary Cartwright made important contributions to the mathematics of chaos in collaboration with JE Littlewood. In 1947 she was the first woman mathematician to be elected Fellow of the Royal Society, and in 1961–2 she was the first woman president of the London Mathematical Society.

GRACE HOPPER
1906 – 1992

Grace Murray Hopper was an American computer scientist and a United States Navy rear admiral. During 1951–2, she led the team that constructed the first compiler for a computing programming language. This compiler was a precursor to COBOL (Common Business-Oriented Language) which was designed in 1959 and became widely adopted due to Hopper's influence.

JULIA ROBINSON
1919 – 1985

Julia Robinson made a remarkable contribution to the solution of Hilbert's 10th Problem, a problem concerning the existence of solutions of a certain type of equation. In 1975 she became the first woman to be elected to the United States National Academy of Sciences, and in 1983 she became the first woman president of the American Mathematical Society.

1906 1906 1919 1977

OLGA TAUSSKY-TODD
1906 – 1995

In 1937 the Austrian mathematician Olga Taussky arrived in London, having left Göttingen due to political unrest. There she met and married the mathematician Jack Todd. During the war she worked on problems arising in flutter analysis of supersonic aircraft. Her most influential work was in matrix theory but she also made important contributions to number theory.

MARYAM MIRZAKHANI
1977 – Present day

Maryam Mirzakhani is an Iranian mathematician working at Stanford University. While at school, she won a gold medal in the Mathematical Olympiad. In 2014 she became the first woman to win a Fields Medal, the highest honour a mathematician can achieve, for her 'striking and highly original contributions to geometry and dynamical systems'.

ADA LOVELACE

10 DECEMBER 1815 – 27 NOVEMBER 1852

Augusta Ada King, Countess of Lovelace, was born Augusta Ada Byron and is now commonly known as Ada Lovelace. She was an English mathematician and writer chiefly known for her work on Charles Babbage's early mechanical general-purpose computer, the Analytical Engine. Her notes on the engine include what is recognised as the first algorithm intended to be carried out by a machine. Because of this, she is regarded as the first computer programmer.

Ada was born 10 December 1815 as the only child of the poet Lord Byron and his wife Anne Isabella Byron. All Byron's other children were born out of wedlock to other women. Byron separated from his wife a month after Ada was born and left England forever four months later, eventually dying of disease in the Greek War of Independence when Ada was eight years old. Ada's mother remained bitter towards Lord Byron and promoted Ada's interest in mathematics and logic in an effort to prevent her from developing what she saw as the insanity seen in her father, but Ada remained interested in him despite this (and was, upon her eventual death, buried next to him at her request).

Ada described her approach as 'poetical science' and herself as an 'Analyst (& Metaphysician)'. As a young adult, her mathematical talents led her to an ongoing working relationship and friendship with fellow British mathematician Charles Babbage, and in particular Babbage's work on the Analytical Engine. Between 1842 and 1843, she translated an article by an Italian military engineer.

WOMEN IN WW1

28 JULY 1914 – 11 NOVEMBER 1918

On 9 September 1915, Adelaide Davin, a computer operator in Karl Pearson's statistical laboratory at University College London, wrote to Pearson:

"I was coming home in a tram just before 11 o'clock when the driver called out that there had been a Zep, and that it had been fired at twice – then the tram stopped, and the lights went out, whereupon several women began to shriek. I got out walked home to find all the neighbours in the street gazing heavenwards. Nobody obeyed the instructions to seek shelter. We could see the flashes of the anti-aircraft guns, but they all went very wide of the mark."

In WW1 combat in the air was a new phenomenon and the theory and practice of anti-aircraft gunnery was in its infancy. Clearly improvement in anti-aircraft systems was a matter of urgency. However, the creation of the necessary high-angled range tables was no trivial task and involved complicated mathematics, extensive computations and good draughtsmanship. The mathematical theory and data collection were done by mathematicians based on board HMS Excellent at Portsmouth and at Woolwich, while the computations and production of the tables were done, much of it by women, in Pearson's laboratory. Women were employed as computers elsewhere as part of the war effort, but Pearson's aim "to maintain a body of trained computers together who would have the force of character and knowledge to meet new problems" provided those in his laboratory with a unique opportunity.

SOFIA KOVALEVSKAYA

15 JANUARY 1850 – 10 FEBRUARY 1891

The Russian mathematician Sofia Vasilyevna Kovalevskaya made important contributions to the theory of differential equations, mathematical analysis and mechanics. She was the first woman in modern Europe to hold a PhD in mathematics, to hold a professorship in mathematics and to be an editor of a scientific journal. She was also a supporter of women's rights, a champion of radical causes, and an accomplished writer.

Sofia was able to study mathematics as a child but in 1868, as Russian universities were closed to women, she engaged in a 'fictitious' marriage with Vladimir Kovalevskij, a paleontology student, so she could emigrate. She first studied in Heidelberg before, in 1871, moving to Berlin to study with the great Karl Weierstrass, although she was not allowed to attend lectures at the university. In 1874 she submitted three papers to the University of Göttingen, including one on the rings of Saturn, and was awarded her PhD summa cum laude. Meanwhile she had visited London and attended George Eliot's salon.

Sofia returned to Russia with Vladimir in 1874 and in 1878 their daughter was born. In 1881 she left Vladimir and went to Paris to continue with her mathematics. Vladimir committed suicide in 1883, and in the same year Sofia was appointed to a temporary position at the University of Stockholm. In 1888 she won the prestigious Prix Bordin of the Paris Academy for her work on the mathematics of a rotating body. In 1889 she was appointed to a permanent professorship in Stockholm. Two years later she died unexpectedly of pneumonia.

BASICS OF ALGORITHMS

BBC MAKE IT DIGITAL

The Open University

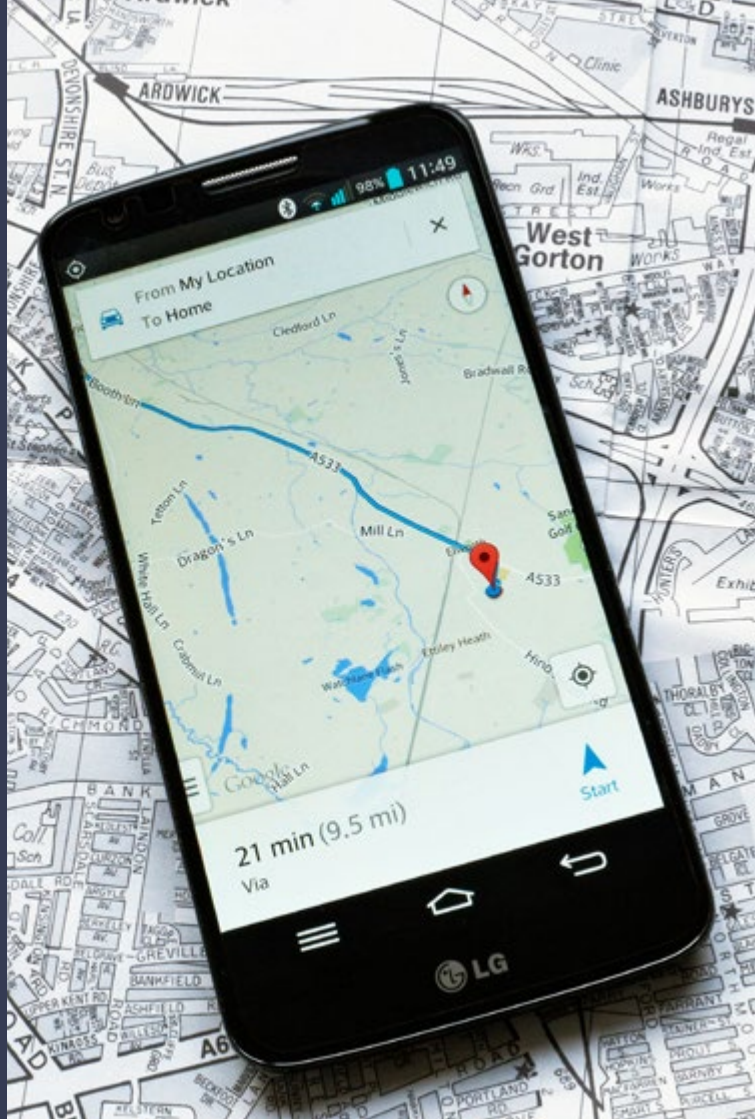
How algorithms work: step by step recipes for everyday life

ALGORITHMS MAKE THE WORLD GO AROUND

An algorithm is a precisely defined step by step procedure that, given some input, will produce the desired output. Whenever you give directions to someone, you are constructing an algorithm with the start location as input and the destination as output. It's a very restricted algorithm: it works only for those start and end points.

A sat nav device has a general algorithm that given any origin and destination (the input) will find the route (the output) from one to the other, according to the map it has. The route is itself an algorithm ('turn left in 200m', etc.) for the driver to execute.

The modern world cannot function without algorithms. Shopping, entertainment, scientific discovery, transportation, and many other things all rely on sophisticated algorithms to process payments, buy stock, analyse DNA, stream video, recognise licence plates, and so on.



THE BASIC INGREDIENTS

An algorithm is a combination of sequences of steps (one after the other), repetition of steps, and choices between steps, depending on some conditions.

Euclid's algorithm

One of the oldest algorithms known was described by Greek mathematician Euclid in c. 300BC. It takes two positive integers (like 1, 2, 3, etc.) and produces their greatest common divisor (GCD), the largest number that divides both without a remainder.

```
Let M and N be two positive integer numbers
while M and N are different:
   if M > N then:
      let M be M - N
   otherwise:
      let N be N - M
the GCD is N (or M because they're the same)
```

The algorithm is a sequence of three steps:

1. Get the input
2. Compute the GCD
3. Show it

The computation is a repetition of a two-way choice, each one being a single step that makes the larger number smaller.

Example of Euclid's algorithm

M	N	STEPS
21	49	M (21) is not more than N (49) so N changes to N - M = 49 - 21 = 28
21	28	M (21) is not more than N (28) so N changes to N - M = 28 - 21 = 7
21	7	M (21) is more than N (7) so M changes to M - N = 21 - 7 = 14
14	7	M (14) is more than N (7) so M changes to M - N = 14 - 7 = 7
7	7	M and N are the same so GCD equals 7

All algorithms impose conditions on what input is acceptable to them. In this case both numbers must be positive integers, to guarantee that the repetition will stop.

SEARCHING: TWO RECIPES

Some algorithms solve very specific problems, others are general and are used in a variety of contexts – for instance search algorithms where we want to know if a word occurs in a text.

Linear search

This simplest algorithm goes through each word in the text and compares it to the search word. If they're the same, the search stops and it reports success (the word has occurred in the text). If it gets to the end of the text without finding the same word, it reports failure.

Search word - "the".

Text - "to be or not to be in this text that is the question"

to	be	or	not	to	be	in	this	text	that	is	the	question
1. No	2. No	3. No	4. No	5. No	6. No	7. No	8. No	9. No	10. No	11. No	12. Yes	
											Success	

The search word is the 12th word of the text, so linear search makes 12 comparisons before reporting success.

The worst case for a linear search is when the searched word occurs at the end of the text or not at all, because all the words the document will be compared before reporting success or failure. Even removing duplicate words beforehand from the text doesn't improve things much (10 instead of 12 comparisons).

Binary search

A faster algorithm, that does fewer comparisons to report success or failure, is binary search. The input must be ordered: in this case the text's words appear in dictionary order.

The search word is compared to the word in the middle of the document. If they're the same, the search stops. If the search word comes alphabetically before the middle word, then the binary search continues on the left half of the document, otherwise on the right half. This is repeated until the word is found (success) or the part of the document to be searched is empty (failure).

be	in	is	not	or	question	text	that	the	this	to
					1. No			2. Yes		
								Success		

Binary search required only 2 comparisons to find the word. After each comparison the search space (the list of words) is halved, making the algorithm much faster. It is an example of a **recursive** algorithm, which is applied in the same form to ever smaller inputs.

AN ALGORITHM'S BEST FRIEND

How the data are structured goes hand in hand with how the algorithm works. Often the key to an efficient algorithm is an efficient data structure.

Hash table

A hash table, a widely used data structure, is a table of key and bucket pairs. Each bucket contains all the data with that key. The key could be the length of a word, and thus each bucket only contains words of the same length.

Given a search word, the algorithm computes its length and then searches only the corresponding bucket.

The search for "the" still requires two comparisons, but other searches become faster. Looking for "today" reports failure after zero comparisons with hash search because there is no 5-letter word bucket. It would take 11 comparisons with linear search and 4 with binary search.

Hashing is an example of a **divide-and-conquer** strategy. In this case the text's words are divided into buckets so that each search is performed on a single bucket.

The postcode is an example of a hash key, the bucket being all addresses with the same postcode, thus helping to sort and deliver post more quickly.

Key (number of letters in the word)	**Bucket (words of the same length)**				
2-letter words	be	in	is	or	to
3-letter words	not	the			
	1. No	2. Yes Success			
4-letter words	text	that	this		
8-letter words	question				

Search	**Comparisons**			**Result**
	Linear	**Binary**	**Hash**	
the	12	2	2	Success
not	4	3	1	Success
today	11	4	0	Failure

BRUTE FORCE: THE SURE BUT SLOW WAY

A word processor can quickly search for any sequence of characters (a **string**). It is impractical for a word processor to construct a hash table of all strings occurring in the document, memorising each string's position in the text, and to change the information constantly as the document is edited. A different algorithm is needed.

Brute force approach

This approach tries all possible matches systematically, like the linear search. To search for the string 'quest' the 1st character of the string is compared to the 1st character of the text. If they are the same, the 2nd string character is compared to the 2nd of the text, and so on, until the whole string matches or one comparison fails. Following a failure the string is shifted right by one character and the comparisons re-start.

	t	h	a	t		i	s		t	h	e		q	u	e	s	t	i	o	n	
1	**q**	u	e	s	t																Failure
2		**q**	u	e	s	t															Failure
3			**q**	u	e	s	t														Failure
…																					
12												**q**	u	e	s	t					Failure
13													**q**	u	e	s	t				Match
14													**q**	**u**	e	s	t				Match
15													**q**	**u**	**e**	s	t				Match
16													**q**	**u**	**e**	**s**	t				Match
17													**q**	**u**	**e**	**s**	**t**				Success

17 comparisons are made:
12 mismatches of 'q' (first letter of string) followed by matching the 5-letter string.

THINKING OUTSIDE THE BOX

A different perspective can sometimes lead to a more efficient algorithm.

The Boyer–Moore algorithm

In 1977, computer scientists Boyer and Moore had three key insights into the problem.

1. If the whole string matches, so must the last character. Hence we can do comparisons in backwards order.
2. If the match fails because the character in the text does not occur in the string, we can shift the string by its whole length.
3. If the match fails but the text character occurs in the string, the string is moved enough positions so that the text character matches the corresponding string character.

'**t**' (the last letter of the string) is compared against a space, which doesn't occur in the string and the whole string shifts to the right.

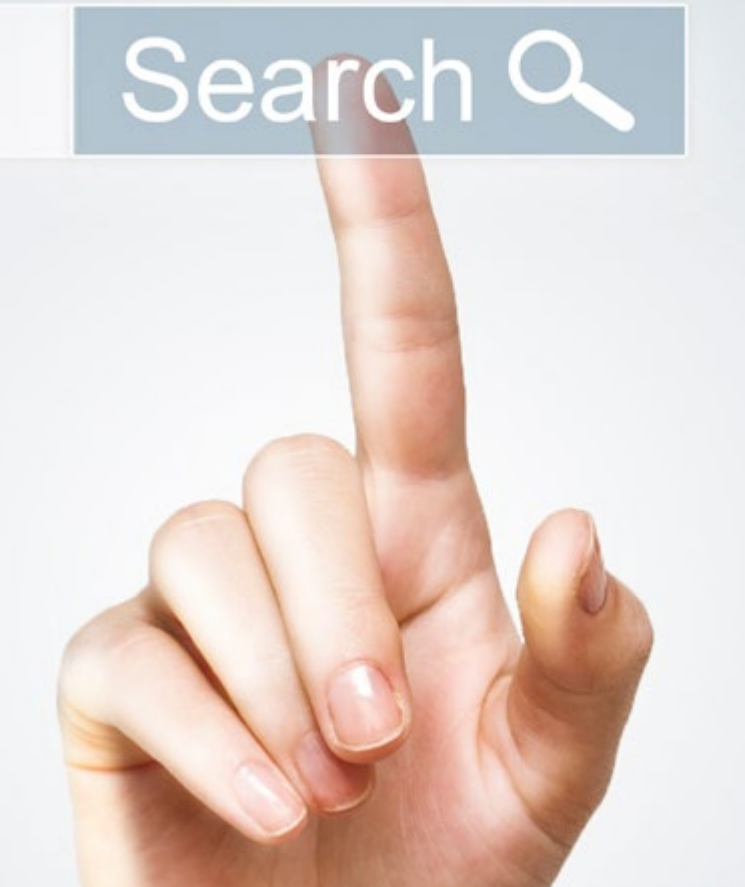
'**t**' is compared with 'h', which also doesn't occur in the string.

'**t**' is compared with 'e', which is the 3rd letter of the string and we therefore shift the string to align the 'e's.

'**t**' is compared, which matches, and then in turn each letter is matched backwards until all match.

8 comparisons are made, less than half used by brute force.

	t	**h**	**a**	**t**		**i**	**s**		**t**	**h**	**e**		**q**	**u**	**e**	**s**	**t**	**i**	**o**	**n**	
1	q	u	e	s	**t**																Failure (no space in word)
2						q	u	e	s	**t**											Failure (no 'h' in word)
3											q	u	e	s	**t**						Failure (e in word so move to e)
4													q	u	e	s	**t**				Match
5													q	u	e	**s**	**t**				Match
6													q	u	**e**	**s**	**t**				Match
7													q	**u**	**e**	**s**	**t**				Match
8													**q**	**u**	**e**	**s**	**t**				Success



SUMMING UP

Great algorithms are worth millions to companies. Google became leader of search engines, attracting millions in adverts, due to its ranking algorithm, while Amazon, Netflix and others use recommendation algorithms to attract further business. Creating good algorithms requires both creativity and technical expertise, but we all occasionally think algorithmically, e.g. when planning how best to organise a trip.

We have looked at some key algorithmic concepts and strategies that you can apply in everyday problem solving. All algorithms use **sequence**, **repetition** and **choice** of steps. **Brute force** explores all potential solutions until finding one. **Divide-and-conquer** partitions the solution space to work on smaller sub-problems. Divide-and-conquer algorithms are often **recursive**, i.e. they repeat themselves. Algorithms work only for the input they expect (e.g. two positive integers). Good algorithms use to their advantage what is known about the input (e.g. that the input is ordered). Algorithms depend heavily on how data are organised. Finding the appropriate **data structure** often helps find an efficient algorithm.

LEARN MORE ABOUT algorithms and data structures with The Open University. ***M269 Algorithms, data structures and computability*** will help you become a computational thinker, exploring a range of computing concepts and applying these to a variety of problems.

THE LIMITS OF ALGORITHMS

BBC MAKE IT DIGITAL

The Open University

What are the limitations of algorithms and the challenges of writing them?

ALGORITHMS EVERYWHERE

Whether we are aware of it or not, algorithms dominate our lives today. From online retailing to share trading, from drug design to driverless cars, from online dating to taxi services, from setting insurance premiums to surveillance – algorithms have exploded into every corner of modern life.

All this raises a serious question: it seems that algorithms can do everything, but are there things that they can't do? What are their limitations, if any?

One way to think about this is to consider the efficiency of an algorithm. There is nothing mysterious about the concept itself: it is simply a recipe – a set of steps which, if followed, solve a problem. So it is usually easy enough to write an algorithm. What is much harder is to write an efficient algorithm.

WHAT ALGORITHMS CAN'T DO – UNCOMPUTABILITY

Some problems can take far too long to solve. But are there problems that can't be solved at all by an algorithm?

The Turing machine

In 1936, the great mathematician Alan Turing succeeded in proving an abstract mathematical problem known, dauntingly, as the Entscheidungsproblem. His proof was based on the idea of a hypothetical machine, now known as the Turing machine, which can simulate any algorithm, regardless of its complexity.

The machine has a register that indicates what state it is currently in, and an infinitely long tape divided into cells, each cell containing a symbol, 1, 0 or blank.

The Turing machine's head (shown in grey) is able to read the symbol in the cell under it, write a symbol to that cell, and shift the tape one cell to the left or right.

The machine carries out the following steps:

1. Read the symbol under the head.
2. Get the current state from the register.
3. Consult a table of instructions to decide what to do next.
4. Execute those instructions and move to a new state.

For the following (partial) example of a table

State	Head reads	Head writes	Tape shift	Move to State
1	Blank	None	Left	2
1	0	Write 1	Right	2
1	1	Write 0	Right	3
2	Blank	Write 0	Right	1

If the machine is in State 1, and 0 is under the head, then it writes 1 to the tape, shifts the tape right one space, and moves to State 2. It goes on repeating Steps 1–4 until it stops.

Turing used the Turing machine to prove that certain problems were undecidable and could never be solved by any algorithm.

MEASURING ALGORITHMS

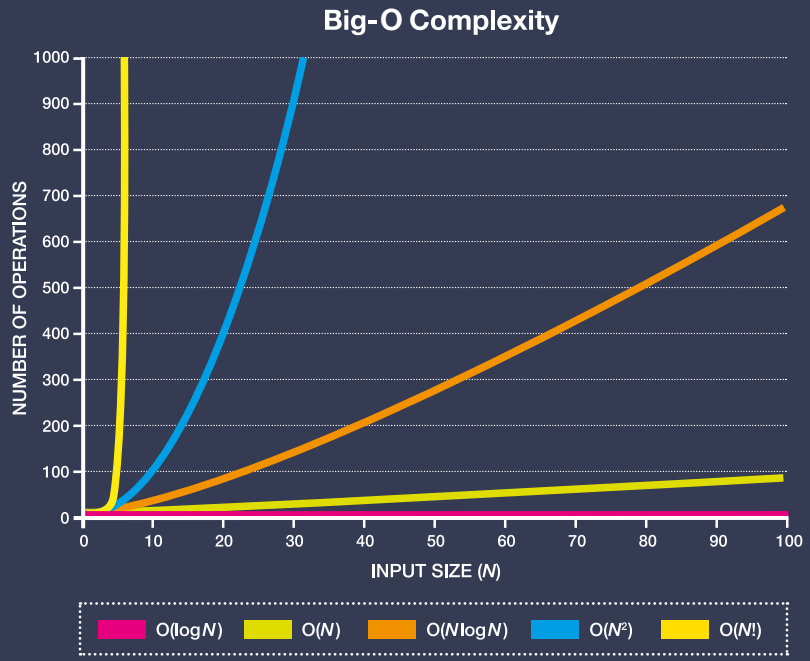
Algorithms can be measured by how many operations they perform and how long it takes to do those operations.

Big-O analysis

It is futile to try to compare algorithms directly by timing the actual speeds at which they run. Different computers will yield different timings on exactly the same algorithms. However, computer scientists have developed a way of estimating and comparing the efficiency of algorithms. It is known as complexity analysis or Big-O.

Big-O analysis compares the efficiency of algorithms in terms of how the number of operations they have to do (and thus how long they will take to finish) increases as the size of their input grows.

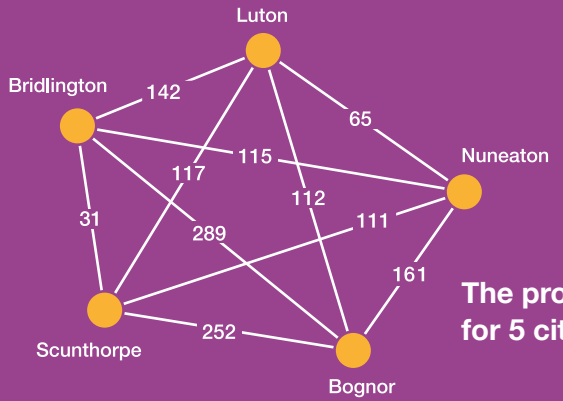
We can use Big-O analyses to compare the efficiency of algorithms. This is illustrated graphically by plotting the number of operations against input size *N*. The figure shows plots of algorithms of various complexity.



This shows the O(*N*) algorithms' workload increases steadily, while O(log *N*) algorithms grow much more slowly. Algorithms' workload in the important class $O(N^m)$ (e.g. $O(N^2)$ – known as polynomial algorithms) increases very rapidly, whereas that for O(*N*!) skyrockets.

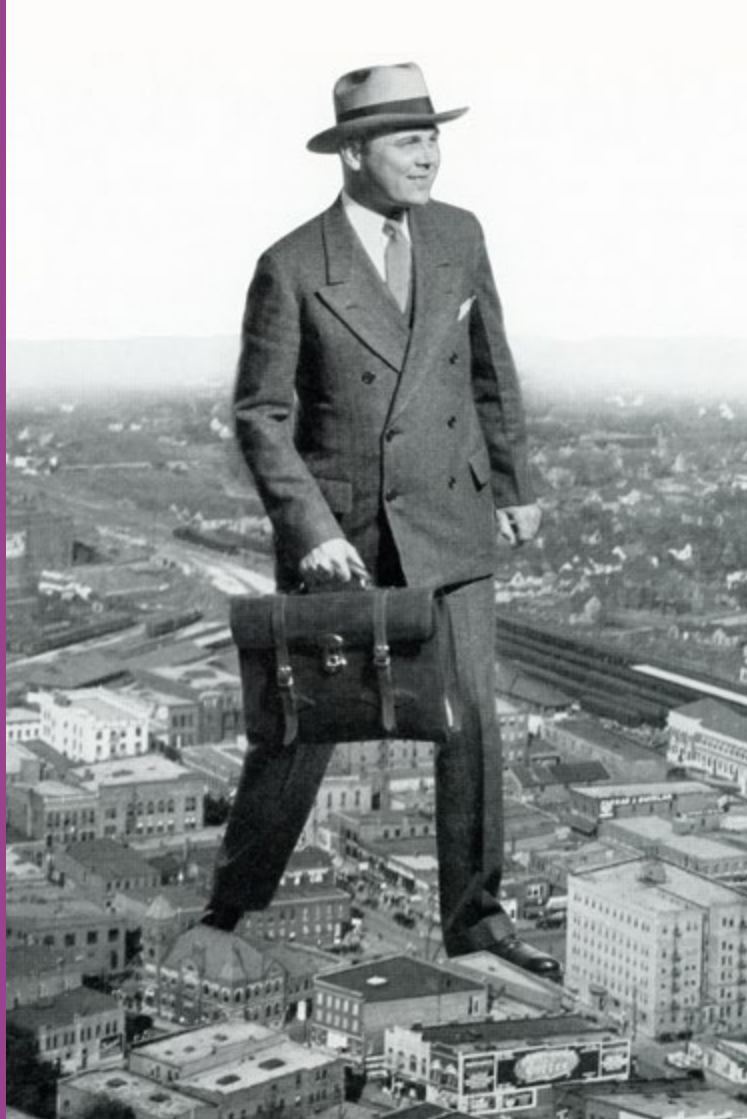
TRAVELLING SALESPERSON PROBLEM

A salesperson has to visit a number (*N*) of cities, each a certain distance from the others. The problem is to find a route that, from a given start point, visits all cities once only, and is the shortest round trip.



The problem for 5 cities

One obvious algorithm would be to generate every possible route and pick the shortest. In the 5 city problem, there are 24 possible routes. Any modern computer could find the shortest route in microseconds. However, the problem is that our algorithm increases very rapidly (like $O(N!)$). If there are 10 cities, there are 362,880 possible routes; 25 cities gives roughly 1.22×10^{17}; and for 75 cities, an unimaginably huge 3.31×10^{107} routes. (The number of atoms in the observable universe is somewhere between 4×10^{79} and 4×10^{81}.) Therefore no conceivable computer could examine this number of routes in the lifetime of the universe.



POLYNOMIAL ALGORITHMS

Realistically, only polynomial algorithms (the blue line in the graph overleaf), or better, are fast enough solving real-world problems.

Computer scientists refer to problems that can be solved by algorithms running in polynomial time (the time required to solve the problem) as belonging to **class P**. Another class of problem – called **NP** – comprises those in which an answer can be verified in polynomial time. For instance, it is very easy to verify whether a list of numbers is sorted, so sorting is an NP problem; and there are numerous polynomial algorithms for sorting that list, so the problem is P also.

Many problems are known to be able to be verified (NP), but with no known polynomial algorithm to solve them.

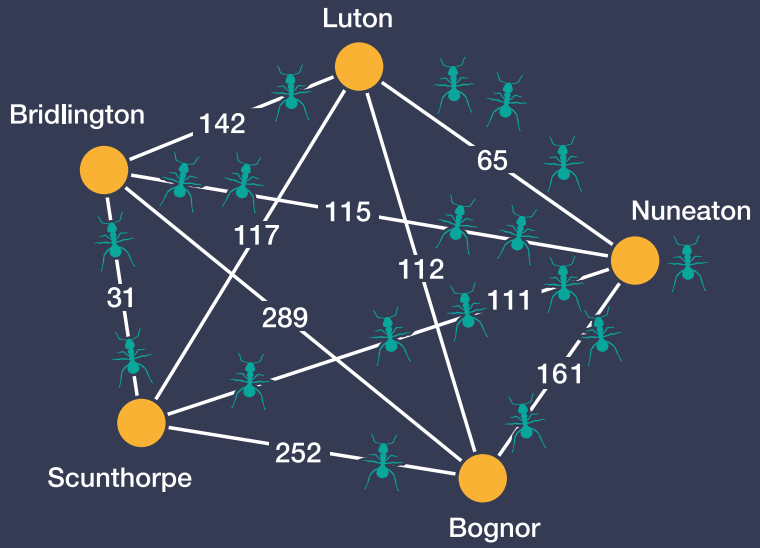
A major unsolved problem in computing and mathematics is whether P = NP. In other words, for every NP problem, is there a polynomial algorithm that can solve it? This is one of seven **Millennium Prize Problems**. Solve it and you will win US$1,000,000!

UNCONVENTIONAL ALGORITHMS

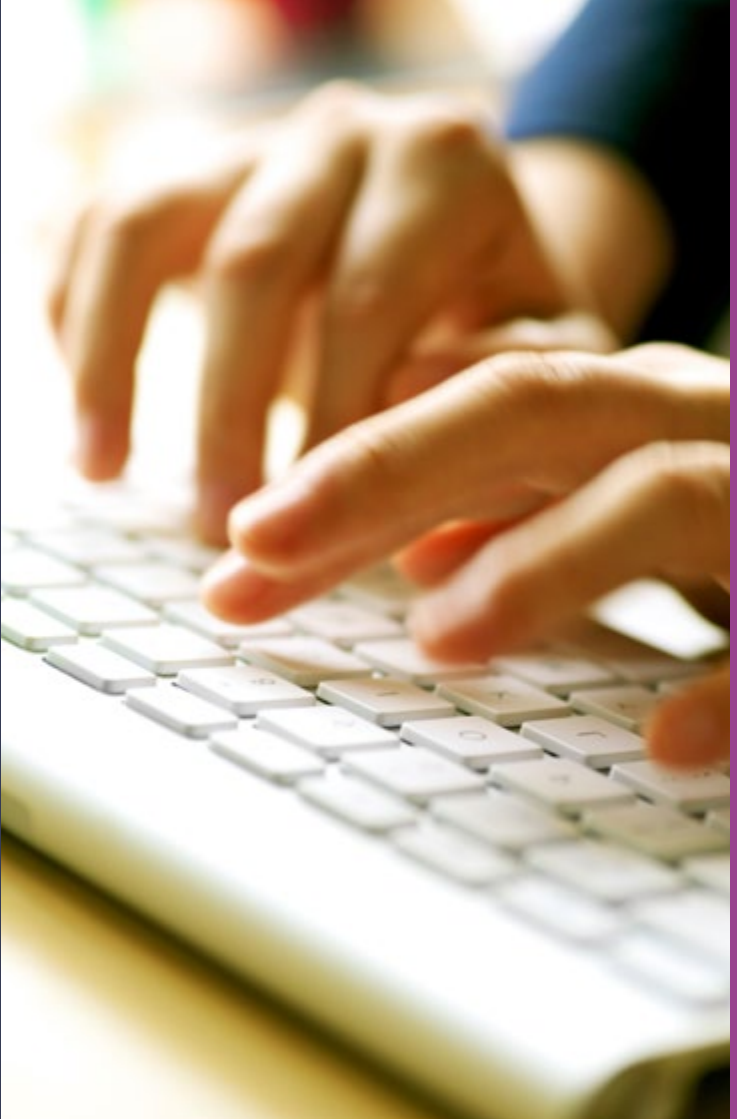
Unconventional algorithms, from the natural world, are often used to find a good answer.

The travelling salesperson problem may seem unrealistic, but it is representative of a huge class of important problems called **optimisation problems**. Computer scientists have devised a number of algorithms (**heuristic algorithms**) to tackle such problems. None of these can be certain to find the best possible answer, only good answers.

Travelling ants



A number of unconventional algorithms have also been developed, many of them based on the way in which problems are solved in the natural world. For example, it is known that swarms of ants are able to find the shortest distance between points. A similar principle can be applied to the travelling salesperson problem. A swarm of virtual 'ants' is distributed across a graph representing the cities. By applying simple rules, the swarm eventually congregates on the shortest path.



SUMMING UP

Despite some restrictions, there are still countless problems that can be solved algorithmically, and algorithms are creeping inexorably into daily life. But does this matter? Won't algorithms just make our lives easier and better?

We are quite accustomed to the idea that algorithms can deal with all the boring, repetitive jobs that humans generally don't like doing, supposedly liberating us for more creative and enriching work. But the evidence is now starting to accumulate that algorithms can also replace humans working in fields where we would consider that human intelligence, knowledge and skill are required – medicine, the law, planning, and so on.

A study conducted at the Martin School in Oxford examined 702 of such types of job, and concluded that 47% of current employment was at risk of replacement by technologies that are already operational, or are being tested in laboratories.

The future of work, and of society as we know it, may be threatened by algorithms …

LEARN MORE ABOUT algorithms and data structures with The Open University. ***M269 Algorithms, data structures and computability*** will help you become a computational thinker, exploring a range of computing concepts and applying these to a variety of problems.

CALCULATING DEVICES

BBC MAKE IT DIGITAL

The Open University

Innovations through history that have helped us to count

HELPING US COUNT

You may be surprised to know that the calculator you have in your pocket or on your smart phone has a rich history which stretches back over several millennia.

From clay tablets to marble counting boards, from abacuses to logarithm tables, from slide rules to calculating machines, the types of device have been many and varied. Each device has been an innovation of its day, yet all build on the same basic principle of mathematics.

LEARN MORE ABOUT mathematical concepts and techniques with The Open University. ***MST124 Essential mathematics 1*** provides a broad and enjoyable foundation for mathematics. It teaches you the essential ideas and techniques that underpin mathematical subjects.

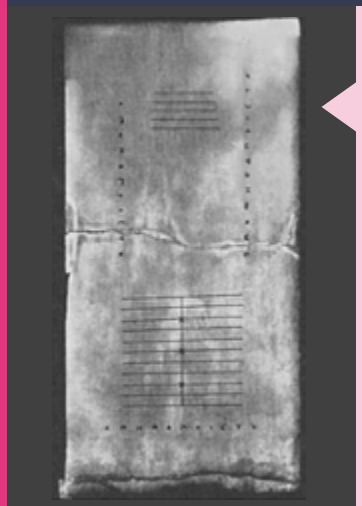
CALCULATING DEVICES

SUMERIAN CLAY TABLET

c.2600BC

The world's oldest datable mathematical table comes from the Sumerian city of Shuruppag to the north of Uruk. It is ruled into three columns on each side: the first two columns list length measures followed by the Sumerian word for 'equal', and the final column gives the products of these lengths in area measure.

c.2600BC — **c.300BC**



SALAMIS TABLET

c.300BC

The Salamis tablet, which is the earliest known surviving example of a counting board, was found on the Greek island of Salamis in 1846. It is made of white marble, has three sets of Greek numbers arranged around its edges, and measures approximately 150cm x 75cm x 4.5cm.

OPUS PALATINUM DE TRIANGULIS
1596

Georg Joachim de Porris, also known as Rheticus (1514–1574), is famous for facilitating the publication of Copernicus's *On the Revolutions of the Heavenly Spheres*. Rheticus's own masterwork, the 1500 page *Palatine Work on Triangles*, published posthumously, provides tables for all six trigonometric functions that were still being used in the 20th century.

NAPIER'S RODS
1617

Napier's Rods – often known as Napier's Bones, supposedly because the more expensive sets were made of bone or ivory – were created by John Napier to aid multiplication (only addition is needed to do the calculation). They consist of 10 rectangular blocks, and Napier explained how to use them in his book *Rabdologia*.

1596 **1614** **1617** **1620**

NAPIER'S LOGARITHMS
1614

The Scottish laird John Napier (1550–1617), in an effort to aid astronomers, spent twenty years developing his tables of logarithms, computing almost 10 million entries. His initial formulation was rather awkward but thanks to the mathematician Henry Briggs (1561–1630), who visited Napier in 1615, the logarithms were reformulated into the more practical form we know today.

WILLIAM OUGHTRED SLIDE RULE
1620

In 1620 Edmund Gunter (1581–1626) made a straight logarithmic scale (or rule) on which calculations could be performed using a set of dividers. William Oughtred (1574–1660) capitalised on this idea and used two such scales sliding by one another to perform direct multiplication and division, thus inventing the slide rule. He also developed a circular slide rule.

BLAISE PASCAL'S CALCULATOR
1648

Pascal appears to have been stimulated to develop his calculator while helping his father, a supervisor of taxes. It was a machine for addition and subtraction, and it went through 50 prototypes before Pascal presented it to the public in 1648. It is one of the world's first mechanical calculators.

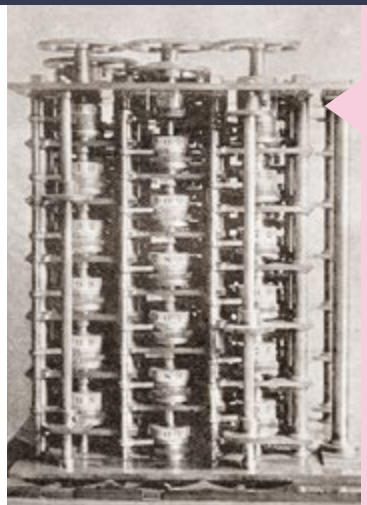
JACQUARD'S LOOM
1801

The Jacquard loom was a mechanical loom invented by the Frenchman Joseph Marie Jacquard. The loom was controlled by a chain of punched cards laced together in a continuous sequence. One card corresponded to one row of the design. Jacquard's punch-card system was later adapted by Charles Babbage.

1648 1671 1801 1822

STEPPED RECKONER
1671 – 1673

Gottfried Wilhelm Leibniz's calculator was inspired by Pascal's but had for its operating mechanism a new mechanical feature, a stepped drum. Although it was the first machine to perform all four arithmetic operations, its intricate precision gear-work, which was rather beyond the construction methods of the time, meant it rarely worked reliably.



CHARLES BABBAGE'S DIFFERENCE ENGINE
1822

In 1822 Charles Babbage proposed a mechanical calculator for computing tables of various mathematical functions using the method of divided differences. Between 1823 and 1842 the British government gave him £17,000 to build it but he did not succeed in constructing a working engine.

ARITHMOMETER
1851

The arithmometer was patented in in France in 1820 by Charles Xavier Thomas de Colmar although not manufactured until 1851. It could add and subtract directly, and perform long multiplication and long division by using an accumulator. It was the first commercially successful mechanical calculator, staying in production until 1915.

CURTA CALCULATOR
1948

The Curta was introduced by Curt Herzstark in 1948. It a small, hand-cranked digital mechanical calculator which can perform the four arithmetic operations as well as square roots and other operations. Its design derives from both Leibniz's stepped reckoner and Thomas de Colmar's arithmometer.

1851 1893 1948 1971

MILLIONAIRE CALCULATOR
1893

The 'Millionaire' mechanical calculator was designed by Otto Steiger, a Swiss engineer, and was in production until 1935. It was the first commercially successful calculator that could perform multiplication directly, and it was very fast for its day. The standard model weighed 72lb and some models could weigh up to 120lb!

POCKET CALCULATOR
1971

In 1971 the Japanese company Busicom released the LE-120A 'Handy', the first hand-held four-operation calculator on a chip. This was followed by the first hand-held scientific calculator, the Hewlett-Packard HP-35. Integrated circuits further reduced the size and cost, and by 1978 the power consumption was so low that pocket calculators could be driven by solar cells.

THE ABACUS

The word ‘abacus’ is derived from the Greek word ‘abax’ or ‘abakon’ meaning ‘table’ or ‘tablet’. It is a calculating device used for addition, subtraction, multiplication and division, with the advantage that it does not require pen and paper. There are two basic forms: a counting board with counters and a frame with beads strung on wires.

The counting board is a piece of stone, wood or metal, with carved or painted lines between which the counters are moved. Small stones, called calculi, were used with counting boards in Greece and Rome, while stamped metal discs were used with counting boards in Europe. The earliest known counting board is the Salamis tablet, dating from c.300BC.

There are three main forms of the bead frame abacus in use today: the Chinese, the Japanese and the Russian.

The Chinese abacus, the suanpan, for which there is documentary evidence as early as AD190, is typically about 20cm tall and usually has 13 wires. There are two beads on each wire above the divider and five beads on each wire below the divider. The Japanese abacus, the soroban, derives from the Chinese abacus which was imported to Japan in the 14th century. Its modern form has one bead above the divider and four beads below. The Russian abacus, the schoty, usually has ten beads on each wire with no divider. It is often used vertically, with wires from left to right in the manner of a book.

LOGARITHM TABLES

The idea of a logarithm first emerged independently and almost simultaneously in the work of two men, the Scottish laird John Napier (1550–1617) and the Swiss craftsman Joost Bürgi (1552–1632), and within years of one another they had both produced tables for its use, Napier in 1614 and Bürgi in 1620. Napier and Bürgi were working in an era when the computation of mathematical tables, particularly those involving trigonometric functions (sin, cos, tan, etc.), was very important for navigation.

It was clearly vital that such tables were as accurate as possible. Thus there was a strong motivation to find a way of simplifying the processes of multiplication and division to the level of addition and subtraction, which is exactly what logarithms do.

Napier's tables contained the logarithms of sines and tangents but were difficult to use. Fortunately, Napier's invention was rapidly and enthusiastically taken up by Henry Briggs (1561–1630), the professor of geometry at Gresham College in London, who set about rectifying what he saw as the defects in Napier's construction. In 1615 Briggs visited Napier in Edinburgh and together they agreed on a much more convenient form, now called logarithms to base 10.

Briggs's tables formed the basis of all logarithm tables published for the next 200 years or so.

DIFFERENCE ENGINE

By the beginning of the 19th century, the difficulty of producing error-free mathematical tables was a long-standing problem. Errors were made by human 'computers' doing the calculations and by the typesetters printing the tables. In 1822 Charles Babbage (1791–1871) proposed a mechanical calculator that would both calculate and print the tables, thereby eliminating both types of error. It would compute tables of various mathematical functions using the method of divided differences, a way to tabulate functions using polynomial coefficients. Babbage's machine had the potential to produce many useful (and error-free) tables.

In 1823 the British government awarded Babbage £1700 to kick-start the project. However, although there was nothing wrong with Babbage's design, to build the machine required technical expertise beyond anything that had been called on before, and it turned out to be much more expensive than Babbage, or the British government, had anticipated. It required an estimated 25,000 parts! In 1833 Babbage fell out with his chief engineer and production stopped. By 1842, with £17,000 spent and with no completed difference engine in sight, the British government ceased the funding, and the project ground to a complete halt. It was revived again only at the end of the 20th century. In 1991, a working difference engine, constructed to Babbage's later designs of 1847–9, was completed by the Science Museum in London, in in time for the bicentennial of Babbage's birth. It consists of 4000 parts, weighs three tons, and measures 11 feet long. The printing mechanism was completed in 2000.

SIMPLE CODING

BBC MAKE IT DIGITAL

The Open University

How to code the building blocks of algorithms in Python

INTRODUCTION

Software makes the world go round. Cars and TVs have software that controls how they work, and global commerce and finance are impossible without software to control the stocks, carry out payments, find the best transport route, etc.

Coding (or programming) is the construction of software. Coding involves writing a 'recipe', which in computing is called an **algorithm** (see the algorithms leaflets), in a so-called **programming language** that a computer can understand. When the computer **runs** the code, it follows the 'recipe', step by step.

I will use Python, a popular programming language for teaching and for professional software development. You will see that Python code reads almost like plain English. Writing simple programs in Python is not very difficult, once you have come up with the 'recipe', i.e. the algorithm.

All code shown in this leaflet can be run online, without installing any software on your computer. Just go to **www.open.edu/openlearn/makeitdigital.** There you will also be able to change the code and share your creations with family and friends via email or on social media.

In this leaflet we will look at how to code in Python the building blocks of all algorithms (sequence, condition, repetition), and how to ask the user for input and produce some output on the screen. Let's start!

SEQUENCE

Let's imagine we are developing software for a restaurant, where a tip of 10% is added to the bill. The code is on the right. It's simply a sequence of **instructions**, written one per line, and executed one by one from top to bottom. Our first program has only two kinds of instructions.

The first instruction is an **assignment**: the computer evaluates the **expression** on the right of the assignment (=) and stores the result in the **variable** on the left of the assignment. Each piece of information we need has to be stored in a variable.

For example, the third assignment states 'let the tip be the expenses multiplied by the percentage'. Note that in Python the asterisk is the multiplication operator.

The last instruction, **`print`**, prints some text on the screen, followed by the computed result. Note the following:

- The comma separates the two things to be printed, in the same way we use commas in English to enumerate two, three, or more things.
- Text is written between double-quotes, which are not printed themselves. In Python, a sequence of characters surrounded by double-quotes is called a **string**.

I'm pointing out these details because they're important once you start writing your own code. Computers are not as smart and accommodating as human readers: at the slightest spelling mistake (like **`pint`** instead of **`print`**) or missing punctuation (like forgetting the comma or double-quotes), the computer will give up on understanding and running your code.

Writing the program

```
1  expenses = 54
2  percentage = 0.10
3  tip = expenses * percentage
4  bill = expenses + tip
5  print "Total bill:", bill
```

Running the program

```
Total bill: 59.4
```

CONDITIONS

Let's further assume the tip is automatically added only for groups of more than 6 people. We need one more variable, to store the number of people in the group, and one new instruction to handle both cases: if there are more than 6 people, the percentage is 10%, otherwise it's 0%. The code is on the right.

The instruction '`if` condition: block `else`: block' works as follows. The computer checks the condition after the `if`. If it is true, the computer then executes the **block** of code (the indented instructions) belonging to the `if` part. If the condition is false, the computer executes instead the `else` block. The indentation is needed to know which instructions belong to which part. Afterwards the computer continues executing the non-indented instructions.

You should of course not just believe the code is correct, but **test** it yourself. Large software companies employ many testers to check their code. Good testing includes choosing enough inputs (preferably borderline cases) to exercise all possible conditions. In this case, we should at least test for 6 and 7 people. If you go online, you can change the number of people to 6 and check that the total bill is just the expenses (54).

Notice there is a colon (:) at the end of the `if` and `else` lines. Forgetting the colons and forgetting to indent the instructions will lead to error messages.

Writing the program

```
1  expenses = 54
2  people = 7
3  if people > 6:
4    percentage = 0.10
5  else:
6    percentage = 0
7  tip = expenses * percentage
8  bill = expenses + tip
9  print "Total bill:", bill
```

Running the program

```
Total bill: 59.4
```

REPETITION

A restaurant bill is made up from the prices of the various food and drink items ordered. Our program should sum those prices to obtain the expenses, to which the tip will then be added. The algorithm (which is independent of any programming language) is as follows.

1. Let **`items`** be a list of the prices of the items ordered.
2. Let the expenses be zero.
3. For each item in the items list:
   add it to the expenses.
4. Print the expenses.

This can be directly translated to Python, as shown on the right. In Python, **lists** are simply comma-separated values, within square brackets. (Note that I chose values that add up exactly to the same expenses as before.)

The '**`for`** variable **`in`** list: block' instruction goes through the given list and successively stores each value of the list in the variable, then executes the block, which will refer to the variable to access its value.

In this case the value is added to the current expenses and the result stored again in **`expenses`**, so that it is always up to date. Therefore the new expenses are the old expenses plus the item ordered.

Writing the program

```
1  items = [4.35, 2, 19.95, 22.70, 5]
2  expenses = 0
3  for item in items:
4    expenses = expenses + item
5  print "Food and drinks:", expenses
```

Running the program

```
Food and drinks: 54.0
```

FUNCTIONS

The previous algorithm is calculating the sum of a list of numbers. This is such a common need that Python already provides a **function** for that, appropriately called `sum`. A function takes some data as input and returns some other data as output, e.g. `sum` takes a list of numbers and returns a single number.

To apply a function to some data (whether a variable, a number or a string), just write the name of the function followed by the data in parentheses. The computer will calculate the function's output, which can be used in further calculations or assigned to a variable.

As you can see on the right, using the `sum` function shortens our code and makes it easier to understand, because the function's name explicitly states what the code is doing. Good programmers don't just write code, they write readable code. They know that code always changes to accommodate further customer requests, and trying to modify cryptic code you wrote some weeks or months ago is no fun.

Another useful function is `input`: it takes a string that it shows on the screen to the user, and returns a string with what the user typed on the keyboard until they pressed the RETURN or ENTER key. If you go online, you can run the code on the right and type in your own name.

NOTE: the `input` function always returns a string, even if the user types in a number. This is important, as we'll see next.

Writing the program

```
1 items = [4.35, 2, 19.95, 22.70, 5]
2 expenses = sum(items)
3 print "Expenses:", expenses
4 name = input("What's your name?")
5 print "Nice to meet you,", name
```

Running the program

```
Expenses: 54.0
What's your name? Mary
Nice to meet you, Mary
```

REPETITION, AGAIN

Let's use **`input`** to ask for the prices of orders instead of fixing them in a list. This requires a new iteration: an infinite loop that keeps asking until the user types 'stop', for example. We also need to use the **`float`** function to convert the string returned by **`input`** into a decimal number (also called a **floating-point** number) we can add to the expenses. The algorithm is:

1. Let the expenses be zero.
2. Forever repeat the following:
    1. Let **`answer`** be the reply to the question "Price of order?".
    2. If the answer is equal to "stop":
        Exit the loop.
    3. Otherwise:
        1. Let the price be the answer converted to a decimal number.
        2. Add the price to the expenses.
3. Print the expenses.

The translation to Python is on the right.

NOTE: we write two equals signs to check for equality, because a single equals sign is the assignment instruction.

Now go online to **www.open.edu/openlearn/makeitdigital** and put the complete program together. It starts as on the right, then asks the user for the number of people, calculates the tip, and prints the total bill.

Writing the program

```
1  expenses = 0
2  while True:
3    answer = input("Price of order?")
4    if answer == "stop":
5      break
6    else:
7      price = float(answer)
8      expenses = expenses + price
9  print "Expenses:", expenses
```

Running the program

```
Price of order? 4.35
Price of order? 2
Price of order? 19.95
Price of order? 22.70
Price of order? 5
Price of order? stop
Expenses: 54.0
```

SUMMING UP

In this brief introduction to programming you have seen the fundamental building blocks provided by most programming languages:

- assignments (=) to store data in variables
- simple data types (strings and numbers)
- data structures (lists)
- sequence (one instruction per line)
- iteration (`for` and `while` loops)
- conditional instruction (`if-else`),
- comparisons (>, <, ==, >=, <=)
- functions (`sum` for lists, `float` to convert strings to numbers)
- input from the keyboard (`input` function)
- output to the screen (`print` instruction).

Programming languages have to be automatically understood by a machine, so the syntax and grammar are much more constrained than in English. Any spelling mistake like writing `flat` instead of `float`, or forgetting punctuation like commas and colons, or using the wrong data type like adding a string to a number, leads to an error.

You have also seen that programming involves writing clear and understandable code (note the plain English names of our variables and functions) to make it easier to change later, and testing it thoroughly.

Learning to program forces us to think clearly and rigorously when solving a problem, because the solution has to be described in very small and precise steps that even a machine can understand. Python makes it easy to write the code once we come up with a sufficiently detailed algorithm, but the thinking (still) has to be done by us.