A History of the World in Four Algorithms
Professor Sarah Hart
15 Oct 2026

Introduction

Algorithms are fundamental to the story of mathematics. A theorem may prove that a solution to a problem exists, but an algorithm gives you the directions to actually find it. In this lecture, we'll explore four brilliantly ingenious algorithms that have not just shaped pure mathematics, but have found unexpected applications in areas as diverse as nuclear physics, film-making, medicine, and music. The four algorithms I have chosen highlight different times and places of mathematical development in history, and follow the impact through to modern applications both within and beyond mathematics.

Algorithm 1: The Babylonian "equation-solver"

A mathematical algorithm is a step-by-step procedure for solving a given problem. Opinions differ on the best definition, but there's broad consensus that an algorithm must have a finite number of executable and unambiguous instructions and must terminate with an output – the solution to the problem.

The earliest surviving examples come from Babylonian clay tablets. They are algorithms for solving specific problems about finding lengths and widths of rectangles – effectively solving what we now call quadratic equations. One example is from the catchily named AO8862 (the catalogue number in the Louvre's Département des Antiquités orientales). It was probably created in the ancient city of Larsa (modern-day Tell as-Senkereh in southern Iraq).

The Babylonians used sexagesimal notation (that is, 60 rather than 10 was the base. So the number 260 would have been written 4,20 because it is $4 \times 60 + 20$). But translating into English and base 10, tablet AO 8862 includes the following problem:

I have multiplied length and width to find the area. Then to the area I added the excess of the length over the width, and got 183. Moreover, I have added length and width: 27. Required: length, width and area.

Then there is a procedure for finding the solution. It may not look like much, but here are the beginnings of mathematics (and of algorithms), four thousand years ago. Initially rooted in practical problems like division of land areas, these kinds of questions soon took on a life of their own as more abstract geometric problems. The word "algorithm" comes from the 9th century CE Persian mathematician Al-Khwarizmi, some of whose algorithms are processes for solving quadratic equations that are much more sophisticated and general than the foundational Babylonian versions.

The search for corresponding algorithms to solve higher order equations, which started in earnest in Renaissance Italy, would lead to some of the most exciting mathematical developments of the 19th and 20th centuries, as mathematicians battled to find formulae for cubics, quartics, and beyond, culminating in the proof by Évariste Galois that the general quintic equation with rational number coefficients has no solution by radicals (that is, a formula involving only rational numbers and their square, cube and higher roots).

Algorithm 2: The Euclidean Algorithm

My next example is the Euclidean Algorithm, an ingenious method of finding greatest common divisors (gcds) that is also at the heart of fundamental theorems in mathematics. It is thousands of years old, but remains vital in mathematics, and has found modern applications that Euclid could never have dreamed of. The basic Euclidean Algorithm finds the greatest common divisors of pairs of numbers. (There is disagreement among historians of mathematics about whether it really is due to Euclid or earlier mathematicians, but it certainly appears in Euclid's Elements.) It works by repeated division. Take positive integers a and b, and for the sake of argument assume $a \leq b$. Then we can divide b by a to get a quotient q and remainder r, with $0 \leq r < a$. That is, $b = qa + r$. If the remainder is zero, then $b = qa + 0 = qa$. In that case, since a cannot have any larger divisors than itself, $\gcd(a, b) = a$, and we can stop right there. Let's suppose, then, that $r > 0$. Now comes the key insight: $\gcd(a, b) = \gcd(r, a)$. Here's why. Anything that divides both a and r must also divide $qa + r$, which equals b. By the same token, anything that divides both a and b must also divide $b - qa$, which equals r. Therefore, the set of common divisors of a and b is *the same* as the set of common divisors of a and r. In particular, $\gcd(a, b) = \gcd(r, a)$. So, we repeat the division/remainder step with r and a, and keep going. For a toy example, when $a = 5$ and $b = 13$ the steps would be these.

$$\begin{aligned} 13 &= 2 \times 5 + 3 & \gcd(5,13) &= \gcd(3,5) \\ 5 &= 1 \times 3 + 2 & \gcd(3,5) &= \gcd(2,3) \\ 3 &= 1 \times 2 + 1 & \gcd(2,3) &= \gcd(1,2) \\ 2 &= 2 \times 1 + 0 & \gcd(1,2) &= 1 \end{aligned}$$

Because $r < a$, the remainder in the calculation keeps decreasing at each stage until inevitably we reach a remainder of zero in a finite number of steps. When that happens, for whatever "a" and "b" are in play at that stage, it means that $\gcd(a, b) = a$. In other words, the gcd of our original pair of number is the last nonzero remainder in the Euclidean algorithm. This calculation may look less efficient than the alternative method of calculating greatest common divisors, which is to factorise both numbers and find the highest factor appearing in both lists. But factorisation is a terrible method for large numbers, because it is notoriously hard. It's much quicker to do the Euclidean algorithm to find the greatest common divisor of a pair like 1009 and 1013 than it would be to do a brute force factorisation (especially as they are both prime).

Incidentally, the Euclidean algorithm is the first algorithm we know to have received any kind of formal complexity, or runtime, analysis. In 1844, a French mathematician called Gabriel Lamé (one of several who worked on this question in the first half of the 19th century) showed that the number of steps in the algorithm is at most five times the number of digits in the smaller of the two initial input numbers. For instance, the calculation of $\gcd(89, 144)$ will take, since 89 is a 2-digit number, at most 10 steps, and the calculation of $\gcd(5702887, 9227465)$ will take at most 35 steps. Factorising 5,702,887 requires checking every potential prime factor up to $\sqrt{5702887} \approx 2388$, which is a lot more work (it factorises as 1597×3571, while 9227465 is $5 \times 13 \times 141961$). Interestingly, the "worst case" scenario, meaning, for a given n, the smallest value of $a + b$ such that the calculation of $\gcd(a, b)$ requires n steps, turns out to be when a and b are the $(n + 1)^{st}$ and $(n + 2)^{nd}$ Fibonacci numbers (the sequence 1, 1, 2, 3, 5 and so on). Since the eleventh and twelfth Fibonacci numbers are 89 and 144, that is the smallest pair of numbers requiring ten steps to calculate the gcd, almost as if I chose that example earlier on purpose.

I won't have time to discuss this in the lecture, but there's another side to the Euclidean algorithm, and that's what happens when you reverse it. Taking the steps above for $\gcd(5,13)$ we can rearrange each equation so that the remainder is the subject of the formula. Working up from the last nonzero remainder we then get $\mathbf{1} = \mathbf{3} - 1 \times \mathbf{2} = \mathbf{3} - (\mathbf{5} - 1 \times \mathbf{3}) = 2 \times \mathbf{3} - \mathbf{5} = 2 \times (\mathbf{13} - 2 \times \mathbf{5}) - \mathbf{5} = 2 \times \mathbf{13} - 5 \times \mathbf{5}$. We have expressed the greatest common divisor of our starting numbers as a combination of those two numbers (in our case, $2 \times 13 - 5 \times 5 = \gcd(5,13) = 1$. It can be proved that this "extended Euclidean algorithm" always works. That is, given any integers a and b, there are integers u and v with the property that $au + bv = \gcd(a, b)$. The extended Euclidean algorithm has many useful applications in number theory and cryptography.

The Euclidean algorithm's "divide and conquer" process has been adapted for related purposes in nuclear physics and music. In the case of nuclear physics, it's sometimes important to find the most equal way to distribute a number of energy pulses within a fixed time period (say 5 pulses in 13 milliseconds). For obvious reasons, it's inadvisable to bombard your reactor with all the energy all at once. It turns out that the

Euclidean algorithm gives rise to an optimal solution. In music, the corresponding question is what rhythms arise from stressing a certain number of beats within an overall total (say 3 stressed beats from a total of 8). “Euclidean rhythms” have the property that the stressed beats are as evenly spaced as possible, and have emerged organically in global traditions. The Euclidean “3 from 8” prediction, for instance, gives what is commonly known as the Tresillo rhythm, one of the most popular patterns in world music.

Algorithm 3: The Fast Fourier Transform

The next algorithm we’ll discuss has its roots in the work of 19th century mathematicians. Joseph Fourier showed that periodic functions can be broken down as sums of sine waves. (Not quite every periodic function has this property, but the kinds of periodic functions used in applied maths are usually OK, as they tend not to have infinitely many discontinuities or other pathological behaviour of the kind pure mathematicians like to dream up.) Moreover, given one of these periodic functions, there are formulae that allow us to calculate its “Fourier series” – that expression as a sum of sine waves – to as much accuracy as we like. There are more general results that allow us to analyse a wider class of functions using “Fourier transforms”. That’s all very well, but we aren’t always “given” the function in the sense of having a known $f(x)$ on which we can carry out this kind of Fourier analysis. In real life, what we usually have is an input or signal that we are trying to understand based on sampling a finite number of points. What would be wonderful is to have an algorithm for coping with the “discrete Fourier transform” problem – finding a sort of best-fit collection of sine waves to approximate the inferred signal from a finite number of measured points. Algorithms that do this are known as fast Fourier transforms, or FFTs, and the best-known and most widely used of them is the version put forward by mathematicians James Cooley and John Tukey in 1965 (though it later emerged that Carl Friedrich Gauss had, unbeknownst to them, discovered the underlying technique in 1805). The reason such a process is useful is because it converts information in the “time domain”, where we look at how the signal varies over time, to information in the “frequency domain”, where we look at the strength of the signal at each frequency. Knowing the constituent frequencies of the signal allows us to store it digitally and reconstruct it. The FFT really is extraordinarily impressive in terms of efficiency and runtime. It is indispensable in almost any form of signal processing. To give a rough idea of how these processes work, let’s first consider the original problem of a “nice” periodic function that we want to break into its constituent sine waves. If you have a periodic function of period P, then the only sine waves that have a chance of forming part of the sum are ones that fit a whole number of times into that period. So each one has period $\frac{P}{n}$ for some n. Traditionally, because sine and cosine are functions related to travelling around a circle, and the circumference of the unit circle is 2π, we tend to do our analysis of periodic functions by scaling if necessary to give them a period of 2π, the same as $\sin(t)$. Our essential ingredients, then, will be $\sin(t), \sin(2t)$, $\sin(3t)$, and so on. But the basic wave $\sin(nt)$ can be shifted to the left or right on the axis by changing its phase. In $\sin(nt-\phi)$, the phase shift is ϕ; that is, the curve is shifted to the left by ϕ, without affecting the frequency. How on earth are we going to manage all these infinitely many potentially waves all of the same frequency? Fortunately, we only need to worry about two of them, because of a lovely formula you may remember from school: $\sin(a-b) = \sin a \cos b + \cos a \sin b$. Here we get

$$\sin(nt-\phi) = \cos\phi \sin(nt) + \sin\phi \cos(nt).$$

The contributions of all of these phase-shifted $\sin(nt)$ curves combine to make a single expression $A\sin(nt) + B\cos(nt)$. The upshot is that our function is going to be an infinite sum

$$f(t) = \sum_{n=0}^{\infty} A_n \sin(nt) + B_n \cos(nt)$$

And now “all” we have to do is find those coefficients. They tell us how much of each sine wave goes into the recipe for our function. Fortunately, there’s a neat way to do this. The intuition is as follows. If we multiply two different sine functions together and look at the graph of this product $\sin(mt)\sin(nt)$, because they are not the same frequency we find that the times when both are positive or both are negative seem to cancel out the times when one is positive and the other is negative. That means the area under the graph (counting above the axis as positive and below the axis as negative) is actually zero. The only time this doesn’t happen is when $m = n$. Then they reinforce each other. We can use calculus to prove this, and to find the exact appropriate coefficient when $m = n$. This basic idea is how we isolate the coefficients: multiply $f(t)$ by $\sin(nt)$, find the area under the graph and hence the appropriate coefficient, and do the same for $\cos(nt)$.

The discrete Fourier transform is built on the same principles. For n data points you have n candidate sine and cosine waves, so there are n^2 calculations to do. (Why not $2n^2$? If you’ve used complex numbers you’ll

know that there's a way to do the sine and cosine calculation at the same time. Euler's formula tells us that $e^{i\theta} = \cos(\theta) + i\sin(\theta)$, so by multiplying our $f(t)$ by the appropriate $e^{i\theta}$ we get a complex coefficient whose real and imaginary parts give the separate cosine and sine coefficients.)

The problem with the discrete Fourier transform is runtime. With 10 points, you have on the order of 100 calculations. But with 1000 points there are a million calculations, and if you had 44,100 points (that's the number of samples per second used to record CDs, we're looking at almost 2 billion calculations per second. The FFT tackles this issue by exploiting the symmetries of the sine curve. For example, suppose we are dividing a period of 2π into eight samples, at times t_0, t_1 up to t_7, at intervals of $\frac{\pi}{4}$. Then we want to know (among other things) $\sin(nt_1) \times f(t_1)$ for $0 \leq n < 8$. But the sine function has the property that $\sin(\pi + x) = -\sin(x)$. For instance, $\sin 5t_1 = \sin\frac{5\pi}{4} = -\sin\frac{\pi}{4} = -\sin t_1$, $\sin 6t_1 = \sin\frac{6\pi}{4} = -\sin\frac{2\pi}{4} = \sin 2t_1$ and so on. Similarly, $\sin(2\pi + x) = \sin x$, so that, for example, $\sin 5t_2 = \sin\frac{5\pi}{2} = \sin\frac{\pi}{2} = \sin t_2$. So we can reduce the number of different calculations significantly. In fact, we can essentially halve the work by pairing calculations in this way. Even better, this is a recursive process because in effect it splits the work into two smaller discrete Fourier transforms each with half the number of points, on which we can play the same game. The FFT reduces the runtime of the basic discrete Fourier transform from n^2 to $n\log_2 n$. In other words, if $n = 2^k$ then instead of n^2 calculations you only need nk of them. For a sample of $16 = 2^4$ points, instead of 256 calculations you only need 64. For 1000 points it's 10,000 rather than the 1 million we would have needed before. And those 44,100 points require not 2 billion calculations, but only 680,000. That's verging on 3000 times as fast. The FFT has countless scientific and engineering uses. It's a critical ingredient in sound recording, sampling, and compression algorithms like mp3. It's also integral to signal processing for mobile phones, enabling high speed data transmission needed for streaming by splitting the signal into multiple carrier frequencies. It is arguably the most important algorithm of the last century.

Interlude: the age of the algorithm

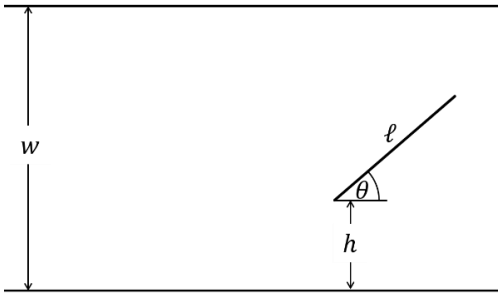
We've seen that algorithms have existed for thousands of years. Every child knows several: the column method for addition; long multiplication; the quadratic formula. But the computer has transformed our relationship with algorithms and made them fundamental to all our lives. Before our final choice, I wanted briefly to highlight this impact. The FFT would not exist without the computer, nor would many other algorithms that might have made the list. Examples include PageRank, the algorithm created by Google that revolutionised search engines, and compression algorithms behind efficient storage of images, audio and video in the form of JPEG, MPEG, MP3 and others. The very first computer program, before either of those words meant what they do now, was written by Ada Lovelace for the Analytical Engine (which Charles Babbage was never able to get funding to build), and was an algorithm to calculate Bernoulli numbers. Computer programs themselves are often algorithms (not always, because they do not always terminate, and that was one of the conditions of our definition). There is an important intermediary class of algorithms associated with programming: compilers. A compiler turns one programming language into another, usually a high-level language that makes sense to humans, like C++, into a lower level language like assembly language or machine code. The first compilers were built in the early 1950s, and depending on exactly how you define "compiler", the credit for the absolutely first one would go either to Grace Hopper or Alick Glennie, in 1952. At the turn of the millennium, the IEEE journal "Computing in Science and Engineering", devoted a special issue to the ten algorithms with "the greatest influence on the development and practice of science and engineering in the 20th century". One of those ten algorithms was a compiler: the Fortran I compiler developed by John Backus and his team at IBM in the 1950s. It translates the high-level Fortran (FORmula TRANslation) code into efficient machine code, including optimization algorithms that reduced redundancy and dramatically improved runtimes. The moral is that even the algorithms are made from algorithms.

Algorithm 4: The Monte Carlo method

My final selection is the Monte Carlo method – it's an algorithmic technique that has spread far beyond its initial beginnings, but the essential idea emerged in the 1940s in the work of mathematicians Stanislaw Ulam and John von Neumann, who were next door neighbours for many years at Los Alamos. Ulam was recuperating after an illness, playing games of patience to pass the time, when he started to wonder about the probability of dealing a winning hand. The number of possible shuffles of a deck of 52 cards is astronomical: 52 possible first cards, multiplied by 51 for the next, 50 for the next, and so on, to get $52 \times 51 \times 50 \times \cdots 3 \times 2 \times 1 \approx 8 \times 10^{67}$. How many of those lead to a win? A case-by-case analysis is impossible. But, thought Ulam, we could get a rough answer by sampling. Play a hundred games, see how many we win, and thus get an estimate of the probability of success. That's the idea behind the Monte Carlo

method. We don't know precisely which form of patience Ulam was playing – he called it *Canfield* (similar to an English game Demon). If it's the version we play today, then the latest estimates (and the Monte Carlo method is still the way we obtain these estimates), say that he'd expect to win about 71 of the 100 games.

You might object to calling Monte Carlo an algorithm – shouldn't an algorithm with a given input give the same answer every time? Not necessarily. By our definition, an algorithm is a process with unambiguous instructions, that terminates in a finite number of steps, and gives an answer. And "toss a coin" is an unambiguous instruction, even though the outcome is not known in advance. By the way, Ulam wasn't the first person to experiment with probabilistic methods. One candidate for this honour would be the French mathematician and scientist Georges Louis LeClerc, le Comte de Buffon. If you believe everything you read about him, you'd conclude that it's probably just as well he died in 1788, the year before the French Revolution, given that one of his experiments allegedly (spoiler alert: this isn't true) involved throwing perfectly good baguettes over his shoulder onto the floor behind him to estimate the value of π, which has rather a "let them eat cake" vibe. This would never be done genuinely to find the value of π – by this time it was already known to great precision – but to establish that random processes could be used to answer questions with a defined answer. The experiment is known as Buffon's needle, because one formulation of it is based on dropping a needle onto a wooden floor and asking what is the probability that the needle crosses a horizontal edge of a floorboard.



Suppose the needle has length ℓ, is at an angle θ to the horizontal, where $0 \leq \theta < \pi$ radians, and that its lowest point is at height h above the closest edge below it. (I'm talking about above and below because the diagram is drawn that way, not because the floor is vertical!) If each floorboard has width w (with $\ell < w$), then $0 < h \leq w$ and the needle will cross the edge when $h + \ell \sin\theta \geq w$. That is, when $w - \ell \sin\theta < h \leq w$. In other words, it crosses the edge $\frac{\ell \sin\theta}{w}$ of the time. But that's for a specific angle. To find the expected value of $\frac{\ell}{w}\sin\theta$ what do we do? The expected value of $\sin\theta$ is the average value v of $\sin\theta$ over the appropriate range (from 0 to π). A rectangle of height v and width π would have the same area as the area under the graph. We can find this area using calculus – it turns out to be 2 – and from that deduce that $v = 2/\pi$. The probability of the needle crossing a floorboard edge is therefore $2\ell/(\pi w)$. If, for instance the needle's length is half the width of a floorboard, then $2\ell = w$, giving a probability of $1/\pi$. So if we throw the needle a hundred times and find that it lands across an edge 32 times, then we estimate $1/\pi = 32/100$, which gives the estimate $\pi \approx 3.125$. A note of caution here. Buffon did carry out statistical experiments to test mathematical theory, so it is fair to give him some credit in this area. (At least, he caused experiments to be carried out. In one case, to explore the probabilities in a coin-tossing game, he played the game 2,048 times, getting a nearby child to do the actual coin-tossing for him.) But there is no real evidence of him having used the "Buffon's needle" problem to find π statistically, and the bread-based experiment story is a mere baguette bagatelle, arising from a mistranslation. Buffon phrased his question like this: *Sur un plancher qui n'est formé que de planches égales & parallèles, on jette une baguette d'une certaine longueur, et qu'on suppose sans largeur. Quand tombera-t-elle sur une seule planche?* Translated, this says "you throw a stick of a given length and negligible thickness onto a floor consisting of equal and parallel floorboards. When will it fall onto a single floorboard?" Baguette just means stick.

Anyway, back to Monte Carlo. It took off from the 1940s on because with computers it was suddenly possible to run thousands of simulations in fractions of a second. For instance, we can run a very simple Monte Carlo algorithm to find an approximation to π by inscribing a unit circle (that is, a circle of radius 1) inside a square, randomly choosing N points inside the square for some large N, and counting how many of those lie in the circle. Since the area of the unit circle is π and the area of the square containing it is 4, we'd expect $\pi/4$ of the points to lie in the circle. So, if k of the N points lie in the circle, then our approximation of π, using this algorithm, would be $4k/N$. This is a bad way to calculate π, because we have much better deterministic ways to do it, but that's not true for the *Canfield* problem, and nor is it true for many real-world problems where solutions are either not known, or theoretically can be found but with methods that would take too much computing power or time. In those situations, the Monte Carlo method is often the best and only hope. It's used in particle physics, climate science, finance, and many other fields. I'll give just two examples.

The first is in film-making, in a technique called ray tracing that helps produce hyper-realistic computer-generated images that fit seamlessly into real scenes. (It's also used to produce entirely imagined worlds in computer animation.) The idea is this. Suppose you want to put a computer-generated object into a scene

in your movie. To do this convincingly, you need to know how the light sources in the scene would interact with the object if it were really there. It's inefficient to map every single ray of light coming from a light source, because the vast majority won't encounter the object, and of the ones that do, the majority won't then bounce off it in the exact direction of our eyes. So ray tracing works in reverse. From the virtual "camera", you send a ray through each pixel of the intended image towards the object. The simplest next step is this: when the ray hits the object, a "shadow ray" is directed towards the light source. If there is an obstacle blocking its path, then the object is in shadow at that point and the pixel turns dark. If not, then it is light. No randomness involved. But that's a very crude approximation of reality. Real objects don't tend to have perfectly smooth, perfectly reflective surfaces. Materials like wood or fabric with rough surfaces will scatter light in unpredictable directions; materials like glass or water may also refract it. And light from light sources is reflected and refracted off other surfaces in the scene, so it isn't coming just from one or more light sources. It's also not enough to say that each pixel is "light" or "dark": reflection from (say) a green object will make the light subtly greener – we must be able to gauge both the intensity and the colour of the light. Finally, if the object being rendered is reflective, we have to know what images to reflect in its surface. All of this subtlety makes a deterministic calculation of each potential ray's potential path impossible in practice.

The solution is Monte Carlo simulations. For instance, light sources are not single points. A Monte Carlo simulation can send virtual rays from the object to multiple random points on the light source – if, say, 100 rays are sent but 70 are blocked, then the pixel can be set at 30% brightness. This creates a much more realistic shadow gradient. Other simulations can help create the "colour bleed" effect of light bouncing off brightly coloured parts of a scene, like a verdant lawn or red brick wall. To deal with reflection, unless we want every object to act like an impossibly perfect mirror, we need a Monte Carlo simulation to create the effect of not-quite-smooth surfaces. Instead of an exact "angle of incidence equals angle of reflection" effect, the simulation randomly generates reflected rays within a small cone of the idealised reflection angle, to mimic microscopic imperfections in the surface. It can then average out the colours of the surfaces those rays hit, to create a realistically less reflective surface. Refraction can be incorporated similarly. There are thousands of simulated rays per pixel, and modern processors can extend the process to multiple steps, where rays are reflected and/or refracted dozens of times to build up an extremely rich and realistic image. A single frame of a Pixar movie can take days to render, even in 2026, because what they are doing now is so much more ambitious than it used to be, and it is only possible with the help of the Monte Carlo method.

The second example I'll discuss is one of the many applications of Monte Carlo simulations in medicine: the PET scan. "PET" stands for "positron emission tomography". Tomography, from the Greek *tomos* (slice) and *graphein* (to write or record), is imaging the body, or part of the body, by sections or slices. A CT (computed tomography) scan achieves this by firing X-rays from a rotating machine to build up a 3D model of the inside of the body. It is excellent for assessing bone, muscle, organs and spotting internal bleeding. But it is not very good at distinguishing healthy tissue from cancerous tissue. That's where a PET scan comes in. For a PET scan, the patient is injected with a small amount of a radioactive tracer, most commonly a modified glucose molecule called fluorodeoxyglucose, or FDG, which is disproportionately absorbed by cancer cells – they need the energy of glucose for their abnormal growth. The effect is that cancerous tumours show up as bright, glowing regions on the scan images. But how are these images created?

FDG is glucose ($C_6H_{12}O_6$) in which an OH has been replaced with an atom of radioactive isotope Fluorine-18 to make $C_6H_{11}{}^{18}FO_5$. Standard fluorine has ten neutrons, while Fluorine-18 only has nine (both, being forms of Fluorine, have nine protons). When it decays, one of the protons spontaneously converts into a neutron, which turns the Fluorine-18 into a stable Oxygen-18 isotope (eight protons, ten neutrons, compared to the usual eight of each), and a positron – an antielectron. (Actually, a neutrino is created at the same time but neutrinos are harmless so we can ignore it.) That's the "positron emission" part of the PET acronym. As soon as the positron encounters an electron, they will annihilate, producing two high-energy gamma-ray photons that zoom off in opposite directions. It's these photon pairs that the PET scanner detects. The software calculates where each annihilation took place, and from that builds up an image of the hotspots in the body, indicating tumours. The mathematics is happening in the phrase "the software calculates". There are several challenges preventing a neat deterministic solution to the question "where are the hotspots?" the first of which is the physical idiosyncrasies of each individual machine. The scanner has a cylindrical array of detectors, made up of tiny crystals. There are minuscule spaces between them and microscopic flaws inside them. The detectors themselves also have edges and gaps. No two machines are identical. So, large-scale Monte Carlo simulations are run on each machine, mapping trillions of virtual photon pairs hitting the scanner ring. Over time, a "system matrix" is built. It is a high-precision probability map showing the chance of a photon originating at each point being successfully detected at each scanner location. When the actual scans happen, this pre-existing map lets the software correct for the scanner's imperfections, so

that specific locations in the body are not over- or under-represented in the hotspot map.

The next challenge involves something that cannot be pre-calculated. When two photons are detected at (almost) the same instant, it's safe to deduce that they are a pair produced by a positron annihilation which sent them off in opposite directions. The annihilation would then have happened somewhere along the line joining the detectors that picked the photons up. Millions of these "lines of response" are produced to triangulate the tumour's location. But there's a problem. The photons may start out moving in exactly opposite directions, but they are often deflected by collisions with atoms inside the patient's body – this is called Compton scattering. So the lines of response are often incorrect, with a lot of random noise. The result is a blurry, poorly-defined image. To deal with this, the software runs a Monte Carlo simulation of hundreds of millions of virtual photons passing through the patient's body (which is mapped with a CT scan because every patient is different). This allows the noise to be removed, resulting in a much clearer and more accurate image.

The final piece of the puzzle is the problem that what we are detecting is not the creation of the positron, but its annihilation. The actual decay occurs in the FDG molecule, which is likely to be inside the tumour. But a newly created positron hurtles away from the nucleus in which it was made and can get quite a long way – up to a millimetre or two – before it happens to meet an electron. More Monte Carlo simulations, specific to different tissue densities in the human body (bone, lung, muscle and so on), are used in algorithms that sharpen the final image. PET scans save thousands of lives every year by giving medical teams a better understanding of their patients' cancers. They help detect cancers earlier and pinpoint their location better.

Conclusion

Selecting four "most important" or "best" algorithms of all time is, of course, impossible. I do not claim to have achieved, or even attempted, this. My choices were motivated by a desire to reflect the fundamental importance of algorithms, from the very beginnings of mathematics right through its entire history. A computer scientist, an engineer, or even a different mathematician, would likely make a different selection. Which algorithms would you have chosen?

Further Reading

1. The Babylonian tablet AO8862 is in the Louvre collection. The image I used is © 2015 GrandPalaisRmn (musée du Louvre) / Mathieu Rabeau. https://collections.louvre.fr/en/ark:/53355/cl010169558
2. For a deep dive into Mesopotamian mathematics, including a detailed analysis of AO8862, see the book *Algebra in Cuneiform: Introduction to an Old Babylonian Geometrical Technique* (2017) by noted Danish historian of mathematics Jens Høyrup. It is an open-access publication of the Max Planck Research Library for the History and Development of Knowledge (https://www.mprl-series.mpg.de/media/textbooks/2/textbooks2.pdf).
3. Background on early investigations into the complexity of the Euclidean algorithm is given in Jeffrey Shallit, "Origins of the analysis of the Euclidean algorithm", *Historia Mathematica*, Volume 21, Issue 4, 1994, 401 – 419, (https://www.sciencedirect.com/science/article/pii/S0315086084710317)
4. The Euclidean algorithm and music: G. T. Toussaint, "The Euclidean algorithm generates traditional musical rhythms", *Proceedings of BRIDGES: Mathematical Connections in Art, Music, and Science*, Banff, July 31 to August 3, 2005, pp. 47–56. (https://cgm.cs.mcgill.ca/~godfried/publications/banff.pdf)
5. The extent to which Gauss discovered the FFT is discussed in Heideman, Michael T., et al. "Gauss and the History of the Fast Fourier Transform." *Archive for History of Exact Sciences*, vol. 34, no. 3, 1985, pp. 265–77. (http://www.jstor.org/stable/41133773)
6. Cooley and Tukey's FFT paper is "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comput.* 19(2), 297–301 (1965). https://pubs.ams.org/MCOM/1965-19-090/S0025-5718-1965-0178586-1
7. There is a good exposition of Fourier analysis, the FFT and much else related to signal processing in music, in David Benson's excellent book *Music: A Mathematical Offering*, Cambridge University Press 2013.
8. The editors' introduction to the IEEE journal issue dedicated to the most influential algorithms of the 20th century is J. Dongarra and F. Sullivan, "Guest Editors Introduction to the top 10 algorithms," in *Computing in Science & Engineering*, vol. 2(1), 22-23 (2000). https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=814652
9. For more on how Monte Carlo simulations actually calculate probabilities, see R. L. Harrison, "Introduction to Monte Carlo simulation", AIP Conf Proc. 2010 Jan 5; 1204:17-21 (though don't believe the "Buffon's baguettes" part!) at https://pmc.ncbi.nlm.nih.gov/articles/PMC2924739/, and his linked paper on medical applications: R. L. Harrison, "Monte Carlo Simulation Of Emission Tomography And Other Medical Imaging Techniques", AIP Conf Proc. 2010 Jan 5;1204:126-132, https://pmc.ncbi.nlm.nih.gov/articles/PMC2924737/ (both are open access).
10. An entertaining takedown of some of the more excitable claims about Buffon's needle, read "Buffon: Did He Actually Throw Sticks?", by Ehrhard Behrends and Jorge Buescu, published in the European Mathematical Society Newsletter in 2014. (https://www.ehrhard-behrends.de/pdf_populaeres/buffon_ems_2014.pdf)