8 Everything Is a Number

In this digital age of ours we have grown accustomed to representing all forms of information as numbers. Text, drawings, photographs, sound, music, movies — everything goes into the digitization mill and gets stored on our computers and other devices in ever more complex arrangements of 0s and 1s.

In the 1930s, however, only numbers were numbers, and if somebody was turning text into numbers, it was for purposes of deception and intrigue.

In the fall of 1937, Alan Turing began his second year at Princeton amidst heightened fears that England and Germany would soon be at war. He was working on his doctoral thesis, of course, but he had also developed an interest in cryptology — the science and mathematics of creating secret codes or ciphers (cryptography) and breaking codes invented by others (cryptanalysis).[1] Turing believed that messages during wartime could be best encrypted by converting words to binary digits and then multiplying them by large numbers. Decrypting the messages without knowledge of that large number would then involve a difficult factoring problem. This idea of Turing's was rather prescient, for it is the way that most computer encryption works now.

Unlike most mathematicians, Turing liked to get his hands dirty building things. To implement an automatic code machine he began building a binary multiplier using electromagnetic relays, which were the primary building blocks of computers before vacuum tubes were demonstrated to be sufficiently reliable. Turing even built his own relays in a machine shop and wound the electromagnets himself.

The German Army and Navy were already using quite a different encrypting device. The *Enigma* was invented by a German electrical engineer named Arthur Scherbius (1878–1929). After Scherbius had unsuccessfully attempted to persuade the German Navy to use the machine in 1918, it had gone on sale for commercial

[1] Andrew Hodges, *Alan Turing The Enigma* (Simon & Schuster, 1983), 138

purposes in 1923. The Navy became interested soon after that, eventually followed by the rest of the German military.[2]

The Enigma had a rudimentary 26-key keyboard arranged like a typewriter but without numbers, punctuation, or shift keys. Above the keyboard were 26 light bulbs arranged in the same pattern. Messages were encrypted by typing them on the keyboard. As each letter was pressed, a different letter would light up. These lighted letters were manually transcribed and then sent to the recipient. (The encrypted message could be hand delivered or sent by mail; later, encrypted messages were sent by radio using Morse code.) The person receiving the message had his own Enigma machine, and would type the encrypted message on the keyboard. The flashing lights would then spell out the original text.

The keys of the keyboard were electrically connected to the lights through a series of rotors. Each rotor was a small disk with 26 contacts on each side representing the letters of the alphabet. Inside the rotor, these contacts were connected symmetrically: If contact A on one side connected to contact T on the other, then T on the first side would connect to A on the other. This symmetry is what allowed the machine to be used for both encrypting and decrypting.

The standard Enigma had three connected rotors, each of which was wired differently, and each of which could be set to one of 26 positions. The three rotors on the encrypting and decrypting machines had to be set identically. The three-letter keys to set the rotors could, for example, be changed on a daily basis in accordance with a list known only to the Enigma operators.

So far, nothing I've described about the Enigma makes it capable of anything more than a simple letter-substitution code, easily breakable by even the most amateur cryptanalysts. It's even simpler than most letter-substitution codes because it's symmetrical: If D is encoded as S then S is also encoded as D.

Here's the kicker: As the user of the Enigma pressed the keys on the keyboard, the rotors *moved*. With each keystroke, the first rotor moved ahead one position. If a string of 26 A's were typed, for example, each successive A would be encoded differently as the rotor went through its 26 positions. When the first rotor had completed a full turn, it would move the second rotor ahead one position. Now another series of 26 A's would encode to a different sequence of letters. When the second rotor finished a revolution, it would bump the third rotor up a notch. A fourth stationary rotor routed the electrical signal back through the rotors in reverse order. Only after 17,576 keystrokes (that's 26 to the third power) would the encryption pattern repeat.

But wait, it gets worse: The rotors were replaceable. The basic machine was supplied with five different rotors, which could be used in any of the three rotor

[2]David Kahn, *Seizing the Enigma. The Race to Break the German U-Boat Codes, 1939–1943* (Houghton-Mifflin, 1991), ch 3

slots. Another enhancement involved a plug-board that added another layer of letter scrambling.

In 1932, three Polish mathematicians began developing methods to decode Enigma messages.[3] They determined that they needed to build devices that simulated the Enigma in an automated manner. The first "bombs" (as they were called) became operational in 1938 and searched through possible rotor settings. One of these mathematicians was Marian Rejewski (1905–1980), who had spent a year at Göttingen after graduation. He wrote that the machines were called bombs "for lack of a better name"[4] but it's possible the name was suggested by the ticking sound they made, or by a particular ice cream sundae enjoyed by the mathematicians.[5]

Traditionally, the British government had employed classics scholars for breaking codes under the reasonable assumption that these were the people best trained to decode difficult languages. As the war approached, it became evident that for analyzing sophisticated encoding devices like the Enigma, the Government Code and Cypher School (GC & CS) would require mathematicians as well.

When Alan Turing returned from Princeton to England in the summer of 1938, he was invited to take a course at the GC & CS headquarters. It's possible the government was in touch with him as early as 1936.[6] In 1939, the GC & CS purchased a large estate with a Victorian mansion called Bletchley Park 50 miles northeast of London. In a sense, Bletchley Park was the intellectual focal point of England — where the rail line between Oxford and Cambridge connected with the rail south to London.

On September 1, 1939, Germany invaded Poland. Two days later, Great Britain declared war on Germany, and on September 4, Alan Turing reported for duty at Bletchley Park. Eventually about ten thousand people would be working there intercepting and decoding covert communications. To accommodate everyone, huts were built around the grounds. Turing was in charge of Hut 8, dedicated to the decryption of codes used by the German Navy. The Germans used these codes to communicate with submarines, which were a particular threat to convoys in the Atlantic between the United States and Great Britain.

Earlier in 1939, the British had met with the Polish mathematicians to learn about the Enigma and the bombs. Soon after Turing started at Bletchley Park, he began redesigning and improving the devices, now known by the French spelling *bombe*. The first Turing Bombe (as they are sometimes called) became operational

---

[3]Marian Rejewski, "How Polish Mathematicians Deciphered the Enigma," *Annals of the History of Computing*, Vol 3, No 3 (July 1981), 213–234 See also Elisabeth Rakus-Andersson, "The Polish Brains Behind the Breaking of the Enigma Code Before and During the Second World War," in Christof Teuscher, ed , *Alan Turing Life and Legacy of a Great Thinker* (Springer, 2004), 419–439

[4]Rejewski, "How Polish Mathematicians Deciphered the Enigma," 226

[5]Kahn, *Seizing the Enigma*, 73

[6]Hodges, *Alan Turing*, 148

in 1940. It weighed a ton and could simulate 30 Enigma machines working in parallel.[7]

Prior to attacking the message with the Turing Bombe, it was necessary to narrow down the possibilities. The cryptanalysts searched for "cribs," which were common words or phrases that often appeared in the encoded messages. These would establish the initial position of the first rotor. Much valued were cases where the same message was transmitted using two encodings: These were known as "kisses." Another technique used heavy white paper in various widths and printed with multiple rows of the alphabet, much like punched cards later used in computers. The analysts would punch holes in the paper corresponding to the letters of the encoded messages. Different messages from the same day (which would all be based on the same settings of the Enigma) could then be compared by overlapping the sheets. Because the paper used for this came from a nearby town named Banbury, the procedure was called "banburismus."

These varieties of techniques were refined to a point where, by mid-1941, the successes achieved in decoding Enigma communications had greatly decreased naval losses.[8] Many people working at Bletchley Park deserve some credit for this success, although Alan Turing's work played a significant role.

Even in the unusual assemblage of mathematicians and classics scholars at Bletchley Park, Turing established a certain reputation for eccentricity:

> In the first week of June each year [Turing] would get a bad attack of hay fever, and he would cycle to the office wearing a service gas mask to screen the pollen. His bicycle had a fault: the chain would come off at regular intervals. Instead of having it mended he would count the number of times the pedals went round and would get off the bicycle in time to adjust the chain by hand.[9]

In the spring of 1941, Alan Turing made a proposal of marriage to Joan Clarke, one of the rare women at Bletchley Park who wasn't relegated to a mindless clerical job. Joan Clarke had been studying mathematics at Cambridge when she was recruited for code-breaking. A few days after the proposal Turing confessed to her that he had "homosexual tendencies"[10] but the engagement continued for several more months before he felt he had to call it off.

---

[7]Stephen Budiansky, *Battle of Wits. The Complete Story of Codebreaking in World War II* (Free Press, 2000), 155 See also Jack Gray and Keith Thrower, *How the Turing Bombe Smashed the Enigma Code* (Speedwell, 2001)

[8]Hodges, *Alan Turing*, 218–9

[9]I J Good, "Early Work on Computers at Bletchley," *Annals of the History of Computing*, Vol 1, No 1 (July 1979), 41

[10]Hodges, *Alan Turing*, 206

In November 1942, Turing was sent on a mission to Washington, D.C., to help coordinate code-breaking activities between England and the United States. Following that assignment, he spent the first two months of 1943 at Bell Laboratories, at the time located on West Street in New York City. There he met Harry Nyquist (1889–1976), who pioneered the theory of digital sampling, and Claude Elwood Shannon (1916–2001), whose paper "A Mathematical Theory of Communication" (1948) would found the field of information theory and introduce the word "bit" to the world.

For Turing the primary object of interest at Bell Labs was a speech-scrambling device that was intended to secure telephone communications over the Atlantic. Sound waves were separated into various frequency ranges, digitized, and then encrypted by modular addition, which is addition that wraps around a particular value (such as the value 60 when adding seconds and minutes). On the receiving end, the numbers were decrypted and then reconstituted as speech.

In Nyquist's research and Shannon's work, and in the speech-encryption device, we can see the origin of ideas that would later result in the technologies used for digitizing images in JPEG files and sound in MP3 files, but these particular innovations required decades to come to fruition. The earliest digital computers, on the other hand, did little but emit numbers. Even Babbage's original Difference Engine was conceived solely to print error-free tables of logarithms. In this context, it's not surprising that Turing Machines also generate numbers rather than, for instance, implement generalized functions.

Turing is about to take the paper in a more unusual direction by using numbers to encode other forms of information. The next section of Turing's paper demonstrates how numbers can represent not photographs or songs, but the machines themselves.

Yes, everything is a number. Even Turing Machines are numbers.

> 5. *Enumeration of computable sequences.*
>
> A computable sequence γ is determined by a description of a machine which computes γ. Thus the sequence 001011011101111... is determined by the table on p. 234, and, in fact, any computable sequence is capable of being described in terms of such a table.

That's the Example II machine on page 87 of this book.

> It will be useful to put these tables into a kind of standard form.

Turing actually started out with a standard form that he described in Section 1 (page 70 of this book). He indicated that a particular operation can cause the machine to print or erase a symbol, and to move one square to the left or right.

After showing one machine in this format (Example I on page 81, the example that Turing will also soon mention), Turing quickly abandoned his own rules. He allowed printing multiple symbols and moving multiple squares in single operations. This was done solely so the machine tables didn't go on for pages and pages. Now he'd like to return to his original restrictions.

> In the first place let us suppose that the table is given in the same form as the first table, for example, I on p. 233. That is to say, that the entry in the operations column is always of one of the forms ***E***: ***E***, ***R***: ***E***, ***L***: *Pα*: *Pα*, ***R***: *Pα*, ***L***: ***R***: ***L***: or no entry at all.

Turing uses colons to separate the nine different possibilities. These possibilities result from the three types of printing (erase, print a character, or neither) in combination with the three kinds of movement (left, right, or none).

> The table can always be put into this form by introducing more *m*-configurations.

For example, the table for Example II (page 87) began with configuration 𝔟:

Configuration		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
𝔟		*P*ə, *R*, *P*ə, *R*, *P*0, *R*, *R*, *P*0, *L*, *L*	𝔬

To adhere to Turing's original (and reinstated) restrictions, this single configuration must be split into six simple configurations. For the additional configuration I'll use the German lower-case letters for c, d, e, g, and h (f was already used in the original table).

Configuration		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
𝔟		*P*ə, *R*	𝔠
𝔠		*P*ə, *R*	𝔡
𝔡		*P*0, *R*	𝔢
𝔢		*R*	𝔤
𝔤		*P*0, *L*	𝔥
𝔥		*L*	𝔬

Now each operation consists solely of a printing operation (or not) followed by possible left or right movement by one square.

> Now let us give numbers to the *m*-configurations, calling them $q_1, \ldots, q_R$, as in § 1. The initial *m*-configuration is always to be called q_1.

If there happen to be 237 different *m*-configurations in a machine, they are now to be labeled q_1 through q_{237}.

For the revised beginning of Example II, the first six *m*-configurations can be renamed q_1 through q_6. The initial *m*-configuration that Turing always named b becomes q_1. The table is now:

Configuration		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
q_1		P ə, R	q_2
q_2		P ə, R	q_3
q_3		P 0, R	q_4
q_4		R	q_5
q_5		P 0, L	q_6
q_6		L	q_7

> We also give numbers to the symbols $S_1, \ldots, S_m$
>
> [240]
>
> and, in particular, blank $= S_0, 0 = S_1, 1 = S_2$.

It's a little confusing that a subscripted 1 means the symbol 0 and a subscripted 2 means the symbol 1, but we'll have to live with it. The Example II machine also needs to print ə and *x*, so the following equivalencies would be defined for this machine:

S_0 means a blank,
S_1 means 0,
S_2 means 1,
S_3 means ə, and
S_4 means *x*.

The machine that computes the square root of 2 requires symbols up to S_{14}.

The first six configurations of the Example II machine are now:

Configuration		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
q_1		PS_3, R	q_2
q_2		PS_3, R	q_3
q_3		PS_1, R	q_4
q_4		R	q_5
q_5		PS_1, L	q_6
q_6		L	q_7

The imposition of a uniform naming system has resulted in these lines taking on very similar patterns. In the general case, Turing identifies three different standard forms:

The lines of the table are now of form

m-config.	*Symbol*	*Operations*	*Final m-config.*	
q_i	S_j	PS_k, L	q_m	(N_1)
q_i	S_j	PS_k, R	q_m	(N_2)
q_i	S_j	PS_k	q_m	(N_3)

At the far right, Turing has labeled these three standard forms N_1, N_2, and N_3. All three print something; the only difference is whether the head moves Left, Right, or not at all.

What about erasures? Because Turing defined S_0 as a blank symbol, erasures can be performed by printing simply S_0:

Lines such as

q_i	S_j	E, R	q_m

are to be written as

q_i	S_j	PS_0, R	q_m

Operations that consist of a Right or Left shift without printing anything can be written to reprint the scanned symbol:

> and lines such as
>
> $q_i \qquad S_j \qquad R \qquad q_m$
>
> to be written as
>
> $q_i \qquad S_j \qquad PS_j, R \qquad q_m$
>
> In this way we reduce each line of the table to a line of one of the forms $(N_1), (N_2), (N_3)$.

To illustrate the process of standardizing the table, I've been using the first configuration of the Example II table, but that first configuration doesn't even have anything in its *symbol* column because the configuration does the same thing regardless of the symbol. A machine starts with a blank tape so we know that the symbol it reads is a blank. The first configuration of the Example II table converted to standard form becomes:

Configuration		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
q_1	S_0	PS_3, R	q_2
q_2	S_0	PS_3, R	q_3
q_3	S_0	PS_1, R	q_4
q_4	S_0	PS_0, R	q_5
q_5	S_0	PS_1, L	q_6
q_6	S_0	PS_0, L	q_7

That's easy enough, but let's take a look at the second *m*-configuration of the Example II machine:

o	1	R, Px, L, L, L	o
	0		q

The *m*-configuration o will become the numbered configuration q_7. When the scanned character is 1, the head must move right once, and then left three times.

These three left-shifts will require three more *m*-configurations, q_8, q_9, and q_{10}. The *m*-configuration q then becomes q_{11}. Here's *m*-configuration q_7:

Configuration		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
q_7	S_2	PS_2, R	q_8
q_7	S_1	PS_1	q_{11}

In both cases, the machine prints the scanned character. Here are *m*-configurations q_8, q_9, and q_{10}:

q_8	S_0	PS_4, L	q_9
q_9	S_2	PS_2, L	q_{10}
q_{10}	S_0	PS_0, L	q_7

The problem is the *symbol* column. To fill it in correctly you really have to know what the machine will be encountering. For q_8, the machine is scanning a blank square and printing an *x*. Once it moves left, what's the next scanned character? It's the 1 that was scanned in q_7, but in other cases it might not be so obvious. The words "Any" or "Not" or "Else" don't work with this scheme, and in some cases you may have to add specific configurations for every single character the machine is using.

It's a mess, but there are always a finite number of characters involved, so it can definitely be done. Let's assume that we have converted all the configurations of a particular machine into the standard forms that Turing denotes as (N_1), (N_2), and (N_3). When we're finished, and we dispose of the original table, have we lost any information? Yes, we have lost a little bit. We know that S_0 is a blank, S_1 is a 0, and S_2 is a 1, but we no longer know the exact characters meant by S_3, S_4, and so on. This shouldn't matter. The machines use these characters internally. All that matters is that they're unique. We're really only interested in the 0s and 1s that the machine prints, and not what it uses as a scratchpad.

Instead of a table, we can express each configuration with a combination of the *m*-configurations, symbols, *L*, and *R*.

> From each line of form (N_1) let us form an expression $q_i\, S_j\, S_k\, L\, q_m$;

This form is sometimes known as a *quintuple* because it's composed of five elements. Despite its cryptic nature, it's still readable: "In *m*-configuration q_i, when character S_j is scanned, print character S_k, move Left, and switch to *m*-configuration q_m." Similarly for N_2 and N_3:

> from each line of form (N_2) we form an expression $q_i S_j S_k R q_m$; and from each line of form (N_3) we form an expression $q_i S_j S_k N q_m$.

Notice that when the head is not to be moved, the letter is *N* (meaning No move).

> Let us write down all expressions so formed from the table for the machine and separate them by semi-colons. In this way we obtain a complete description of the machine.

Turing will show an example shortly. Each configuration is a quintuple, and an entire machine is now expressed as a stream of quintuples. (Interestingly enough, the quintuples don't have to be in any specific order. It's like a programming language where each statement begins with a label and ends with a *goto*.)

The next substitution is a radical one. It gets rid of all those subscripts and turns the machine into a stream of capital letters:

> In this description we shall replace q_i by the letter "*D*" followed by the letter "*A*" repeated i times, and S_j by "*D*" followed by "*C*" repeated j times.

For example, q_1 is replaced by *DA* and q_5 is replaced by *DAAAAA*. (Remember that the first configuration is q_1. There is no q_0.) As for the symbols, S_0 (the blank) is now denoted by *D*, S_1 (the symbol 0) is *DC*, and S_2 (the symbol 1) is *DCC*. Other symbols are assigned to S_3 and greater and become *DCCC* and so on.

> This new description of the machine may be called the *standard description* (S.D). It is made up entirely from the letters "*A*", "*C*", "*D*", "*L*", "*R*", "*N*", and from ";".

The *L*, *R*, and *N* indicate the moves. Semicolons separate each configuration.

> If finally we replace "*A*" by "1", "*C*" by "2", "*D*" by "3", "*L*" by "4", "*R*" by "5", "*N*" by "6", and ";" by "7" we shall have a description of the machine in the form of an arabic numeral.

This is an important step. Turing has standardized his machines to such an extent that he can now uniquely identify a machine by an integer, and this

integer encodes all the states of the machine. Turing was undoubtedly inspired by the approach Gödel took in his Incompleteness Theorem in converting every mathematical expression into a unique number.

> The integer represented by this numeral may be called a *description number* (D.N) of the machine. The D.N determine the S.D and the structure of the
>
> [241]
>
> machine uniquely. The machine whose D.N is n may be described as $\mathcal{M}(n)$.

Turing has now introduced another font. He will use this script font for representing entire machines.

> To each computable sequence there corresponds at least one description number, while to no description number does there correspond more than one computable sequence.

Since the order of the quintuples doesn't matter, the quintuples can be scrambled without any effect on the sequence the machine computes. It is very clear, then, that multiple description numbers are associated with each computable sequence, but each description number defines a machine that generates only one computable sequence (at least when beginning with a blank tape).

Without much fanfare Turing concludes with a result he mentioned in the very beginning of the article:

> The computable sequences and numbers are therefore enumerable.

You can enumerate the computable sequences by listing all possible description numbers, since these are just integers. The unstated implication is that the computable numbers are only an enumerable subset of the real numbers. Because the computable numbers are enumerable and the real numbers are not, there are many real numbers that are not computable. This, however, is a subject that will be explored more in later sections.

> Let us find a description number for the machine I of § 3.

That machine was originally defined by this table:

Configuration		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
𝔟	None	*P*0, *R*	𝔠
𝔠	None	*R*	𝔢
𝔢	None	*P*1, *R*	𝔣
𝔣	None	*R*	𝔟

When we rename the *m*-configurations its table becomes:

q_1	S_0	PS_1, R	q_2
q_2	S_0	PS_0, R	q_3
q_3	S_0	PS_2, R	q_4
q_4	S_0	PS_0, R	q_1

This is a very straightforward translation.

Other tables could be obtained by adding irrelevant lines such as

q_1	S_1	PS_1, R	q_2

That is, *other tables that produce the same computable sequence* could be obtained by adding lines that never come into play. If the tape is blank when the machine begins, and it always shifts right when a square is printed, the machine will never scan the digit 0.

Our first standard form would be

$$q_1\, S_0\, S_1\, R\, q_2;\ q_2\, S_0\, S_0\, R\, q_3;\ q_3\, S_0\, S_2\, R\, q_4;\ q_4\, S_0\, S_0\, R\, q_1\ ;.$$

That's just taking the four-line table and separating the configurations with semicolons. Converting this to the Standard Description form requires replacing q_i with D followed by a quantity of i A's (one or more) and replacing S_i with D followed by i C's (zero or more).

The standard description is

DADDCRDAA ;DAADDRDAAA ;

DAAADDCCRDAAAA ;DAAAADDRDA ;

The Standard Description can be hard to read, but it's used a lot so you should try to get accustomed to it. To decode it into its components, begin by taking note of each *D*. Each *D* represents either a configuration or a symbol.

- If the *D* is followed by one or more *A*'s, it's a configuration. The configuration number is the number of *A*'s.
- If the *D* is *not* followed by any *A*'s, it's a symbol. The *D* in this case is followed by 0 or more *C*'s. *D* by itself is a blank, *DC* is a 0, *DCC* is a 1, and more *C*'s indicate other symbols.

Turing does not use the Description Number as much as the Standard Description. The Description Number exists more in abstract; Turing doesn't perform any calculations with the number. For the example Turing is showing, you can replace *A* with 1, *C* with 2, *D* with 3, *R* with 5 and the semicolon with 7 to create a description number:

A description number is

3133253117311335311173111332253111173111133531731323253117

and so is

313325311731133531117311133225311117311113353173132325311 7

The second of those numbers is the same as the first except it has extra digits at the end (31323253117) corresponding to the "irrelevant" configuration $q_1 S_1 S_1 R q_2$ that Turing defined. The point is this: These two numbers define two different machines, but the two machines both compute exactly the same number, which (as you'll recall) is the binary version of 1/3. A machine with its configurations rearranged still calculates the same number, but its Description Number is different.

These numbers are huge! Turing obviously doesn't care how large the numbers are. To represent q_{35}, for example, he might have figured out some way to embed the number 35 in the Description Number, but no. To represent q_{35}, the Standard Description uses:

DAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

and the Description Number includes the digits

3111111111111111111111111111111111

not only once, but at least twice!

The accomplishment here is quite interesting. Consider a Turing Machine that calculates π. Normally, we indicate the digits of π with an infinite sequence:

$$\pi = 3.14159265358979323846264338327 95\ldots$$

Now we can represent π with a *finite* integer — the Description Number of the Turing Machine that calculates the digits. Which is the better representation of π? The first 32 digits followed by an ellipsis? Or the Description Number of the Turing Machine that can generate as many digits as our patience will allow? In a sense, the Description Number is a more fundamental numerical representation of π because it describes the algorithm of calculating the number.

By reducing each machine to a number, Turing has also made it possible, in effect, to generate machines just by enumerating the positive integers. Not every positive integer is a valid Description Number of a Turing Machine, and many valid Description Numbers do not describe circle-free machines, but this enumeration certainly includes all circle-free Turing Machines, each of which corresponds to a computable number. Therefore, computable numbers are enumerable.

That's an important finding, although possibly a disturbing one, for it implies that most — nay, from what we know about the extent of the real numbers, *virtually all* — real numbers are not computable.

This revelation, combined with some mathematical paradoxes and investigations into quantum gravity, have prompted mathematician Gregory Chaitin to ask "How Real are Real Numbers?"[11] The evidence of the existence of real numbers is slim indeed.

To modern programmers it is natural to think of computer programs being represented by numbers, because a program's executable file is simply a collection of consecutive bytes. We don't normally think of these bytes as forming a single number, but they certainly could. For example, the Microsoft Word 2003 executable is the file WinWord.exe, and that file is 12,047,560 bytes in size. That's about 96 million bits, or 29 million decimal digits, so the number representing WinWord.exe is somewhere in the region of $10^{29,000,000}$. That's certainly a big number. In a book of about 50 lines per page and 50 digits per line, that number would stretch out over more than 11,000 pages. That's a much larger number than the famed googol (10^{100}), but it's still a finite integer. WinWord.exe is one of many possible executables that — like all the possible Turing Machines — turn up in

[11] Gregory J Chaitin, "How Real are Real Numbers?", *International Journal of Bifurcation and Chaos*, Vol 16 (2006), 1841–1848 Reprinted in Gregory J Chaitin, *Thinking About Godel and Turing Essays on Complexity, 1970–2007* (World Scientific, 2007), 267–280

an enumeration of the integers, along with every other word processing program, even those that haven't yet been written.

For future use, Turing finishes this section with a definition.

> A number which is a description number of a circle-free machine will be called a *satisfactory* number. In § 8 it is shown that there can be no general process for determining whether a given number is satisfactory or not.

It's easy to determine whether a particular integer is a well-formed Description Number, but Turing is now asserting that there's no general process to determine whether a particular Description Number represents a circle-free machine and prints a continuing series of 0s and 1s like it's supposed to. There's no general process for determining whether the machine might scan a character it's not expecting, or gets into an infinite loop printing blanks, whether it crashes, burns, goes belly up, or ascends to the great bit bucket in the sky.

9 The Universal Machine

The machine that Turing describes in the next section of his paper is known today as the Universal Turing Machine, so called because it's the only machine we need. The individual computing machines presented earlier were not guaranteed to be implemented similarly or even to have interchangeable parts. This Universal Machine, however, can simulate other machines when supplied with their Standard Descriptions. The Universal Machine is, we would say today, *programmable*.

> 6. *The universal computing machine.*
>
> It is possible to invent a single machine which can be used to compute any computable sequence. If this machine $\mathcal{U}$ is supplied with a tape on the beginning of which is written the S.D of some computing machine $\mathcal{M}$,
>
> [242]
>
> then $\mathcal{U}$ will compute the same sequence as $\mathcal{M}$. In this section I explain in outline the behaviour of the machine. The next section is devoted to giving the complete table for $\mathcal{U}$.

There's that script font again. Turing uses $\mathcal{M}$ for an arbitrary machine and $\mathcal{U}$ for the Universal Machine.

When speaking of computer programs, it's common to refer to *input* and *output*. A program reads input and writes output. The machines described so far basically have no input because they begin with a blank tape. The machines generate output in the form of a sequence of 0s and 1s, temporarily interspersed, perhaps, with some other characters used as markers or a scratchpad.

In contrast, the Universal Machine $\mathcal{U}$ requires actual input, specifically a tape that contains the Standard Description of $\mathcal{M}$ — the sequences of letters *A*, *C*, *D*, *L*, *N*, and *R* that describe all the configurations of $\mathcal{M}$. The $\mathcal{U}$ machine reads and interprets that Standard Description and prints the same output that $\mathcal{M}$ would print.

But that's not entirely true: The output of $\mathcal{U}$ will *not* be identical to the output of $\mathcal{M}$. In the general case, there is no way that $\mathcal{U}$ can perfectly mimic $\mathcal{M}$. Machine $\mathcal{M}$ probably begins with a blank tape, but machine $\mathcal{U}$ doesn't get a blank tape — it gets a tape with the Standard Description of $\mathcal{M}$ already on it. What happens if $\mathcal{M}$ doesn't quite follow Turing's conventions but instead writes output in both directions? Any attempt to emulate $\mathcal{M}$ precisely could easily result in writing over that Standard Description.

Turing says that $\mathcal{U}$ is supplied with a tape "on the beginning of which" is a Standard Description of machine $\mathcal{M}$. A tape that is infinite in both directions does not have a "beginning." Turing is implicitly restricting the output of $\mathcal{U}$ to that part of the tape after the Standard Description.

If we limit our consideration to machines that print in only one direction (which is Turing's convention anyway), can we write a Universal Machine that reads the Standard Description of the machine located at the beginning of a tape, and then exactly duplicates the output of the machine in the infinite blank area of the tape beyond that Standard Description?

That doesn't seem likely either. Certainly this Universal Machine would require its own scratchpad area, so its output will be different from the machine that it's trying to emulate. Even if we require only that the Universal Machine duplicate the $\mathcal{M}$ machine's *F*-squares, that Universal Machine would probably be significantly more complex than the one that Turing describes.

Turing doesn't guarantee that his Universal Machine will faithfully duplicate the output of the machine that it is emulating. He says only that "$\mathcal{U}$ will compute the same sequence as $\mathcal{M}$." In reality, $\mathcal{U}$ prints a lot of extra output *in addition to* this sequence.

Turing approaches the design of the Universal Machine from a rather odd direction.

> **Let us first suppose that we have a machine $\mathcal{M}'$ which will write down on the *F*-squares the successive complete configurations of $\mathcal{M}$.**

As you'll recall, a complete configuration is a "snapshot" of the tape after an operation has completed, together with the position of the head and the next *m*-configuration. The successive complete configurations provide an entire history of the operations of the machine.

> **These might be expressed in the same form as on p. 235, using the second description, (C), with all symbols on one line.**

That's page 92 of this book, in the form that shows the information in a single stream:

$$\mathfrak{b} : \mathfrak{a}\mathfrak{a}\mathfrak{o}0 \quad 0 : \mathfrak{a}\mathfrak{a}\mathfrak{q}0 \quad 0 : \ldots$$

In this notation the successive complete configurations are separated by colons. Within each complete configuration, the German letter representing the next *m*-configuration precedes the next scanned symbol.

> Or, better, we could transform this description (as in §5) by replacing each *m*-configuration by "*D*" followed by "*A*" repeated the appropriate number of times, and by replacing each symbol by "*D*" followed by "*C*" repeated the appropriate number of times. The numbers of letters "*A*" and "*C*" are to agree with the numbers chosen in §5, so that, in particular, "0" is replaced by "*DC*", "1" by "*DCC*", and the blanks by "*D*".

Turing devised this Standard Description (as he called it) to encode the states of a machine. Now he is proposing to use it to represent the complete configurations.

> These substitutions are to be made after the complete configurations have been put together, as in (C). Difficulties arise if we do the substitution first.

I think what Turing means here is that *m*-configurations and symbols will now be represented with multiple symbols (for example a 1 becomes *DCC*), so care must be taken to slip in the next *m*-configuration so that it doesn't break up the code for a symbol.

> In each complete configuration the blanks would all have to be replaced by "*D*", so that the complete configuration would not be expressed as a finite sequence of symbols.

The letter *D* represents a blank square. Turing doesn't want any breaks to appear in the complete configurations. He wants each complete configuration to be an unbroken series of letters. Turing's phrase, "so that the complete configuration would not be expressed as a finite sequence of letters," is not quite clear. I suggest the word "not" should be "now." Certainly he doesn't want an infinite series of *D* symbols to represent a blank tape. Each complete configuration is finite.

> If in the description of the machine II of §3 we replace "𝔟" by "*DAA*", "𝔞" by "*DCCC*", "𝔮" by "*DAAA*", then the sequence (C) becomes:
>
> $$DA : DCCCDCCCDAADCDDC : DCCCDCCCDAAADCDDC : \ldots (C_1)$$
>
> (This is the sequence of symbols on *F*-squares.)

Turing's not mentioning *all* the substitutions he's making. He's also replacing 𝔟 with *DA*, blanks with *D*, and 0s with *DC*.

The parenthetical comment refers to the output of the $\mathcal{M}'$ machine that Turing is proposing. The normal $\mathcal{M}$ machine prints 0s and 1s on *F*-squares and uses the *E*-squares for other symbols to help it in computing the 0s and 1s. The $\mathcal{M}'$ machine prints the successive complete configurations of $\mathcal{M}$ on *F*-squares and uses the *E*-squares to aid itself in constructing these successive complete configurations.

The complete configurations represented in this way can be hard to read. As I've said before, it helps to take note of each *D*, which represents either a configuration or a symbol.

- If the *D* is followed by one or more *A*'s, it's a configuration. The configuration number is the number of *A*'s.
- If the *D* is *not* followed by any *A*'s, it's a symbol. The *D* in this case is followed by zero or more *C*'s. *D* by itself is a blank, *DC* is a 0, *DCC* is a 1, and more *C*'s indicate other symbols.

> It is not difficult to see that if $\mathcal{M}$ can be constructed, then so can $\mathcal{M}'$. The manner of operation of $\mathcal{M}'$ could be made to depend on having the rules of operation (*i.e.*, the S.D) of $\mathcal{M}$ written somewhere within itself (*i.e.* within $\mathcal{M}'$); each step could be carried out by referring to these rules.

This idea of $\mathcal{M}'$ having the Standard Description of $\mathcal{M}$ "written somewhere within itself" is an entirely new concept. Where is it written? How is it accessed? Turing is pursuing this $\mathcal{M}'$ machine in a way that's distracting from his goal, although it does seem reasonable that $\mathcal{M}'$ could be constructed.

> We have only to regard the rules as being capable of being taken out and exchanged for others and we have something very akin to the universal machine.

Ahh, now it becomes a little clearer. Turing said at the outset of this section that $\mathcal{U}$ is supplied with a tape containing the Standard Description of $\mathcal{M}$. That's what "capable of being taken out and exchanged for others" means. We can give $\mathcal{U}$ a tape containing the Standard Description of whatever machine we want $\mathcal{U}$ to emulate.

Conceptually, $\mathcal{U}$ now seems almost, well, not *exactly* straightforward, but much less difficult. $\mathcal{U}$ starts with a tape on which the Standard Description of $\mathcal{M}$ is printed. It is responsible for printing the successive complete configurations of $\mathcal{M}$. The Standard Description and the complete configurations use the same encoding: Each complete configuration contains a sequence of letters, mostly indicating the symbols printed on the tape. Each complete configuration also includes a *D* followed by one or more *A*'s indicating the next *m*-configuration preceding the scanned symbol, for example:

DAAADCC

This sequence of letters appearing in a complete configuration indicates that the next *m*-configuration is q_3 and the next scanned symbol is a 1. Somewhere in the Standard Description of $\mathcal{M}$ is a sequence of letters matching these letters exactly. (If not, then something has gone wrong, and $\mathcal{M}$ is not circle-free.) All that $\mathcal{U}$ needs to do to determine the next configuration is to find a match. When $\mathcal{U}$ finds the matching configuration, it has immediate access to the configuration's operation — the symbol to be printed, a code indicating how to move the head, and the next *m*-configuration. $\mathcal{U}$ must then create a new complete configuration based on the last complete configuration and incorporating the printed character and the next *m*-configuration.

The Universal Machine might be easier to conceive if you consider that the first complete configuration of a machine's operation is trivial, and each step from one complete configuration to the next involves only a small change. It's really just a matter of comparing and copying symbols, and Turing has already defined an arsenal of *m*-functions that perform these very chores.

For now he's still talking about $\mathcal{M}'$ rather than $\mathcal{U}$, and $\mathcal{M}'$ only prints the complete configurations of $\mathcal{M}$.

> One thing is lacking : at present the machine $\mathcal{M}'$ prints no figures.

That's true. In all this excitement we've forgot that $\mathcal{M}'$ (or $\mathcal{U}$) is only printing successive complete configurations of $\mathcal{M}$ on *F*-squares using letters *A*, *C*, and *D* and the colon separators, and it's probably using *E*-squares as a scratch pad. The real object of this game is to print 0s and 1s.

> We may correct this by printing between each successive pair of complete configurations the figures which appear in the new configuration but not in the old. Then (C_1) becomes
>
> $$DDA : 0 : 0 : DCCCDCCCDAADCDDC : DCCC.... \qquad (C_2)$$
>
> It is not altogether obvious that the ***E***-squares leave enough room for the necessary "rough work", but this is, in fact, the case.

The extra *D* at the beginning of line (C_2) is a typographical error. The only difference between (C_2) and the beginning of (C_1) should be the two 0s and the colons. These are the result of the first operation, so they are printed after the first complete configuration.

Turing wants $\mathcal{M}'$ (and $\mathcal{U}$) to print the same 0s and 1s that $\mathcal{M}$ prints, because then it's possible to say that $\mathcal{M}'$ computes the same sequence as $\mathcal{M}$. The only difference is that these digits will now be buried in the output between successive complete configurations of the machine.

This is why Turing requires his machines to print the computed numbers consecutively, and to not change a number once it's been printed. Without this requirement, the numbers printed by $\mathcal{M}'$ (and $\mathcal{U}$) would be a total jumble.

Turing says that $\mathcal{M}'$ should print all figures (0s or 1s) "which appear in the new configuration but not in the old." When you reduce a machine to the standard form (that is, only one printed symbol and one head movement per operation), there are frequently occasions when the machine scans a 0 or 1 symbol on its way somewhere else. The machine must reprint the 0 or 1 in these cases. $\mathcal{M}'$ should ignore the times that $\mathcal{M}$ prints a 0 or 1 over itself. $\mathcal{M}'$ (and, by implication, the Universal Machine) should print a 0 or 1 *only when the scanned symbol is a blank.*

Turing concludes this section by suggesting that the complete configurations could be expressed in numerical form, but this is something he never uses:

> The sequences of letters between the colons in expressions such as (C_1) may be used as standard descriptions of the complete configurations. When the letters are replaced by figures, as in §5, we shall have a numerical
>
> [243]
>
> description of the complete configuration, which may be called its description number.

Now let's forget all about $\mathcal{M}'$ and start looking at $\mathcal{U}$.

It is well known that Turing's description of the Universal Machine contains a few bugs. (It's quite surprising how few bugs it contains considering that Turing wasn't able to simulate it on a real computer.) In analyzing the Universal Machine, I am indebted to Emil Post's corrections[1] and an analysis by Donald Davies.[2]

Because the Universal Machine is so essential to Turing's arguments in the rest of his paper, he proves the existence of such a machine by actually constructing it in full, excruciating detail. Once you understand the basic mechanism, however, you might find these details to be rather tedious. No one will punish you if you don't assimilate every symbol and function in Turing's description.

> 7. *Detailed description of the universal machine.*
>
> A table is given below of the behaviour of this universal machine. The *m*-configurations of which the machine is capable are all those occurring in the first and last columns of the table, together with all those which occur when we write out the unabbreviated tables of those which appear in the table in the form of *m*-functions. ***E.g.***, $\mathfrak{e}$ (anf) appears in the table and is an *m*-function.

The *m*-configuration anf is part of Turing's Universal Machine. Towards the end of the machine, a particular configuration has $\mathfrak{e}$(anf) in its *final m-config* column. The skeleton table for $\mathfrak{e}$ appears on page 239 of Turing's paper (and page 125 of this book):

$\mathfrak{e}(\mathfrak{C})$	ə	*R*	$\mathfrak{e}_1(\mathfrak{C})$
	Not ə	*L*	$\mathfrak{e}(\mathfrak{C})$
$\mathfrak{e}_1(\mathfrak{C})$	Any	*R*, *E*, *R*	$\mathfrak{e}_1(\mathfrak{C})$
	None		$\mathfrak{C}$

[1] In an appendix to the paper Emil Post, "Recursive Unsolvability of a Problem of Thue," *The Journal of Symbolic Logic*, Vol. 12, No 1 (Mar 1947), 1–11 The entire paper is reprinted in Martin Davis, ed., *The Undecidable* (Raven Press, 1965), 293–303. The appendix is reprinted in B Jack Copeland, ed , *The Essential Turing* (Oxford University Press, 2004), 97–101

[2] Donald W Davies, "Corrections to Turing's Universal Computing Machine" in C. Jack Copeland, ed., *The Essential Turing*, 103-124 Anyone interested in programming a simulation of the Universal Machine will want to study Davies' paper

Turing now shows the unabbreviated table when anf is substituted for 𝔈:

> Its unabbreviated table is (see p. 239)
>
> | | | | |
> |---|---|---|---|
> | 𝔢(anf) | ə | *R* | 𝔢1(anf) |
> | | not ə | *L* | 𝔢(anf) |
> | 𝔢1(anf) | Any | *R*, *E*, *R* | 𝔢1(anf) |
> | | None | | anf |
>
> Consequently 𝔢1(anf) is an *m*-configuration of 𝔘.

Turing begins by describing a tape encoded with the Standard Description of some machine. This is the tape the Universal Machine will read and interpret.

> When 𝔘 is ready to start work the tape running through it bears on it the symbol ə on an *F*-square and again ə on the next *E*-square; after this, on *F*-squares only, comes the S.D of the machine followed by a double colon "::" (a single symbol, on an *F*-square). The S.D consists of a number of instructions, separated by semi-colons.

That, by the way, is Turing's first use of the word *instructions* in this paper. The word is appropriate here because the configurations of the machines are now playing a different role; they have become instructions to the Universal Machine.

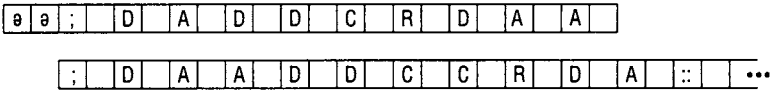
Earlier (in Section 5 on page 140 of this book) Turing showed each configuration followed by a semicolon, however, the Universal Machine requires that each instruction *begin* with a semicolon. This is just one of several little "bugs" in the description of the Universal Machine.

To illustrate the workings of 𝔘, let's supply it with a simple 𝔐. This machine is a simplified form of the machine that prints alternating 0s and 1s:

m-config.	*symbol*	*operations*	*final m-config.*
q_1	S_0	PS_1, R	q_2
q_2	S_0	PS_2, R	q_1

This simplified machine has just two configurations rather than four and doesn't skip any squares. Here's a tape prepared in accordance with Turing's directions,

but with the semicolons preceding each instruction. Because the tape is so long, I've shown it on two lines:



The double colon separates the instructions of $\mathcal{M}$ from the successive complete configurations of $\mathcal{M}$ that $\mathcal{U}$ will print. Turing reminds us how these instructions are coded:

> Each instruction consists of five consecutive parts
>
> (i) "*D*" followed by a sequence of letters "*A*". This describes the relevant *m*-configuration.

At least one *A* must follow a *D* to signify an *m*-configuration; that is, the configurations begin at q_1 and there is no q_0.

> (ii) "*D*" followed by a sequence of letters "*C*". This describes the scanned symbol.

For symbols, a *D* by itself means a blank; a *D* with one *C* means 0, and with two *C*'s means 1.

> (iii) "*D*" followed by another sequence of letters "*C*". This describes the symbol into which the scanned symbol is to be changed.
>
> (iv) "*L*", "*R*", or "*N*", describing whether the machine is to move to left, right, or not at all.
>
> (v) "*D*" followed by a sequence of letters "*A*". This describes the final *m*-configuration.

The Universal Machine needs to print complete configurations, which require the letters *A*, *C*, and *D*, and it also needs to print the computable sequence, which is composed of 0s and 1s. The Universal Machine uses lower-case letters as markers in the *E*-squares. In summary:

> The machine $\mathcal{U}$ is to be capable of printing "*A*", "*C*", "*D*", "0", "1", "*u*", "*v*", "*w*", "*x*", "*y*", "*z*".

Turing forgot to include the colon (which separates the successive complete configurations) in this list.

> The S.D is formed from ";", "*A*", "*C*", "*D*", "*L*", "*R*", "*N*".

Turing next presents one last function that the Universal Machine requires.

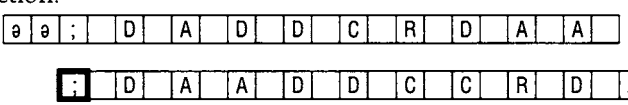
> [244]
>
> *Subsidiary skeleton table.*
>
> $\mathrm{con}(\mathfrak{C}, \alpha)$ { Not *A* *R*, *R* $\mathrm{con}(\mathfrak{C}, \alpha)$; *A* *L*, *P*α, *R* $\mathrm{con}_1(\mathfrak{C}, \alpha)$ }
>
> $\mathrm{con}(\mathfrak{C}, \alpha)$. Starting from an *F*-square, *S* say, the sequence *C* of symbols describing a configuration closest on the right of *S* is marked out with letters α. $\rightarrow \mathfrak{C}$.
>
> $\mathrm{con}_1(\mathfrak{C}, \alpha)$ { *A* *R*, *P*α, *R* $\mathrm{con}_1(\mathfrak{C}, \alpha)$; *D* *R*, *P*α, *R* $\mathrm{con}_2(\mathfrak{C}, \alpha)$ }
>
> $\mathrm{con}_2(\mathfrak{C}, \alpha)$ { *C* *R*, *P*α, *R* $\mathrm{con}_2(\mathfrak{C}, \alpha)$; Not *C* *R*, *R* $\mathfrak{C}$ }
>
> $\mathrm{con}(\mathfrak{C},)$. In the final configuration the machine is scanning the square which is four squares to the right of the last square of *C*. *C* is left unmarked.

The *m*-function con stands for "configuration," and it's missing a line[3]:

$\mathrm{con}_1(\mathfrak{C}, \alpha)$ None *PD*, *R*, *P*α, *R*, *R*, *R* $\mathfrak{C}$

We'll see how this missing line comes into play shortly.

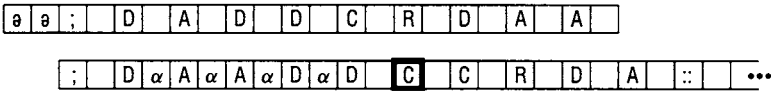
The job of the con function is to mark a configuration with the symbol given as the second argument. Suppose the head is on the semicolon preceding an instruction:



The con function moves right two squares at a time until it encounters an *A*. It prints an α to the left of the *A*. The con_1 function continues printing markers to the right of each *A* until it encounters a *D*. It prints a marker to the right of that

[3] As suggested by Post, "Recursive Unsolvability of a Problem of Thue," 7

D as well and then goes to con_2. The con_2 function prints markers to the right of each *C* (if any). For this example, there are no *C*'s in the configuration because the scanned square is a blank, so the result is:



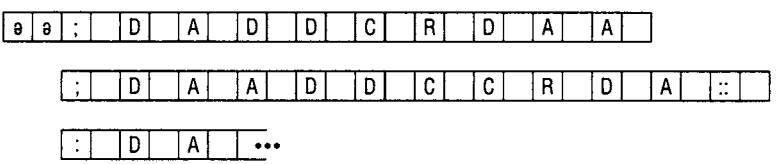
The explanatory paragraphs in the skeleton table for con are a bit confusing because Turing uses the letter *C* to stand for a whole sequence of symbols defining a configuration, and the same letter is part of the Standard Description. The first sentence of the second paragraph (beginning "In the final configuration") indicates that the head is left four squares to the right of the last square of the configuration (that is, the last square of the scanned character). The sentence "*C* is left unmarked" meaning "The configuration is left unmarked" applies only when the second argument to con is blank.

The description of the Universal Machine occupies just two pages in Turing's paper. Turing has previously defined his *m*-functions with such skill that in many cases, the *m*-configurations of $\mathcal{U}$ simply refer to a particular function. As usual, the machine begins with *m*-configuration $\mathfrak{b}$:

The table for $\mathcal{U}$.

$\mathfrak{b}$	$f(\mathfrak{b}_1, \mathfrak{b}_1, ::)$	$\mathfrak{b}$. The machine prints :*DA* on the *F*-squares after :: → anf.
$\mathfrak{b}_1$	*R, R, P:, R, R, PD, R, R, PA* anf	

The *m*-function f finds the double colon that separates the instructions from the complete configurations. As you'll recall, each complete configuration shows all the symbols on the tape, with the *m*-configuration preceding the scanned square. When a machine begins, the first *m*-configuration is q_1, which has a Standard Description of *DA*. That's what $\mathfrak{b}_1$ prints, starting with a colon that will delimit each complete configuration:

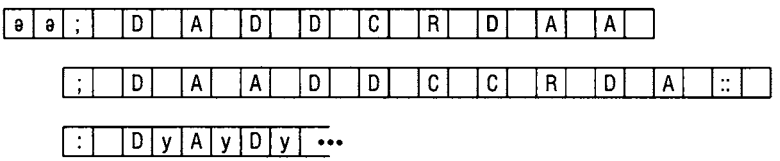


The next *m*-configuration of $\mathcal{U}$ is anf, which Donald Davies suggests stands for *anfang*, the German word for *beginning*. The g function in the first line was

mistakenly indicated as q in the tables of functions. It searches for the last occurrence of its second argument:

anf	g(anf$_1$, :)		anf. The machine marks the configuration in the last complete configuration with y. → fom.
anf$_1$	con(fom, y)		

After g finds the colon (which precedes the current complete configuration), con marks the *m*-configuration with the letter *y*. The additional line I've added to con$_1$ also comes into play: It prints a *D* (representing a blank square) and marks that square as well:

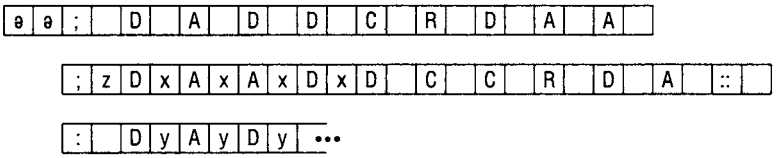


Whenever con is marking an *m*-configuration in a complete configuration and comes to a blank square when it is expecting to find a *D* that represents the scanned symbol, con$_1$ prints a *D*. This is how the tape gets progressively longer as more squares are required.

Now the machine must locate the instruction whose configuration matches the symbols in the complete configuration marked with *y*. There are multiple instructions, of course, but they are easy to locate because each one is preceded by a semicolon. These instructions are tested starting with the last instruction and working towards the beginning. The *m*-configuration fom looks like *fom* but is actually *kom*, possibly one of several abbreviations meant to suggest the word *compare*.

fom	;	R, Pz, L	con(fmp, x)	fom. The machine finds the last semi-colon not marked with z. It marks this semi-colon with z and the configuration following it with x.
	z	L, L	fom	
	not z nor ;	L	fom	

The first time through, fom finds the last (rightmost) instruction, prints a z following the semicolon, and then marks the configuration that follows using con.



The z marker indicates that this instruction has been checked. On subsequent attempts to find a match, fom skips past all semicolons previously marked with z.

The m-configuration fmp (another abbreviation for *compare*?) uses cpe to compare the configuration marked x (which is the m-configuration and scanned symbol of an instruction) and the configuration marked y (which is current m-configuration and scanned symbol indicated in the complete configuration):

fmp	cpe(e(fom, x, y), sim, x, y)	fmp. **The machine compares the sequences marked x and y. It erases all letters x and y. → sim if they are alike. Otherwise → fom.**

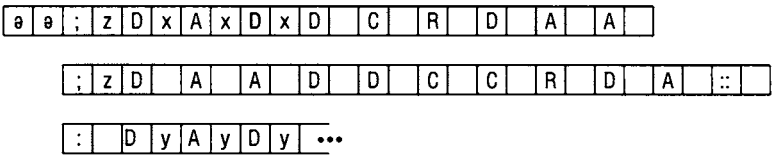
The cpe function erases the markers as it compares the letters marked with those markers. If there's a match, then all the x and y markers have been erased, and we head to sim (meaning *similar*).

If the configurations marked x and y do not match (as they won't in our example), then the first argument of cpe takes over, which is an e (erase) function that erases all the remaining x and y markers and eventually retreats to fom to try the next instruction.

A little problem with the fmp function is that Turing never defined a version of e that has one m-configuration argument and two symbol arguments. Moreover, he can't go back to fom because some or all of the y markers have been erased by cpe. He really needs to go back to anf to mark the configuration again. Donald Davies suggests that the instruction should really read:

fmp cpe(e(e(anf, x), y), sim, x, y)

In our example, anf$_1$ will re-mark the *m*-configuration and scanned symbol in the complete configuration, and fom will mark the next instruction (working backwards through the instructions):



This time, the cpe function invoked by fmp will detect a match and head to əim. All the *x* and *y* markers will be gone, but the *z* markers remain. The leftmost *z* marker precedes the instruction that $\mathcal{U}$ must carry out. Turing summarizes the progress so far:

> anf. **Taking the long view, the last instruction relevant to the last configuration is found. It can be recognised afterwards as the instruction following the last semi-colon marked *z*.** → əim.

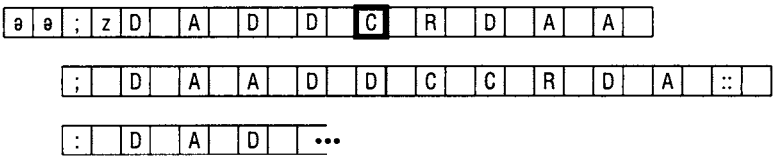
Actually, it's the *first* (leftmost) semicolon marked *z*, but the last instruction tested. The *m*-configuration əim begins by using f to find that marker and position itself at the semicolon preceding the instruction. As you'll recall, the instruction has five parts: The *m*-configuration, the scanned symbol, the symbol to print, an *L*, *N*, or *R*, and the final *m*-configuration.

[245]

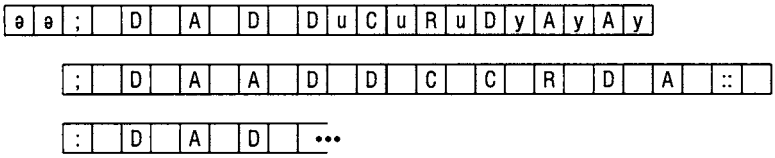
əim			f′(əim$_1$, əim$_1$, *z*)	
əim$_1$			con(əim$_2$,)	
əim$_2$	*A*			əim$_3$
	not *A*	*R*, *Pu*, *R*, *R*, *R*		əim$_2$
əim$_3$	not *A*	*L*, *Py*		c(mf, *z*)
	A	*L*, *Py*, *R*, *R*, *R*		əim$_3$

əim. **The machine marks out the instructions. That part of the instructions which refers to operations to be carried out is marked with *u*, and the final *m*-configuration with *y*. The letters *z* are erased.**

The *m*-configuration əim$_1$ refers to the con function with a blank second argument. This essentially skips past the *m*-configuration and the scanned symbol, putting the head at the second character of the print operation.



The second line for m-configuration $\mathfrak{sim}_2$ is incorrect: Emil Post suggests it should move the head *left* before printing a u. The two m-configurations $\mathfrak{sim}_2$ and $\mathfrak{sim}_3$ mark the operation (the symbol to be printed and the head-movement letter) and the next m-configuration. The $\mathfrak{e}$ function erases the z marker before heading to $\mathfrak{mf}$.



The m-configuration $\mathfrak{mf}$ (which looks like *mf* but is actually *mk* and perhaps stands for *mark*) now marks the last complete configuration. The first argument to the $\mathfrak{g}$ function (which is mistakenly $\mathfrak{q}$ in the tables of functions) should be $\mathfrak{mf}_1$ rather than $\mathfrak{mf}$.

$\mathfrak{mf}$			$\mathfrak{g}(\mathfrak{mf}, :)$
$\mathfrak{mf}_1$	not A	R, R	$\mathfrak{mf}_1$
	A	L, L, L, L	$\mathfrak{mf}_2$
$\mathfrak{mf}_2$	C	R, Px, L, L, L	$\mathfrak{mf}_2$
	:		$\mathfrak{mf}_4$
	D	R, Px, L, L, L	$\mathfrak{mf}_3$
$\mathfrak{mf}_3$	not :	R, Pv, L, L, L	$\mathfrak{mf}_3$
	:		$\mathfrak{mf}_4$
$\mathfrak{mf}_4$			$\mathrm{con}(!(!(\mathfrak{mf}_5)),)$
$\mathfrak{mf}_5$	Any	R, Pw, R	$\mathfrak{mf}_5$
	None	$P:$	$\mathfrak{sh}$

$\mathfrak{mf}$. The last complete configuration is marked out into four sections. The configuration is left unmarked. The symbol directly preceding it is marked with x. The remainder of the complete configuration is divided into two parts, of which the first is marked with v and the last with w. A colon is printed after the whole. → $\mathfrak{sh}$.

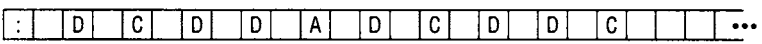
The m-configuration $\mathfrak{mf}$ uses $\mathfrak{g}$ to find the rightmost colon. That colon precedes the last complete configuration. The complete configuration is on F-squares

and, in general, consists mostly of *D*'s followed by zero or more *C*'s, each of which represents a symbol on the tape. Buried somewhere within these symbols is an *m*-configuration, which is a *D* followed by one or more *A*'s.

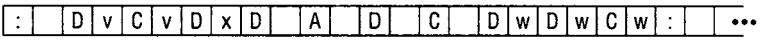
The *m*-configuration $\mathfrak{m}\mathfrak{f}_1$ looks for the *m*-configuration buried within the complete configuration. When it finds an *A*, it moves the head left to the last symbol of the square that precedes the *m*-configuration. That square is marked with *x*. Then, $\mathfrak{m}\mathfrak{f}_3$ has the job of marking all the preceding characters with *v*.

When $\mathfrak{m}\mathfrak{f}_3$ gets to the colon, $\mathfrak{m}\mathfrak{f}_4$ takes over. It uses con to skip over the *m*-configuration and the scanned character. It stops when it finds something other than a *C*. Except for the scanned character, the other symbols are marked with *w*. Finally, $\mathfrak{m}\mathfrak{f}_5$ prints a colon.

Here's a complete configuration that's a bit more complex than the simple example we've been looking at:

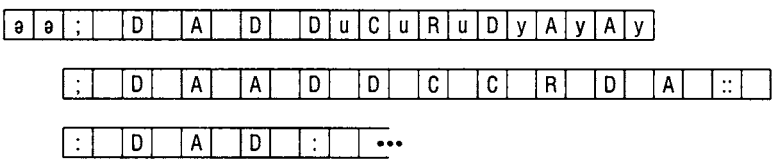


This complete configuration represents a tape starting with a 0 (*DC*) and a blank (*D*). The next square is the scanned square, indicated by the configuration q_1 (*DA*). The scanned square is a 0 (*DC*), which is followed by a blank (*D*) and a 0 (*DC*). When $\mathfrak{m}\mathfrak{f}$ is through with this, it looks like this:



The only thing unmarked is the configuration (which consists of the *m*-configuration *DA* and the scanned symbol *DC*).

In our much simpler example, there are no symbols to the left of the *m*-configuration and no symbols to the right of the scanned square, so the *v*, *x*, and *w* markers don't play a role:



Everything is now marked. The operation and final *m*-configuration of the instruction is marked with *u* and *y*, and portions of the complete configuration are marked with *v*, *x*, and *w*.

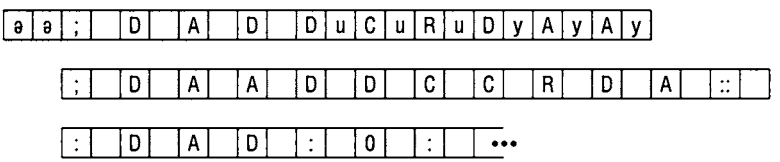
The Universal Machine needs to print a 0 or 1 if the instruction is printing a 0 or 1 except in those cases when a machine reprints a 0 and 1 because it's just scanned a 0 or 1. The Universal Machine should print a 0 or 1 only if the scanned square is blank. That's the job of $\mathfrak{o}\mathfrak{h}$ (which may stand for *show*).

$\mathfrak{sh}$			$\mathfrak{f}(\mathfrak{sh}_1, \mathfrak{inst}, u)$
$\mathfrak{sh}_1$		L, L, L	$\mathfrak{sh}_2$
$\mathfrak{sh}_2$	D	R, R, R, R	$\mathfrak{sh}_2$
	not D		$\mathfrak{inst}$
$\mathfrak{sh}_3$	C	R, R	$\mathfrak{sh}_4$
	not C		$\mathfrak{inst}$
$\mathfrak{sh}_4$	C	R, R	$\mathfrak{sh}_5$
	not C		$\mathfrak{pe}_2(\mathfrak{inst}, 0, :)$
$\mathfrak{sh}_5$	C		$\mathfrak{inst}$
	not C		$\mathfrak{pe}_2(\mathfrak{inst}, 1, :)$

$\mathfrak{sh}$. The instructions (marked u) are examined. If it is found that they involve "Print 0" or "Print 1", then 0: or 1: is printed at the end.

First, $\mathfrak{sh}$ locates the leftmost u marker, and $\mathfrak{sh}_1$ moves the head left three places to be positioned over the last symbol representing the scanned square. That symbol will be a D if the scanned square is a blank. If it's not D, then the rest of these m-configurations are skipped by heading to $\mathfrak{inst}$.

If the scanned character is a blank, then $\mathfrak{sh}_2$ goes to $\mathfrak{sh}_3$ (not $\mathfrak{sh}_2$ as the table indicates) and then $\mathfrak{sh}_3$, $\mathfrak{sh}_4$, and $\mathfrak{sh}_5$ check if the printed instruction is DC (to print 0) or DCC (print 1). If so, then $\mathfrak{pe}_2$ prints that figure and a colon at the end of the tape. The example tape now looks like this:



The $\mathfrak{sh}$ section of the table is obviously simplified by the use of binary numbers rather than decimal. Decimal numbers would require eight more m-configurations ($\mathfrak{sh}_6$ through $\mathfrak{sh}_{13}$) to print digits 2 through 9.

Whether a 0 or 1, or neither, is printed, the Universal Machine goes to $\mathfrak{inst}$ (which may stand for *instruction* but perhaps *instigate* is more descriptive). The last remaining job is to render the next complete configuration of $\mathcal{M}$. The next complete configuration includes all the symbols in the current configuration marked x, v, and w because those symbols will remain unchanged. The m-configuration and the scanned square, however, will be replaced. They will be replaced with the m-configuration marked y and the symbol marked with u.

The inst table has another reference to the g function that was defined originally as q. Also, the ce_5 function on the fifth line should be ce_5 like the third and fourth lines.

[246]

inst			g(l(inst$_1$), u)	inst. The next complete configuration is written down, carrying out the marked instructions. The letters u, v, w, x, y are erased. → anf.
inst$_1$	α	R, E	inst$_1$(α)	
inst$_1$(L)			ce_5 (ov, v, y, x, u, w)	
inst$_1$(R)			ce_5 (ov, v, x, u, y, w)	
inst$_1$(N)			ce_5 (ov, v, x, y, u, w)	
ov			e(anf)	

The function ce_5 wasn't actually defined, nor was ce_4. Basing them on ce_3 we can easily create them:

$$ce_4\ (\mathfrak{B}, \alpha, \beta, \gamma, \delta) \qquad ce(ce_3\ (\mathfrak{B}, \beta, \gamma, \delta), \alpha)$$
$$ce_5\ (\mathfrak{B}, \alpha, \beta, \gamma, \delta, \varepsilon) \qquad ce(ce_4\ (\mathfrak{B}, \beta, \gamma, \delta, \varepsilon), \alpha)$$

The ce_5 function sequentially copies symbols marked α to the end of the tape, then symbols marked β, and so forth, erasing the markers in the process.

The *m*-configuration inst refers to g, which goes to the rightmost symbol marked u; that symbol is L, R, or N. The *m*-configuration inst$_1$ scans that symbol, erases it, and then goes to inst$_1$(L), inst$_1$(R), or inst$_1$(N) depending on the symbol. It's clear what Turing wants to do here, but I really must protest the introduction of a new syntax at this point in the machine, particularly when it's not necessary. Let's replace the entire inst$_1$ configuration with the following:

$$\text{inst}_1 \begin{cases} L & R, E & ce_5\ (ov, v, y, x, u, w) \\ R & R, E & ce_5\ (ov, v, x, u, y, w) \\ N & R, E & ce_5\ (ov, v, x, y, u, w) \end{cases}$$

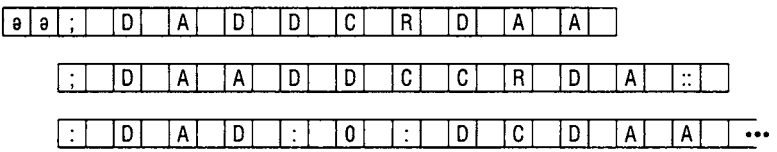
In all three cases, the squares marked v are copied to the end of the tape first, and those marked w are copied last. The symbols marked v are all those on the left part of the complete configuration up to (and not including) the square to the left of the scanned square. That square is marked x. The symbols marked w are all those to the right of the scanned square.

The three copies in the middle of ce_5 depend on whether the head is moving left, right, or not at all. The order is:

Left: Next *m*-configuration / Symbol left of head / Printed symbol.
Right: Symbol left of head / Printed symbol / Next *m*-configuration.
None: Symbol left of head / Next *m*-configuration / Printed symbol.

For example, if the head is moving left, then the next *m*-configuration is inserted before the square to the left of the previous head position. If the head is moving right, the next *m*-configuration is to the right of the printed symbol.

Each of the ce_5 functions goes to ov (which probably stands for *over*). The e function erases all *E*-squares, and goes to anf for the next move. Our tape now looks like this:



The second complete configuration contains the symbols *DC* (meaning 0) followed by *DAA*, which indicates the new *m*-configuration q_2.

The Universal Machine as Turing has defined it has a few limitations. It cannot emulate just any general Turing Machine. It won't work right with any machine that moves its head anywhere left of its initial position because it has no way of inserting blanks to the left of the complete configurations. (Indeed, the process of inserting blanks to the *right* is something that Turing omitted in the con function.) The Universal Machine also works correctly only with machines that replace blanks with 0s or 1s and do so in a uniform left-to-right manner. The Universal Machine can handle machines that perform otherwise, but it won't print the correct sequence of 0s and 1s.

Despite these limitations, and the little misprints and bugs, Turing has done something quite extraordinary. He has demonstrated the generality of computation by showing that a single universal machine can be suitably programmed to carry out the operation of any computing machine. Says one acclaimed book on computability: "Turing's theorem on the existence of a universal Turing machine [is] one of the intellectual landmarks of the last century."[4]

All of which prompts the question:

Did Alan Turing invent the computer?

[4]John P Burgess, preface, in George S Boolos, John P Burgess, and Richard C Jeffrey, *Computability and Logic*, fourth edition (Cambridge University Press, 2002), xi

10 Computers and Computability

By imagining a computing machine that does almost nothing, Turing was actually conceiving a very versatile "general purpose" computer. This was a revolutionary concept. The common assumption at the time was that computers would be designed specifically for particular types of jobs. The early analog computer known as the Differential Analyzer (designed and built by M.I.T. professor Vannevar Bush and his students in the 1920s) exemplified this approach. The Differential Analyzer did something very important — solve ordinary differential equations — but that was all it did.

Even people deeply involved in building digital computers often didn't grasp the generality of digital logic. Howard Aiken, for example, was one of the computer's true pioneers and had been working with digital computers since 1937. Yet, in 1956 Aiken said:

> [I]f it should turn out that the basic logics of a machine designed for the numerical solution of differential equations coincide with the logics of a machine intended to make bills for a department store, I would regard this as the most amazing coincidence I have ever encountered.[1]

Turing, who visualized the computer as a logic machine, knew better. While most early computer builders thought in terms of hardware, Turing had been writing software since 1936. To Turing, even basic arithmetical operations like addition could be achieved in software. Compare Aiken's 1956 statement with what Turing wrote in 1950:

> This special property of digital computers, that they can mimic any discrete state machine, is described by saying that they are

[1]Paul Ceruzzi, *Reckoners: The Prehistory of the Digital Computer, from Relays to the Stored Program Concept, 1935–1945* (Greenwood Press, 1983), 43 Ceruzzi's source is Howard Aiken, "The Future of Automatic Computing Machinery," *Elektronische Rechenanlage und Informationsverarbeitung* (Darmstadt, 1956), 33

> *universal* machines. The existence of machines with this property has the important consequence that, considerations of speed apart, it is unnecessary to design various new machines to do various computing processes. They can all be done with one digital computer, suitably programmed for each case. It will be seen that as a consequence of this all digital computers are in a sense equivalent.[2]

Turing includes the important qualification "considerations of speed apart." Some would argue that where computers are involved, speed isn't everything; it's the *only* thing. Whenever people want specialized computers — for example, to do computer-generated imagery (CGI) for a multimillion dollar Hollywood blockbuster — speed is usually a primary consideration, and beefed-up memory capacity doesn't hurt either. In actual number-crunching capabilities, however, all digital computers are universal.

Alan Turing's status in the general history of computing has never been quite clear. In one standard history[3] he barely merits mention, but when an eminent mathematician writes a history that treats the computer as a physical embodiment of mathematical concepts,[4] Turing becomes a principal player. How Turing fares in the computing history books really depends on whether the computer is approached from an engineering and commercial perspective, or from a mathematical and academic one.

One intriguing role that Turing played involves his relationship with John von Neumann. The two men first met in April 1935 when von Neumann came from Princeton to Cambridge to deliver a lecture course on almost-periodic functions. Soon after that, Turing decided he wanted to go to Princeton University himself.[5] They had contact again when Turing got to Princeton in the fall of 1936.[6] Von Neumann once claimed to have stopped reading papers in mathematical logic following the Gödel Incompleteness Theorem,[7] so it's not clear when von Neumann actually read "On Computable Numbers." The two mathematicians had other common mathematical interests (almost-periodic functions and group theory), and those are the subjects mentioned in von Neumann's letter of June 1, 1937, recommending Turing for a Procter Fellowship for his second year at Princeton.

---

[2]Alan Turing, "Computing Machinery and Intelligence," *Mind*, Vol LIX, No 236 (October 1950), 441–2.

[3]Martin Campbell-Kelly and William Aspray, *Computer: A History of the Information Machine* (Basic Books, 1996)

[4]Martin Davis, *The Universal Computer: The Road from Leibniz to Turing* (Norton, 2000)

[5]Andrew Hodges, *Alan Turing: The Enigma* (Simon & Schuster, 1983), p 95

[6]Hodges, *Alan Turing*, 118

[7]Hodges, *Alan Turing*, 124

Before Turing left Princeton in July 1938, von Neumann offered him a job at the Institute for Advanced Study as his assistant for $1,500 a year, but Turing turned down the offer. By that time, von Neumann had almost certainly read Turing's paper. For his biography of Turing, Andrew Hodges queried physicist Stanislaw Ulam (who was also at the IAS) on von Neumann's estimation of Turing. (Von Neumann himself died in 1957 at the age of 53.) Ulam recalled traveling with von Neumann in the summer of 1938, when von Neumann suggested a game of

> writing down on a piece of paper as big a number as we could, defining it by a method which indeed has something to do with some schemata of Turing's.... von Neumann had great admiration for him and mentioned his name and "brilliant ideas" to me already, I believe, in early 1939.... At any rate von Neumann mentioned to me Turing's name several times in 1939 in conversations, concerning mechanical ways to develop formal mathematical systems.[8]

These early points of contact between Turing and von Neumann become suddenly important in September 1944 when von Neumann arrived at the Moore School of Electrical Engineering of the University of Pennsylvania. Already under construction was a computer called the ENIAC (Electronic Numerical Integrator and Computer), a 30-ton behemoth designed under the supervision of John Presper Eckert (1919–1995) and John William Mauchly (1907–1980). Even as it was being constructed, the limitations of the ENIAC had become apparent and a successor was planned, to be called the EDVAC (Electronic Discrete Variable Automatic Computer).

> From the beginning, von Neumann's perspective was not simply that of a potential user, but of a scientific and technical contributor as well. In the remaining months of 1944 and throughout 1945, when he was not at Los Alamos, he took time to attend technical conferences on the EDVAC and to make technical contributions and suggestions on logic design.[9]

Yet, when a document appeared dated June 30, 1945, entitled "First Draft of a Report on the EDVAC"[10] with John von Neumann as the sole author, a controversy was ignited, and the smoke has yet to clear. The report emphasizes important

---

[8]Hodges, *Alan Turing*, 145

[9]Nancy Stern, "John von Neumann's Influence on Electronic Digital Computing, 1944–1946," *Annals of the History of Computing*, Vol 2 No 4 (October 1980), 353

[10]Reprinted in Brian Randell, ed , *The Origins of Digital Computers* (Springer, 1973)

concepts — that the computer should be electronic, that it should work with binary numbers, and that programs should be stored in memory — but it's never been fully determined whether von Neumann originated these concepts or if he simply articulated ideas that that had been floating around the Moore School since the ENIAC days. For decades, people have referred to "von Neumann architecture" when describing digital computers, but this term is slipping out of use, partially out of respect for those many others who contributed to concepts of computer architecture.

The "First Draft of a Report on the EDVAC" makes reference to just one other publication: a paper entitled "A Logical Calculus of the Ideas Immanent in Nervous Activity" published in the *Bulletin of Mathematical Biophysics*.[11] This reference reveals von Neumann's interest in the relationship between the computer and the human brain, but it's also interesting that the authors of this paper had based their concepts of the physiology of the brain on the functions of Turing Machines. The McCulloch and Pitts paper is also cited by Norbert Wiener (1894–1964) in his classic book *Cybernetics, or Control and Communication in the Animal and the Machine* (1948). I'll have more to say about McCulloch, Pitts, Wiener, and von Neumann in Chapter 17.

The physicist Stanley Frankel, who worked with von Neumann at Los Alamos, remembers von Neumann's enthusiasm about Turing's paper in 1943 or 1944:

> Von Neumann introduced me to that paper and at his urging I studied it with care. Many people have acclaimed von Neumann as the 'father of the computer' (in a modern sense of the term) but I am sure that he would never have made that mistake himself. He might well be called the midwife, perhaps, but he firmly emphasized to me, and to others I am sure, that the fundamental conception is owing to Turing — insofar as not anticipated by Babbage, Lovelace, and others. In my view von Neumann's essential role was in making the world aware of these fundamental concepts introduced by Turing and of the development work carried out in the Moore school and elsewhere.[12]

Throughout the latter 1940s, von Neumann seems to have mentioned the importance of Turing to several people. For example, in 1946, he wrote to Norbert

[11] W S MacCulloch and W Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics*, Vol 5 (1943), 115–133

[12] Letter quoted in B. Jack Copeland, ed , *The Essential Turing: The Ideas that Gave Birth to the Computer Age* (Oxford University Press, 2004), 22 This letter is part of a 6-page section on "Turing, von Neumann, and the Computer" in Copeland's guide to the "Computable Numbers" paper

Wiener of "the great positive contribution of Turing ... one, definite mechanism can be 'universal.'"[13]

Although Alan Turing is remembered mostly for his writings, his name is also linked to three major computer projects.

The first was the Colossus, a code-breaking computer developed and built at Bletchley Park in 1943. It was designed by Max Newman, the mathematician who had taught the Foundation of Mathematics class that inspired Turing to write "On Computable Numbers" and had guided the paper to publication, and who had been at Bletchley Park since the summer of 1942. Although some writers have assumed that Turing was involved in the Colossus project,[14] it appears he was not. He knew about it, of course, but he "declined the invitation to take a direct part."[15] Nevertheless, the influence of Turing's paper on the logical design of the Colossus was clearly acknowledged.[16]

Turing was much more involved in a computer project at the National Physical Laboratory (NPL) at Teddington in southwest London. In 1944 the director of NPL was Sir Charles Darwin (1887–1962), whose grandfather had published some influential books on biology. Darwin created a Mathematics Division which was given the job of developing automated computing machines.

J. R. Womersley, the head of the Mathematics Division, summoned Turing to NPL for an interview in June 1945.[17] Womersley had read "On Computable Numbers" and wanted Turing to design a computer called the Automatic Computing Engine, or ACE, and if the word "engine" evoked memories of Charles Babbage, that was deliberate.

Turing, having read von Neumann's EDVAC Report and having a few ideas about computers of his own, finished the "Proposal for Development in the Mathematics Division of an Automatic Computing Engine (or ACE)" before the end of 1945. Turing's report says that it "gives a fairly complete account of the proposed calculator" but recommends that it "be read in conjunction with J. von Neumann's 'Report on the EDVAC'."[18]

Turing's proposed machine was electronic, used binary numbers, and had a 1 megahertz clock rate, although bits were transferred serially. It used mercury delay line storage, which stored bits as acoustic pulses in tubes of mercury.

---

[13]B. Jack Copeland and Diane Proudfoot, "Turing and the Computer" in B. Jack Copeland, ed , *Alan Turing's Automatic Computing Engine The Master Codebreaker's Struggle to Build the Modern Computer* (Oxford University Press, 2005), 116

[14]Myself included in Charles Petzold, *Code· The Hidden Language of Computer Hardware and Software* (Microsoft Press, 1999), 244

[15]Hodges, *Alan Turing*, 268

[16]Hodges, *Alan Turing*, 554 (note 5 7)

[17]Introduction to B.E Carpenter and R W Doran, *A.M Turing's ACE Report of 1946 and Other Papers* (MIT Press, 1986), 5–6

[18]*ACE Report*, 21.

A five-foot tube of mercury could store 1,024 bits. Each bit required about a millisecond to travel from one end of the tube to the other, whereupon it could be accessed and recycled to the beginning of the tube. Turing expected an addition of two 32-bit numbers to require 32 microseconds (that's one bit per clock cycle) and a 32-bit multiplication to require "rather over two milliseconds."[19]

Turing's design has a fairly small number of primitive instructions, mostly transfers between memory and registers. In this sense it resembles modern Reduced Instruction Set Computers (RISC), which incorporate fast hardware and do more complex jobs in software. Turing seems to have invented the *stack* — eventually a common form of computer storage analogous to the stack of plates in a cafeteria held aloft in a well by a spring. The last plate "pushed" on the stack becomes the next plate "popped" from the stack. Turing's routines for these two operations are called BURY and UNBURY.[20]

Turing presented a more personal vision of computing in a lecture to the London Mathematical Society on February 20, 1947. "[C]omputing machines such as the ACE... are in fact practical versions of the universal machine." The complexity of the job the machine must do "is concentrated on the tape" — that is, in software — "and does not appear in the universal machine proper in any way."[21] Turing recognized the importance of speed in the computer, of course, but he tended to emphasize the advantages of large storage:

> I believe that the provision of proper storage is the key to the problem of the digital computer, and certainly if they are to be persuaded to show any sort of genuine intelligence much larger capacities than are yet available must be provided. In my opinion this problem of making a large memory available at reasonably short notice is much more important than that of doing operations such as multiplication at high speed.[22]

As might be expected, Turing clearly recognized the advantages of binary numbers over decimal:

> Binary working is the most natural thing to do with any large scale computer. It is much easier to work in the scale of two than any other, because it is so easy to produce mechanisms which have two positions of stability.[23]

---

[19] *ACE Report*, 116
[20] *ACE Report*, 76
[21] *ACE Report*, 112–113
[22] *ACE Report*, 112
[23] *ACE Report*, 113–114

Towards the end of the talk, Turing speculated about machines that can modify their own instruction tables:

> It would be like a pupil who had learnt much from his master, but had added much more by his own work. When this happens I feel that one is obliged to regard the machine as showing intelligence.[24]

By September 1947, Turing was feeling frustrated about the lack of progress being made on the ACE. He requested a year's sabbatical at half pay and departed to Cambridge. The expectation at NPL was that he would return for at least another two years, but that never happened. (The Pilot ACE wasn't ready until 1950, and it deviated quite a bit from Turing's original proposal.)

Instead, Turing went to join Max Newman, who had been at the University of Manchester since 1945. Newman had obtained a grant for a new Computing Machine Laboratory and was building a computer called the Mark I. In June 1948, the Mark I became "the first EDVAC-type electronic stored-program computer to be completed."[25]

Turing joined the Manchester mathematics faculty and Newman's project in September. Two months later, an arrangement was reached with Ferranti Limited, a Manchester manufacturer of electronics, to develop a machine that Ferranti would market commercially.

Turing was mostly responsible for the programming aspects of the Mark I. Around 1951, Turing was given the job of writing the first "Programmers' Handbook" for the production machine, in which Turing defined programming as "an activity by which a digital computer is made to do a man's will, by expressing this will suitably on punched tapes."[26]

Rather than mercury delay lines, the Mark I used cathode ray tubes for storage. This type of storage — often called the Williams Tube — was pioneered by F. C. Williams, who had come to Manchester in December 1946. The data to be stored is sent to the CRT as electrical pulses, where it is displayed on the screen as an array of dots, each representing one bit. A different intensity or size of the dots distinguishes between 0 and 1. A metal plate in front of the tube picks up the charges from these dots and allows the tube to be refreshed or read. A second CRT would allow people to view the dots and examine the data. By 1947, each CRT was capable of storing 2,048 bits.

---

[24] *ACE Report*, 123

[25] Martin Campbell-Kelly, "Programming the Mark I Early Programming Activity at the University of Manchester," *Annals of the History of Computing*, Vol 2, No 2 (April 1980), 134

[26] Campbell-Kelly, "Programming the Mark I", 147 The handbook is available at www alanturing net/turing_archive/archive/index/manchesterindex html

The Mark I stored data in 40-bit words, which could also store two 20-bit instructions. These words were displayed on the CRTs in 5-bit groups, and so a base-32 notation developed where each 5-bit code was represented by the teleprinter character corresponding to that code. To read a number, it was necessary to know the code corresponding to each character. These character codes were not in alphabetical order, so following Turing's gallant lead, everybody who programmed for the Mark I was forced to memorize the 32-character sequence:

/E@A:SIU½DRJNFCKTZLWHYPQOBG"MXV£

Turing's involvement in these actual computer projects may cause us to lose sight of the simple fact that Turing's intent in his "Computable Numbers" paper was *not* to design a universal computing machine. The whole purpose of the paper was to use this hypothetical computer to help resolve the Entscheidungsproblem. There are still a few more steps. A crucial one is to demonstrate that Turing's machines are intrinsically limited in what they can do.

Turing stated in the introduction to his paper, "Although the class of computable numbers is so great, and in many ways similar to the class of real numbers, it is nevertheless enumerable" (his page 230; my page 66). In Section 5, he demonstrated how "To each computable sequence there corresponds at least one description number, while to no description number does there correspond more than one computable sequence. The computable sequences and numbers are therefore enumerable" (his page 241; my page 138).

Some doubts may still linger. It's obvious that the class of computable numbers contains at least some transcendentals. Turing Machines that calculate π or e or Liouville's constant are certainly possible. Surely transcendentals whose digits have some kind of order are computable by a Turing Machine. That's the game Ulam and von Neumann played while traveling together.

Nevertheless, the vast majority — no, no, no, let's be realistic about this and say *virtually all* — virtually all transcendentals are ostensibly streams of random digits. In the realm of real numbers, orderly or calculable sequences of digits are rare. Complete and total randomness is the rule.

How exactly do you make a machine that computes a number with no pattern? Do you just generate digits randomly?

Randomness is not something computers do very well, and yet computers are often called upon to behave randomly. Some statistics applications require random numbers, and computer games routinely need random numbers to vary the action. Without random numbers each hand of Solitaire would be exactly the same.

Programming languages often provide some standard way for programs to generate random numbers. For example, a computer program written in the C programming language can use a function named *rand* to obtain a random number between 0 and 32,767. The *rand* function begins with a number known as a *seed*, which by default is initially set to 1. Different *rand* functions may implement the

actual algorithm in different ways; as an example, here's the implementation of *rand* found in Microsoft's version of C[27]:

```
int seed = 1;

int rand()
{
   return ((seed = seed * 214013 + 2531011) >> 16) & 32767;
}
```

The *rand* function multiplies *seed* by 214,013 and adds 2,531,011, and then stores that result back in *seed* for the next time *rand* is called; however, since *seed* is defined as a 32-bit signed integer, overflow or underflow may result. The result of the calculation is truncated to 32 bits, and if the highest bit is 1, the value is actually negative.[28] The calculation continues by shifting the result 16 bits, effectively dividing it by 65,536 and truncating any fractions. Finally, a Boolean AND operation is performed between the bits of that result and the bits of 32,767. That eliminates all bits except the bottom 15 and ensures that the result is between 0 and 32,767.

Even if you didn't follow this convoluted calculation, it should be obvious that in no way is this *rand* function generating random numbers! The function is entirely deterministic. Starting with a *seed* value of 1, repeated calls to the function always result in the calculation of the same series of numbers:

41
18,467
6,334
26,500
19,169
...

The first time a program calls *rand*, the function returns 41, and the 30,546th time a program calls *rand*, the function also returns 41, and then the cycle repeats.

Because this sequence is entirely determined by the seed and the algorithm, it is not truly random. It is instead called a *pseudo-random sequence*. If you performed certain statistical tests on the numbers generated by *rand*, they would appear to exhibit characteristics of randomness. In some applications a pseudo-random sequence is preferred to truly random numbers because it's possible to reproduce results and test that the program is working correctly.

---

[27] rand c © 1985–1997, Microsoft Corporation Some details in the *rand* function have been altered for purposes of clarity

[28] A discussion of overflow and underflow may be found in my book *Code*, 153–154

In games, however, generating the same sequence of random numbers is *not* desirable. For that reason, every time you deal a new hand of Solitaire, the program probably begins by obtaining the current time of the day down to seconds and milliseconds, and then uses that to set a new seed. Assuming you don't deal at the exact — down to the millisecond — same time of the day, you're getting what appears to be a random hand.

John von Neumann once said that "Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."[29] (This was right before he described arithmetical methods for producing random numbers.) Because computers generate random numbers incompetently, applications that *really* need random numbers go outside the computer and use dedicated hardware for this task. A hardware random number generator (RNG) might use ambient noise or quantum processes to generate random numbers.

Curiously enough, Alan Turing seems to have originated the idea of generating random numbers in hardware. Turing requested that the production model of the Mark I at the University of Manchester have a special instruction that generated a random number from a noise source. It turned out to be not quite as random as it should have been, but random enough to prevent it from being properly debugged.[30]

Let's assume we have a hardware RNG that works. We put it to use to generate a sequence of 0s and 1s just like a Turing Machine. Put a binary point in front and you can watch a real number — doubtlessly a transcendental number — being created right before your eyes. (If you tried this with a software pseudo-random sequence, eventually the seed would be recalculated and the sequence would begin again. The resultant number would have a repeating sequence of digits and that means it would be rational.)

Now define a Turing Machine that generates the very same real number as the hardware RNG. You can't do it. The only approach that duplicates the RNG is a Turing Machine that explicitly prints exactly the digits you need, but that's not a Turing Machine with a finite number of configurations.

Maybe the randomness of most real numbers is merely an illusion. After all, the digits of π appear to be random, but π is definitely computable. Maybe real numbers that appear to be random actually have some kind of underlying structure that we just don't know about. Maybe if we approach this question from a different direction, we might instead prove that the computable numbers are *not* enumerable. We might then be able to get a good night's sleep because we'd be comforted with the knowledge that every real number is computable.

---

[29] John von Neumann, *Collected Works, Volume V, Design of Computer, Theory of Automata, and Numerical Analysis* (Macmillan, 1963), 768 The statement was originally made at a symposium on Monte Carlo methods in 1949

[30] Martin Campbell-Kelly, "Programming the Mark I", 136

Turing needs to confront these possibilities head on.

> 8. *Application of the diagonal process.*
>
> It may be thought that arguments which prove that the real numbers are not enumerable would also prove that the computable numbers and sequences cannot be enumerable*. It might, for instance, be thought that the limit of a sequence of computable numbers must be computable.
>
> * *Cf.* Hobson, *Theory of functions of a real variable* (2nd ed., 1921), 87, 88.

Turing is alluding to Georg Cantor's first (1874) proof of the nonenumerability of the real numbers that I described beginning on page 24. It's likely that Turing didn't have access to Cantor's original publication so he refers instead to a text book by E. W. Hobson and published by Cambridge University Press.[31] Hobson follows Cantor very closely, even using much of the same notation.

Turing suggests that Cantor's exercise be repeated using an enumeration of computable numbers rather than real numbers. In both cases, the numbers approach a limit. In Cantor's proof, that limit has to be a real number (What else could it be?), but Cantor was able to demonstrate that the limit wasn't in the enumeration of real numbers, thus proving that the real numbers are not enumerable.

When the same process is attempted with computable numbers, the computable numbers also approach a limit. Could that limit also be a computable number? Turing's answer:

> This is clearly only true if the sequence of computable numbers is defined by some rule.

By "sequence" Turing means the sequence of alphas and betas that approach the limit. That limit is a computable number only if we can compute it — that is, we

[31] The full title of this influential book by Cambridge mathematics professor Ernest William Hobson (1856–1933) is *The Theory of Functions of a Real Variable and the Theory of Fourier's Series*, and the first edition was published by Cambridge University Press in 1907 The second edition that Turing refers to dates from 1921, but so much new material had been added that a second *volume* had to be published in 1926 This Volume II was also referred to as a second edition Volume I was revised as a third edition in 1927 The third edition of Volume I and the second edition of Volume II were republished by Harren Press in 1950 and Dover Books in 1957, and these might be the easiest editions to track down The discussion that Turing refers to is on pages 84 and 85 of this third edition of Volume I It is followed on pages 85 and 86 by Cantor's diagonal proof

can devise some algorithm that tells us the numeric limit approached by these alphas and betas. That does not seem likely. If we don't have a way to compute this limit, it is not a computable number. It is yet another uncomputable real number, and hence we haven't disproved that computable numbers are enumerable.

> Or we might apply the diagonal process.

Turing puts the rest of the paragraph in quotation marks as if an intruder has burst into his paper trying to convince us that the computable numbers are not enumerable. Turing's adversary pursues a more notation-laden arithmetic variation of the diagonal process than the one I offered on page 28.

> "If the computable sequences are enumerable, let α_n be the n-th computable sequence, and let $\phi_n(m)$ be the m-th figure in α_n.

It's just notation. Each computable sequence is a series of 0s and 1s, and each of these binary digits is represented by ϕ, the Greek letter phi. The computable sequences can be listed with a superfluity of subscripts and indices like so:

$$\alpha_1 = \phi_1(1)\ \phi_1(2)\ \phi_1(3)\ \phi_1(4)\ \ldots$$
$$\alpha_2 = \phi_2(1)\ \phi_2(2)\ \phi_2(3)\ \phi_2(4)\ \ldots$$
$$\alpha_3 = \phi_3(1)\ \phi_3(2)\ \phi_3(3)\ \phi_3(4)\ \ldots$$
$$\ldots$$

> Let β be the sequence with $1 - \phi_n(n)$ as its n-th figure.

In other words, β is the diagonal with the 0s and 1s flipped:

$$\beta = (1 - \phi_1(1))\ (1 - \phi_2(2))\ (1 - \phi_3(3))\ (1 - \phi_4(4))\ldots$$

> Since β is computable, there exists a number K such that $1 - \phi_n(n) = \phi_K(n)$ all n.

That is, for some K, there's an α_K in the enumerated list of computable numbers:

$$\beta = \alpha_K = \phi_K(1)\ \phi_K(2)\ \phi_K(3)\ \phi_K(4)\ldots$$

In general, for digit n,

$$1 - \phi_n(n) = \phi_K(n)$$

or

$$1 = \phi_K(n) + \phi_n(n)$$

Turing's adversary now uses this arithmetic argument to demonstrate that β can't exist:

> Putting $n = K$,

that is,

$$1 = \phi_K(K) + \phi_K(K)$$

> we have $1 = 2\phi_K(K)$, *i.e.* 1 is even. This is impossible. The computable sequences are therefore not enumerable".

Well, that's interesting. This mysterious intruder has just described how to compute a number called β based on the computable sequences in the enumerated list, but this computed number is not in the list. Therefore, the intruder says, the computable sequences are not enumerable.

But Turing remains calm, and says:

> The fallacy in this argument lies in the assumption that β is computable.

Fallacy? What fallacy? How can β not be computable? β is computed from the enumeration of computable sequences, so it has to be computable, right?

Well, not exactly.

Let's step back a moment. Turing originally defined computable numbers as those that are calculable by finite means. He constructed imaginary machines

to compute these numbers, and he showed that each machine can be uniquely identified by a positive integer called a Description Number. Because integers are enumerable, Turing Machines are also enumerable, and therefore computable sequences are enumerable.

In one sense, enumerating the Turing Machines is as easy as enumerating the positive integers:

1
2
3
4
5
...

All the Turing Machines will appear in this list in the form of Description Numbers, and from the Description Number we can get the Standard Description, and then we can feed that to the Universal Machine to get the computable sequence.

Of course, we're missing something: We're missing a way to determine exactly which positive integers in that list are Description Numbers of circle-free machines.

As you may recall from the definitions in Section 2, a circle-free machine is one that goes on printing 0s and 1s forever. Although a machine that never stops may appear to be "out of control" or "gone crazy," circle-free machines are necessary to compute irrational numbers and those rational numbers with repeating digits. Even when printing rational numbers like .1 (the binary equivalent of $\frac{1}{2}$), it is preferable for the machine to be circle-free by printing 1 and then a continuous sequence of 0s:

.10000000 ...

A *circular* machine, on the other hand, is one that gets stuck in an undesirable loop. A circular machine could keep printing 0s without advancing the head, for example, or it could forever print symbols other than 0 and 1.

The terms *circle-free* and *circular* are not optimally descriptive: A circle-free machine might spend the rest of eternity in a little loop that prints 0s or 1s, and that might be fine. A circular machine could get jammed because it's directed to an *m*-configuration that doesn't exist, and that's just one of many problems that could befall it.

We need to identify the Description Numbers of circle-free machines because those are the only ones qualified to be interpreted by the Universal Machine. We may have successfully enumerated all the Turing Machines (somewhere within that list of positive integers), but we haven't identified those that are circle-free, so we can't use them to generate computable sequences.

It's very clear that many integers are not Description Numbers of any machine whatsoever. We can easily determine (by human inspection or a Turing Machine) whether a particular integer is a *well-formed* Description Number, which means that it's divided into well-formed instructions, each of which begins with an *m*-configuration, and so forth. We might even determine whether the machine refers to *m*-configurations that aren't present. We could check whether certain *m*-configurations aren't used. We could also easily check to see whether any *m*-configurations include instructions that actually print 0s or 1s. Such a process would determine that the lowest well-formed Description Number is 31,334,317, and this is a circular machine. (It only prints blanks.) It's not until 313,324,317 that the first circle-free machine is encountered, and not until 313,325,317 that we find the first circle-free machine that prints from left to right.

Here's the very beginning of an enumeration of the positive integers where the first two circle-free print-to-the-right Turing Machines are identified:

1
2
3
4
5
...
313,325,317 ← This one prints 0's to the right
...
3,133,225,317 ← This one prints 1s to the right
...

These, of course, are the simplest of simple machines, and the method to identify them is simple as well. Much more difficult — actually, as Turing will show, impossible — is a machine that implements a *general process* to determine whether a particular integer is the Description Number of a circle-free machine.

That general process is precisely what we need to perform the diagonalization. Each digit of β is based on a different computable number, so computing β requires that all the circle-free Turing Machines be identified. Turing will show that these circle-free machines cannot be identified by finite means, which means that we can't explicitly enumerate the computable sequences. It is therefore simply not true that β is a computable sequence.

It would be true if we could enumerate the computable sequences by finite means, but the problem of enumerating computable sequences is equivalent to the problem of finding out whether a given number is the D.N of a circle-free machine, and we have no general process for doing this in a finite number of steps.

Turing now makes a subtle shift in focus. He started by attempting to apply Cantor's diagonalization proof to computable sequences, but now he wants simply to explore what happens when we try to identify all the Description Numbers of circle-free machines.

> In fact, by applying the diagonal process argument correctly, we can show that there cannot be any such general process.

If, as Turing asserts, there's no general process for determining whether a particular integer is a Description Number of a circle-free machine, then β is not computable. That would invalidate the interloper's "proof" that the computable sequences are not enumerable. Nothing would then detract from our confidence that computable sequences are indeed enumerable and hence can't include all the real numbers.

Unfortunately, Turing begins the next paragraph rather vaguely:

> The simplest and most direct proof of this is by showing that, if this general process exists, then there is a machine which computes β.

I think he's saying that there cannot be a general process to determine whether a particular machine is circle-free because, if there were, we'd be able to compute β, and we know we can't compute β, because then the diagonal argument would be valid, and computable sequences would not be enumerable.

> This proof, although perfectly sound, has the disadvantage that it may leave the reader with a feeling that "there must be something wrong".

The paradox still nags at our consciences. For that reason, Turing will now prove more directly that there is no machine that will determine whether a particular integer is a Description Number of a circle-free machine.

> The proof which I shall give has not this disadvantage, and gives a certain insight into the significance of the idea "circle-free".

He might also have added that the implications go far beyond this little exercise in number theory.

All Turing wants now is a machine that extracts one digit from each computable sequence. He doesn't have to bother with subtracting the digits from one. He's actually going to try to compute something a little bit simpler than β:

> It depends not on constructing β, but on constructing β', whose n-th figure is $\phi_n(n)$.

On the very first page of Turing's paper (page 66 of this book) he said, "The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable." Both β and β' are such definable numbers. β' is definable because instructions can be given for how to compute it: Enumerate the whole numbers starting at 1. For each number, determine whether it's a well-formed Description Number of a Turing Machine. If so, determine whether that machine is circle-free. If so, compute that number up to the n-th digit (where n is one more than the number of circle-free machines encountered so far). That digit is the n-th digit of β'.

You can see that β' is completely defined, but can it be computed?

Although Turing defined no instruction that would ever halt the machine, the problem that Turing is now attacking is studied more in the variation known as the *Halting Problem*. (The term originated in Martin Davis's 1958 book *Computability and Unsolvability*.[32]) Can we define a Turing Machine that will determine whether another Turing Machine will either halt or go on forever? If we substitute the idea of circularity for halting, it's a similar problem. Can one Turing Machine analyze another Turing Machine and determine its ultimate fate?

Turing begins by assuming there exists a machine that determines whether any arbitrary machine is circle-free. In the following discussion, he refers to the machine's Standard Description rather than the Description Number, but it doesn't really matter because it's trivial to convert between them.

> [247]
>
> Let us suppose that there is such a process; that is to say, that we can invent a machine $\mathcal{D}$ which, when supplied with the S.D of any computing machine $\mathcal{M}$ will test this S.D and if $\mathcal{M}$ is circular will mark the S.D with the symbol "*u*" and if it is circle-free will mark it with "*s*".

[32] Martin Davis, *Computability and Unsolvability* (McGraw-Hill, 1958), 70. Davis believes he first used the term in lectures in 1952 (See Copeland, *The Essential Turing*, 40, footnote 61) The concept also shows up in Chapter 13 of Stephen Cole Kleene, *Introduction to Metamathematics* (Van Nostrand, 1952)

The machine $\mathcal{D}$ is the Decision machine. The "*u*" stands for *unsatisfactory* (meaning a circular machine) and the "*s*" for *satisfactory* (circle-free). Turing defined these terms at the end of Section 5 (his page 241, my page 142).

> By combining the machines $\mathcal{D}$ and $\mathcal{U}$ we could construct a machine $\mathcal{H}$ to compute the sequence β'.

Actually the $\mathcal{H}$ machine also needs to generate positive integers and then convert them to Standard Descriptions, but that's fairly trivial. For every positive integer that $\mathcal{H}$ generates, $\mathcal{H}$ uses $\mathcal{D}$ to determine whether the number defines a satisfactory machine. If so, then $\mathcal{H}$ passes that Standard Description to the Universal Machine $\mathcal{U}$ to compute the sequence. For the *n*-th computable sequence, $\mathcal{U}$ needs only to run the machine up to the *n*th digit. That digit then becomes the *n*th digit of β'. Because $\mathcal{U}$ is under the control of $\mathcal{H}$, $\mathcal{H}$ can stop $\mathcal{U}$ when it has the particular digit it needs.

It's necessary for $\mathcal{H}$ to check the Standard Description with $\mathcal{D}$ first because we don't want $\mathcal{U}$ to get stuck running an unsatisfactory machine. If $\mathcal{H}$ gives $\mathcal{U}$ the Standard Description of an unsatisfactory machine, and that unsatisfactory machine never prints a digit, then the process gets stuck and can't move forward.

Turing does not actually show us what this magic Decision machine $\mathcal{D}$ looks like, so that should be a big hint that such a machine is impossible. Just off hand, it seems like it would at least be very difficult. How can $\mathcal{D}$ determine that a particular machine is circle-free except by mimicking the machine and tracing through its every step?

At any rate, $\mathcal{D}$ is similar to $\mathcal{U}$ in that it works with a Standard Description (or, equivalently, a Description Number) encoded on a tape.

> The machine $\mathcal{D}$ may require a tape. We may suppose that it uses the E-squares beyond all symbols on F-squares, and that when it has reached its verdict all the rough work done by $\mathcal{D}$ is erased.

It leaves behind only an *s* or *u* for its final verdict.

> The machine $\mathcal{H}$ has its motion divided into sections. In the first $N-1$ sections, among other things, the integers $1, 2, \ldots, N-1$ have been written down and tested by the machine $\mathcal{D}$.

The term "divided into sections" does not mean that there exist different parts of $\mathcal{H}$ that handle the different numbers. Separate sets of configurations for each

number would require that $\mathcal{H}$ be infinite. Turing is really referring to sequential operations over a period of time. The actual process must be a general one that applies to all integers: The $\mathcal{H}$ machine generates positive integers one after another, passes each in turn to $\mathcal{D}$ to determine whether it's satisfactory, and, if so, uses $\mathcal{U}$ to calculate a certain number of digits in the computable sequence.

> A certain number, say $R(N-1)$, of them have been found to be the D.N's of circle-free machines.

R just accumulates a count of circle-free machines that have already been encountered. The machine needs R to determine how many digits to calculate for each circle-free machine that turns up.

> In the N-th section the machine $\mathcal{U}$ tests the number N. If N is satisfactory, *i.e.*, if it is the D.N of a circle-free machine, then $R(N) = 1 + R(N-1)$ and the first $R(N)$ figures of the sequence of which a D.N is N are calculated.

If N is 3,133,225,317, for example, then $R(N-1)$ is 1. (See the list above of positive integers with the first two satisfactory machines identified.) Only one satisfactory machine has been discovered so far. The machine $\mathcal{D}$ will determine that N is indeed the Description Number of a circle-free machine. So, $R(N)$ is set to 2, and $\mathcal{U}$ calculates the first two digits of the machine defined by 3,133,225,317. Those two digits will both be 1. $\mathcal{H}$ uses the second of those digits as the second digit of β'. It's on its way!

> The $R(N)$-th figure of this sequence is written down as one of the figures of the sequence β' computed by $\mathcal{H}$.

The usual case, of course, is that the Description Number is either no machine at all or a circular machine.

> If N is not satisfactory, then $R(N) = R(N-1)$ and the machine goes on to the $(N+1)$-th section of its motion.

The point is that $\mathcal{H}$ must look at the potential Description Numbers one after another, and for each satisfactory Description Number, $\mathcal{H}$ must run the machine until the $R(N)$-th digit.

Turing now takes great pains to demonstrate that $\mathcal{H}$ is circle-free. $\mathcal{H}$ simply runs $\mathcal{D}$ for each potential Description Number and $\mathcal{D}$ is circle-free by the original assumptions.

> From the construction of $\mathcal{H}$ we can see that $\mathcal{H}$ is circle-free. Each section of the motion of $\mathcal{H}$ comes to an end after a finite number of steps. For, by our assumption about $\mathcal{D}$, the decision as to whether N is satisfactory is reached in a finite number of steps. If N is not satisfactory, then the N-th section is finished. If N is satisfactory, this means that the machine $\mathcal{M}(N)$ whose D.N is N is circle-free, and therefore its $R(N)$-th figure can be calculated in a finite number of steps. When this figure has been calculated and written down as the $R(N)$-th figure of β', the N-th section is finished. Hence $\mathcal{H}$ is circle-free.

$\mathcal{H}$ is a Turing Machine, so $\mathcal{H}$ has a Description Number (which Turing calls K). At some point, $\mathcal{H}$ will have to deal with its own Description Number. $\mathcal{H}$ will have to determine whether $\mathcal{H}$ is circle-free.

> Now let K be the D.N of $\mathcal{H}$. What does $\mathcal{H}$ do in the K-th section of its motion? It must test whether K is satisfactory, giving a verdict "s" or "u". Since K is the D.N of $\mathcal{H}$ and since $\mathcal{H}$ is circle-free, the verdict cannot be "u".

Then Turing also adds:

> On the other hand the verdict cannot be "s".

The fundamental problem is that $\mathcal{H}$ gets into an infinite recursion. Before $\mathcal{H}$ encounters the number K (the Description Number of itself) $\mathcal{H}$ has analyzed all positive integers 1 through $K - 1$. The number of circle-free machines so far is $R(K - 1)$ and the first $R(K - 1)$ digits of β' have been found.

What is the $R(K)$-th digit of β'? To get that digit, $\mathcal{H}$ has to trace through its own operation, which means it has to duplicate everything up to the point where it encountered K, and then the process begins again. That is why $\mathcal{H}$ cannot be circle-free.

> For if it were, then in the K-th section of its motion $\mathcal{H}$ would be bound to compute the first $R(K - 1) + 1 = R(K)$ figures of the sequence computed by the machine with K as its D.N and to write down the $R(K)$-th as a figure of the

> sequence computed by $\mathcal{H}$. The computation of the first $R(K) - 1$ figures would be carried out all right, but the instructions for calculating the $R(K)$-th would amount to "calculate the first $R(K)$ figures computed by H and write down the $R(K)$-th". This $R(K)$-th figure would never be found.

(In the penultimate line, H should be $\mathcal{H}$.)

$\mathcal{H}$ is generating a sequence of digits based on the sequences generated by other machines. That's pretty straightforward when we think of machines as generating sequences like the binary equivalent of 1/3, π, and the square root of 2, but where does $\mathcal{H}$ get digit K of this sequence that it's generating? It has to get that digit from itself, but that makes no sense because $\mathcal{H}$ gets digits only from other machines.

OK, so $\mathcal{H}$ has a little problem when encountering its own Description Number. Can't it just skip that one? Well, yes, it can, but as we've seen, every computable sequence can be calculated by a variety of different machines. Machines could calculate the same sequence in different ways, or they could have superfluous instructions. $\mathcal{H}$ would need to skip those similar machines as well. What about the machines that don't calculate β', but calculate something close to β', such as β' with its 27^{th} and 54^{th} digits swapped? There are an infinite number of such machines and avoiding them all puts quite a burden on $\mathcal{H}$ — an *impossible* burden.

> *I.e.*, $\mathcal{H}$ is circular, contrary both to what we have found in the last paragraph and to the verdict "*s*". Thus both verdicts are impossible and we conclude that there can be no machine $\mathcal{D}$.

There can be no general process to determine whether a machine is circle-free. By implication, there can be no computer program that will determine the ultimate fate of other computer programs.

Turing has also resolved the paradox of the diagonal process: He first established that computable numbers are enumerable, yet the diagonal process seemed to indicate that you could create a computable number not in the list. Turing has shown that the diagonal could *not* be calculated by finite means, and hence is not computable. Computable numbers may be enumerable, but they cannot actually be enumerated in a finite number of steps.

Turing is not quite finished with this section. He now hypothesizes a machine $\mathcal{E}$, which might stand for "ever print."

> [248]
>
> We can show further that *there can be no machine $\mathcal{E}$ which, when supplied with the S.D of an arbitrary machine $\mathcal{M}$, will determine whether $\mathcal{M}$ ever prints a given symbol* (0 *say*).

Turing needs this $\mathcal{E}$ machine in the final section of the paper when he uses it to prove that the Entscheidungsproblem has no solution. Here he will prove that $\mathcal{E}$ cannot exist by first showing that the existence of $\mathcal{E}$ implies the existence of a process for determining whether a machine prints 0 infinitely often, but that implies the existence of a similar process to determine whether a machine prints 1 infinitely often. If you had the ability to determine whether a machine prints 0 infinitely often or 1 infinitely often (or both), you'd have the ability to determine whether a machine is circle-free. It's already been proven that such a process is impossible, so machine $\mathcal{E}$ must also be impossible.

> We will first show that, if there is a machine $\mathcal{E}$, then there is a general process for determining whether a given machine $\mathcal{M}$ prints 0 infinitely often.

Turing will demonstrate this through a rather odd method of defining variations of the arbitrary machine $\mathcal{M}$.

> Let $\mathcal{M}_1$ be a machine which prints the same sequence as $\mathcal{M}$, except that in the position where the first 0 printed by $\mathcal{M}$ stands, $\mathcal{M}_1$ prints $\bar{0}$. $\mathcal{M}_2$ is to have the first two symbols 0 replaced by $\bar{0}$, and so on. Thus, if $\mathcal{M}$ were to print
>
> $$ABA01AAB0010AB\ldots,$$
>
> then $\mathcal{M}_1$ would print
>
> $$ABA\bar{0}1AAB0010AB\ldots$$
>
> and $\mathcal{M}_2$ would print
>
> $$ABA\bar{0}1AAB\bar{0}010AB\ldots.$$

If you had a machine $\mathcal{M}$, could you define a machine that reads the Standard Description of $\mathcal{M}$ and manufactures the Standard Descriptions of $\mathcal{M}_1$, $\mathcal{M}_2$, and so forth? Turing says yes, and he calls this machine $\mathcal{T}$:

> Now let $\mathcal{T}$ be a machine which, when supplied with the S.D of $\mathcal{M}$, will write down successively the S.D of $\mathcal{M}$, of $\mathcal{M}_1$, of $\mathcal{M}_2$, ... (there is such a machine).

To convince ourselves that $\mathcal{T}$ is plausible, let's consider that very simple machine that alternatively prints 0 and 1, that is, the binary form of 1/3 without skipping any spaces:

q_1	None	$P0, R$	q_2
q_2	None	$P1, R$	q_1

That's machine $\mathcal{M}$. Here's machine $\mathcal{M}_1$:

q_1	None	$P\overline{0}, R$	q_4
q_2	None	$P1, R$	q_1
q_3	None	$P0, R$	q_4
q_4	None	$P1, R$	q_3

All the original configurations (all two of them) have simply been duplicated and given different *m*-configurations. In the first set of configurations, every line that printed 0 now prints $\overline{0}$ and then jumps to the appropriate configuration in the second set. $\mathcal{M}_2$ has three sets:

q_1	None	$P\overline{0}, R$	q_4
q_2	None	$P1, R$	q_1
q_3	None	$P\overline{0}, R$	q_6
q_4	None	$P1, R$	q_3
q_5	None	$P0, R$	q_6
q_6	None	$P1, R$	q_5

You might notice that these modified machines never enter configurations q_2, but that's just a fluke of this particular machine.

It is therefore entirely plausible that $\mathcal{T}$ exists. Notice the relationship between these $\mathcal{M}$ machines: If $\mathcal{M}$ never prints 0, then neither does $\mathcal{M}_1$, $\mathcal{M}_2$, and so forth. If $\mathcal{M}$ prints 0 just once, then $\mathcal{M}_1$ never prints 0, and neither does $\mathcal{M}_2$, and so forth. If $\mathcal{M}$ prints 0 twice, then $\mathcal{M}_1$ prints 0 once, $\mathcal{M}_2$ never prints 0, and so forth. If $\mathcal{M}$ prints 0 infinitely often, then so does $\mathcal{M}_1$, $\mathcal{M}_2$, and so forth.

You'll recall that $\mathcal{E}$ is assumed to determine whether a machine ever prints 0.

> We combine $\mathcal{F}$ with $\mathcal{E}$ and obtain a new machine, $\mathcal{G}$. In the motion of $\mathcal{G}$ first $\mathcal{F}$ is used to write down the S.D of $\mathcal{M}$, and then $\mathcal{E}$ tests it, : 0 : is written if it is found that $\mathcal{M}$ never prints 0; then $\mathcal{F}$ writes the S.D of $\mathcal{M}_1$, and this is tested, : 0 : being printed if and only if $\mathcal{M}_1$ never prints 0, and so on.

$\mathcal{G}$ uses $\mathcal{F}$ to generate the Description Numbers of $\mathcal{M}$, $\mathcal{M}_1$, $\mathcal{M}_2$, and so forth, and $\mathcal{E}$ to determine whether the resultant machine ever prints 0. If the resultant machine never prints 0, $\mathcal{G}$ prints 0.

The result is this: If $\mathcal{M}$ never prints 0, or prints 0 only a finite number of times, then $\mathcal{G}$ prints 0 infinitely often. If $\mathcal{M}$ prints 0 infinitely often, then $\mathcal{G}$ never prints 0.

> Now let us test $\mathcal{G}$ with $\mathcal{E}$. If it is found that $\mathcal{G}$ never prints 0, then $\mathcal{M}$ prints 0 infinitely often; if $\mathcal{G}$ prints 0 sometimes, then $\mathcal{M}$ does not print 0 infinitely often.

That means $\mathcal{G}$ can tell us that $\mathcal{M}$ prints 0 infinitely often. It tells us this by never printing 0.

> Similarly there is a general process for determining whether $\mathcal{M}$ prints 1 infinitely often. By a combination of these processes we have a process for determining whether $\mathcal{M}$ prints an infinity of figures, *i.e.* we have a process for determining whether $\mathcal{M}$ is circle-free. There can therefore be no machine $\mathcal{E}$.

By another proof by contradiction, Turing has shown that $\mathcal{E}$ cannot exist because it would ultimately imply the existence of $\mathcal{D}$ — the machine that determines whether any machine is circle-free — and that machine cannot exist.

Turing finishes this section with a reminder that we really need to examine this assumed equivalence between human computers and Turing Machines because we've been relying on it quite a lot.

> The expression "there is a general process for determining ..." has been used throughout this section as equivalent to "there is a machine which will determine ..". This usage can be justified if and only if we can justify our definition of "computable".

That examination will come in the next section.

Turing then hints at another aspect of this demonstration that won't be explored until Part III of this book. Turing began by interpreting the output of Turing Machines as "computable numbers," but machines can be more flexible than that. For example, consider a machine that prints a sequence like this:

0011010100010100010100010000010 . . .

That might look like a number, but it's actually the output of a "prime number" machine that we might denote by *IsPrime*(*n*). For the *n*th figure in this sequence (beginning with *n* equal to zero), *IsPrime*(*n*) is 1 if *n* is prime, and 0 if *n* is not prime. The sequence printed by the machine indicates that 2, 3, 5, 7, 11, 13, 17, 19, 23, and 29 are all primes. Such a machine is entirely plausible, but it's not really computing a number. Instead it's telling us something about the natural numbers.

> For each of these "general process" problems can be expressed as a problem concerning a general process for determining whether a given integer *n* has a property *G*(*n*) [*e.g.* *G*(*n*) might mean "*n* is satisfactory" or "*n* is the Gödel representation of a provable formula"], and this is equivalent to computing a number whose *n*-th figure is 1 if *G* (*n*) is true and 0 if it is false.

Turing has now, in a very small way that will become more apparent in Part III of this book, established a link between his computing machines and mathematical logic. The symbols 1 and 0 not only serve as binary digits, but — as George Boole realized many years ago — they can also symbolize *true* and *false*.

Consider a bunch of functions that have arguments of natural numbers and which return values of *true* and *false* (or 1 and 0):

IsPrime(*n*)
IsEven(*n*)
IsOdd(*n*)
IsLessThanTen(*n*)
IsMultipleOfTwentyTwo(*n*)

and so forth. These are sometimes known as Boolean functions, and they can be implemented by Turing Machines that print sequences of 0s and 1s for *n* equal to 0, 1, 2, 3, and so forth. The *IsOdd* function prints the same alternating sequence as Turing's first example machine.

Turing has established that these computable sequences are enumerable. So, too, are the actual function names! They can be alphabetized, for example. In Cantor's notation of transfinite numbers, the cardinality of the set of all computable and alphabetizable Boolean functions of natural numbers is $\aleph_0$, the cardinality of enumerable sets.

Each Boolean function returns *true* or 1 for a subset of the natural numbers. For example, *IsPrime* returns 1 for the following set of natural numbers:

$$\{2, 3, 5, 7, 11, 13, \ldots\}$$

Each of these Boolean functions is associated with a different subset of the natural numbers. As you might recall from Chapter 2, the set of all subsets is called a *power set*, and if the original set has a cardinality of $\aleph_0$, then the power set has a cardinality of $2^{\aleph_0}$.

The set of all *conceivable* Boolean functions has a cardinality of $2^{\aleph_0}$, while the set of all *computable* Boolean functions (and indeed, the set of all Boolean functions that can be described with a name in the English language) has a cardinality of χ_0. That's another big gap between the conceivable and the computable.

11 Of Machines and Men

Alan Turing wrote at the beginning of the first section of his paper (page 68 of this book) of his definition of computable numbers that "No real attempt will be made to justify the definitions given until we reach §9." We have now reached Section 9, and the pages that follow have been called by Turing's biographer Andrew Hodges "among the most unusual ever offered in a mathematical paper."[1]

Turing will attempt to demonstrate that the capabilities of a Turing Machine are equivalent to a human computer carrying out a well-defined mathematical process. Therefore, if an algorithmic process is insolvable by a Turing Machine, it is also unsolvable by a human. This idea — generally expressed more formally — has come to be known as the Turing thesis or (in a related form) the Church-Turing thesis. It's called a "thesis" because it's much too amorphous a concept to be subjected to a rigorous mathematical proof. The thesis nonetheless extends to other digital computers: Their computational capabilities are no greater than the Turing Machine.

Only the first part of Section 9 appears in this chapter; the remainder requires some background in mathematical logic and will conclude in Part III of this book. For the most part, I will not interrupt Turing's analysis. Here's a summary by Martin Davis:

> Turing's "analysis" is a remarkable piece of applied philosophy in which, beginning with a human being carrying out a computation, he proceeds by a process of elimination of irrelevant details, through a sequence of simplifications, to an end result which is the familiar model consisting of a finite state device operating on a one-way infinite linear tape.[2]

---

[1] Andrew Hodges, *Alan Turing The Enigma* (Simon & Schuster, 1983), 104

[2] Martin Davis, "Why Gödel Didn't Have Church's Thesis," *Information and Control*, Vol 54, Nos 1/2, (July/Aug 1982), 14

[249]

9. The extent of the computable numbers.

No attempt has yet been made to show that the "computable" numbers include all numbers which would naturally be regarded as computable. All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically. The real question at issue is "What are the possible processes which can be carried out in computing a number?"

The arguments which I shall use are of three kinds.

(*a*) A direct appeal to intuition.

(*b*) A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).

(*c*) Giving examples of large classes of numbers which are computable.

The (*b*) argument is in Part III of this book; the (*c*) argument continues in Section 10 of Turing's paper.

Once it is granted that computable numbers are all "computable", several other propositions of the same character follow. In particular, it follows that, if there is a general process for determining whether a formula of the Hilbert function calculus is provable, then the determination can be carried out by a machine.

The "Hilbert function calculus" is the system of mathematical logic today commonly called "first-order predicate logic." It is within this logic that Hilbert defined the Entscheidungsproblem. It is unlikely that Turing knew that a process "carried out by a machine" is precisely what Heinrich Behmann called for in the earliest references to the Entscheidungsproblem (page 48). Behmann's address remained unpublished until recently.

I. [Type (*a*)]. This argument is only an elaboration of the ideas of § 1.

Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on

one-dimensional paper, *i.e.* on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent†. The effect of this restriction of the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as

[250]

17 or 999999999999999 is normally treated as a single symbol. Similarly in any European language words are treated as single symbols (Chinese, however, attempts to have an enumerable infinity of symbols). The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 9999999999999999 and 999999999999999 are the same.

---

† If we regard a symbol as literally printed on a square we may suppose that the square is $0 \leqslant x \leqslant 1, 0 \leqslant y \leqslant 1$ The symbol is defined as a set of points in this square, viz. the set occupied by printer's ink. If these sets are restricted to be measurable, we can define the "distance" between two symbols as the cost of transforming one symbol into the other if the cost of moving unit area of printer's ink unit distance is unity, and there is an infinite supply of ink at $x = 2, y = 0$. With this topology the symbols form a conditionally compact space

In the next sentence, Turing refers to a "computer." He is, of course, talking about a *human* computer.

The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be "arbitrarily close" and will be confused. Again, the restriction is not one which seriously affects computation, since the use of more complicated states of mind can be avoided by writing more symbols on the tape.

In 1972, Kurt Gödel wrote a brief note regarding Turing's analysis in this section that he labeled "A philosophical error in Turing's work."[3] Gödel argued that "*mind, in its use, is not static, but constantly developing*" and that mental states of mind might even converge on the infinite. These disagreements represent a fundamental clash between those who believe the mind to be ultimately a mechanical process of the brain, and those who do not.

Let us imagine the operations performed by the computer to be split up into "simple operations" which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the computer and his tape. We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the computer (possibly with a special order), and the state of mind of the computer. We may suppose that in a simple operation not more than one symbol is altered. Any other changes can be split up into simple changes of this kind. The situation in regard to the squares whose symbols may be altered in this way is the same as in regard to the observed squares. We may, therefore, without loss of generality, assume that the squares whose symbols are changed are always "observed" squares.

Besides these changes of symbols, the simple operations must include changes of distribution of observed squares. The new observed squares must be immediately recognisable by the computer. I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount. Let us say that each of the new observed squares is within L squares of an immediately previously observed square.

In connection with "immediate recognisability", it may be thought that there are other kinds of square which are immediately recognisable. In particular, squares marked by special symbols might be taken as imme-

[251]

diately recognisable. Now if these squares are marked only by single symbols there can be only a finite number of them, and we should not upset our theory by adjoining these marked squares to the observed squares. If, on the other hand, they are marked by a sequence of symbols, we

[3]Kurt Gödel, *Collected Works, Volume II Publications 1938–1974* (Oxford University Press, 1990), 306 Judson C Webb's introduction beginning on page 292 — and particularly the identification on page 297 of Gödel's belief that the human mind has an existence separate from the physical matter of the brain — is helpful in understanding Gödel's remarks Another analysis is Oron Shagrir, "Gödel on Turing on Computability," http //edelstein huji ac il/staff/shagrir/papers/Goedel_on_Turing_on_Computability pdf

> cannot regard the process of recognition as a simple process. This is a fundamental point and should be illustrated. In most mathematical papers the equations and theorems are numbered. Normally the numbers do not go beyond (say) 1000. It is, therefore, possible to recognise a theorem at a glance by its number. But if the paper was very long, we might reach Theorem 157767733443477; then, further on in the paper, we might find "... hence (applying Theorem 157767733443477) we have ...". In order to make sure which was the relevant theorem we should have to compare the two numbers figure by figure, possibly ticking the figures off in pencil to make sure of their not being counted twice. If in spite of this it is still thought that there are other "immediately recognisable" squares, it does not upset my contention so long as these squares can be found by some process of which my type of machine is capable. This idea is developed in III below.

When Turing describes "ticking the figures off in pencil," he is probably alluding to the similar machine operation of "marking" squares with non-numeric figures.

> The simple operations must therefore include:
>
> (*a*) Changes of the symbol on one of the observed squares.
>
> (*b*) Changes of one of the squares observed to another square within L squares of one of the previously observed squares.
>
> It may be that some of these changes necessarily involve a change of state of mind. The most general single operation must therefore be taken to be one of the following:
>
> (*A*) A possible change (*a*) of symbol together with a possible change of state of mind.
>
> (*B*) A possible change (*b*) of observed squares, together with a possible change of state of mind.
>
> The operation actually performed is determined, as has been suggested on p. 250, by the state of mind of the computer and the observed symbols. In particular, they determine the state of mind of the computer after the operation is carried out.

That's just the previous page of his paper to which he's referring.

> We may now construct a machine to do the work of this computer. To each state of mind of the computer corresponds an "*m*-configuration" of the machine. The machine scans *B* squares corresponding to the *B* squares observed by the computer. In any move the machine can change a symbol on a scanned square or can change any one of the scanned squares to another square distant not more than *L* squares from one of the other scanned
>
> [252]
>
> squares. The move which is done, and the succeeding configuration, are determined by the scanned symbol and the *m*-configuration. The machines just described do not differ very essentially from computing machines as defined in § 2, and corresponding to any machine of this type a computing machine can be constructed to compute the same sequence, that is to say the sequence computed by the computer.

That is, the *human* computer.

At this point, we stop for now. Turing's second argument in this section begins with a reference to the "restricted Hilbert functional calculus," followed by a statement in that calculus, and for that some background is required that begins Part III of this book.

Turing's fascination with the connection between human brains and machines continued long beyond his 1936 paper on computable numbers. Turing's other famous paper, "Computing Machinery and Intelligence," was published in the October 1950 issue of the philosophy journal *Mind*.

"Can machines think?" Turing asks. He then devises a test with a human being sitting at a teletypewriter. (The modern equivalent might be instant messaging, or anything else that doesn't allow people to see or hear who they're communicating with.) Let the person ask questions and receive answers. If there's actually a computer on the other end, and the person can't tell that it's a computer, then we should say that the computer is intelligent.

This has come to be known as the Turing Test, and it remains as controversial as ever. Anybody who has a pat objection to the Turing Test should read Turing's paper, which already has answers to many reasonable objections.

Turing prefers to deal with this question in terms of "intelligence" rather than "thinking" because "thinking" implies a certain activity going on *inside* the computer.

> The original question, 'Can machines think?' I believe to be too meaningless to deserve discussion. Nevertheless I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted.[4]

[4]Alan Turing, "Computing Machinery and Intelligence," *Mind*, Vol LIX, No 236 (October 1950), 442.

The end of the century has passed and, if anything, more people than ever know that whatever computers do, it is not "thinking." We have not come to expect our computers to be intelligent, and generally we work best with our computer applications when we believe they will act in a completely deterministic way. A computer program that attempts to do something "intelligent" often seems to resemble a two-year old staring up from a newly crayoned wall and pleading, "But I thought you'd *like* it."

In the alternative universe of science fiction, Turing's prediction was right on target, as demonstrated by the most famous fictional computer of all time:

> Whether Hal could actually think was a question which had been settled by the British mathematician Alan Turing back in the 1940s. Turing had pointed out that, if one could carry out a prolonged conversation with a machine — whether by typewriter or microphone was immaterial — without being able to distinguish between its replies and those that a man might give, then the machine was thinking, by any sensible definition of the word. Hal could pass the Turing test with ease.[5]

Alan Turing would have turned 56 years old in 1968, the year that both the book and movie of *2001* came out. He might have been amused by the concept of a computer so intelligent that it would experience a nervous breakdown.

In the summer of 1950, Turing moved to a house in Wilmslow, about ten miles south of Manchester. He had become interested in morphogenesis, which is the study of how cells in an organism develop and differentiate themselves to exhibit various patterns and forms. The research involved running simulations on the Manchester computer.

On March 15, 1951, Alan Turing was elected a Fellow of the Royal Society in recognition of his work on Computable Numbers. His sponsors were Max Newman and Bertrand Russell. That evening, the BBC broadcast a talk Turing had recorded entitled "Can Digital Computers Think?" (No recording of this broadcast or any recording of Turing speaking is known to exist.)

In December 1951, a chain of events was set in motion that would have serious consequences. Turing met a young man on the streets of Manchester. Arnold Murray had a working-class background, he was on probation for theft, and he was unemployed. Turing and Murray had lunch, met again, and went back to Turing's home together. They met several times over the next month.

Late in January 1952, Turing discovered that his house had been burgled. He reported it to the police, who came and dusted for fingerprints. When Turing confronted Arnold Murray, Murray pleaded innocent but said he knew who did it — an acquaintance named Harry. The police also identified Harry from

---

[5]Arthur C Clark, *2001: A Space Odyssey* (New American Library, 1968), ch. 16

the fingerprints taken from Turing's house. Harry was already in custody for something else. When questioned about the Turing robbery, Harry gave the police an earful about what was going on between Turing and his friend.

On February 7, 1952, the day after George VI died and his eldest daughter Elizabeth ascended to the throne, the police called on Alan Turing. After some questioning, Turing admitted to them the nature of his relationship with Murray. This confession made Turing subject to arrest under Section 11 of the Criminal Law Amendment Act of 1885:

> Any male person who, in public or private, commits, or is a party to the commission of, or procures or attempts to procure the commission by any male person of, any act of gross indecency with another male person, shall be guilty of a misdemeanor, and being convicted thereof shall be liable at the discretion of the court to be imprisoned for any term not exceeding two years, with or without hard labour.[6]

The term "gross indecency" was not defined in the law, but was generally taken to mean acts such as mutual masturbation and oral sex. Other statutes covered the more serious offense of anal sex (or "buggery" as it was known within the British legal system).

Section 11 was a notorious law that was controversial from its very beginning. The Criminal Law Amendment Act of 1885 describes itself as "An Act to make further provision for the protection of women and girls, the suppression of brothels, and other purposes." The law raised the age of consent from 13 to 16, and contained several provisions intended to prevent women from exploitation, such as being drugged in brothels or abducted into prostitution.

The law had originally floundered in the House of Commons for a couple years but then became more urgent following a series of articles by liberal journalist William Thomas Stead (1849–1912) concerning child prostitution. Stead's courageous exposé culminated with his actual purchase of a 13-year old girl from her parents.

Following the public uproar over Stead's articles, the Bill was revived. Section 11 was introduced by Member of Parliament Henry Labouchère on August 6, 1885, and was added to the Act the next day, just a week before the Act eventually passed. There was some question at the time whether it was proper to add this section to a bill whose focus was the protection of young girls and women.[7]

---

[6]Wording obtained from http //www swarb.co uk/acts/1885Criminal_Law_AmendmentAct.shtml (accessed April 2008)

[7]H Montgomery Hyde, *The Love That Dared Not Speak Its Name· A Candid History of Homosexuality in Britain* (Little, Brown and Company, 1970), 134 This book was originally published in England under the title *The Other Love*

Section 11 specifically targeted men, and the acts covered under the term of "gross indecency" had never been illegal in Britain before, at least not when performed by consenting adults in private. Even at the time the "in private" clause seemed to allow the law to be used for blackmail.[8]

The most famous victim of Section 11 was Oscar Wilde (1854–1900), who was prosecuted under the law in 1895. Wilde served his time doing hard labor, which probably hastened his death.

By the 1950s, however, different methods of punishment were available. Turing pleaded guilty with the understanding that he was to be placed on a year's probation, during which he was to have hormone treatments.

Experiments with treating homosexuality using sex hormones had begun in the 1940s. At first it was believed that homosexuality resulted from insufficient male hormones, but the administration of testosterone actually had the opposite of the anticipated result. Female hormones were then tried on homosexual men, and those seemed to have more of a desired effect.[9] By the time of Turing's conviction, this treatment was known as organotherepy, but was also called "chemical castration" and seemed intended more to humiliate than anything else. The estrogen rendered Turing impotent and made his breasts grow.

The early 1950s were not a good time to be identified as a homosexual. In the United States, the "red scare" of the early 1950s soon metamorphosed into another type of witch hunt. There actually weren't very many communists working in the State Department, but there were plenty of closeted gay people working in government jobs in Washington D.C. "Over the course of the 1950s and 1960s, approximately 1,000 people were dismissed from the Department of State for alleged homosexuality."[10]

In theory, the term "security risk" could be applied to anyone who might have a tendency to divulge government secrets. In practice the term was basically a euphemism for "homosexual."[11] The assumption was that homosexuals could be blackmailed into revealing state secrets. However, the best example anyone could come up with of this actually happening involved the head of Austrian intelligence before World War I, and it was never quite clear what the real story was.[12]

What was happening to gays in the United States government also had implications for Great Britain. In 1951, the U.S. State Department began advising the British Foreign Office about "the homosexual problem" in the government, and later pressured the British government to be more diligent about supposed security problems regarding homosexuals.[13]

---

[8]Ibid, 136

[9]Hodges, *Alan Turing*, 467–471

[10]David K Johnson, *The Lavender Scare The Cold War Persecution of Gays and Lesbians in the Federal Government* (University of Chicago Press, 2004), 76

[11]Johnson, *Lavender Scare*, 7–8

[12]Johnson, *Lavender Scare*, 108–109

[13]Johnson, *Lavender Scare*, 133.

Alan Turing's employment options were certainly becoming more restricted. A top-secret government job such as Turing had during the war would now be inconceivable, nor would Turing be able to travel to America again. A 1952 law prohibited admission to "aliens afflicted with psychopathic personality," which was interpreted to mean homosexuality.[14]

Was this enough to make Turing suicidal? We don't know.

On the streets of England as well as in the government, life was getting more difficult for gay men. When Sir John Nott-Bower was appointed Commissioner of the London Metropolitan Police in 1953, he swore he would "rip the cover off all London's filth spots." The same year the Home Office issued directives for a new drive against "male vice." At least one London magistrate was tired of coddling and wanted convicted men to be "sent to prison as they were in the old days." From the end of 1953 through early 1954, newspaper headlines heralded the prosecutions of several men.[15]

Had Turing a new relationship with a man who was now threatening to blackmail him?

Or was Turing's suicide merely a careless accident, as his mother believed?

It is indicative of our ignorance that one of the most persuasive explorations of Turing's state of mind during this period comes not from a history or biography but from a novel. Novelist Janna Levin (who is also a professor of astronomy and physics at Barnard College) portrays a man humiliated beyond his ability to express it:

> He doesn't know how to voice his humiliation or even how to experience it. It rattles around in him like a broken part, dislodged and loose in his metal frame. The humiliation won't settle on one place, sink in where it would no doubt fester but at least could be quarantined and possibly even treated. If not steadily eroded by the imperceptible buffing waves of time, then maybe more aggressively targeted, excised by his Jungian analyst. But the shame just won't burrow and bind.[16]

We just don't know what was different about the evening of June 7, 1954. We don't know what prompted Turing to dip his regular evening apple in cyanide before he went to bed.

He was found dead the next morning. Alan Turing was 41 years old.

---

[14]Hodges, *Alan Turing*, 474

[15]Hyde, *The Love That Dared Not Speak Its Name*, 214-6

[16]Janna Levin, *A Madman Dreams of Turing Machines* (Alfred A Knopf, 2006), 214