Alan M. Turing, On Computable Numbers, with an Application to the *Entscheidungsproblem*

Introduction: (pp. 230-231)

The first sentence here is very important: Turing defines what he means by "computable". In this paper, the word "computable" has a similar role to that of "effectively calculable" in Church's paper, but note that Turing's definition is somewhat more well-defined. Also, Church was concerned exclusively with *integers* whereas Turing works with *reals*: however, Church was actually working with *functions* over the integers, and this class of functions has the same cardinality as the reals.

At the time of writing, of course, it was not obvious how the class of "computable" real numbers related to the class of all real numbers. Towards the end of this section Turing points out that the former is computable, and thus must be strictly smaller than the latter.

Section 1: Computing Machines (pp. 231-232)

A very important point here is that Turing is modelling "a **man** in the process of computing a real number": thus, Turing machines are models of *people* not (other) *machines*.

His "*m* -configurations" (presumably "machine configurations") are just the states of the machine.

The symbol 𝔖 is a capital S in Fraktur font, and here 𝔖(n) is just the symbol at position *n* on the input tape.

He defines the "configuration" of the machine as being a tuple consisting of the current state and symbol *(qn, 𝔖(r))* .

The use of the tape is slightly different to the modern version: the symbols on Turing's tape are partitioned into symbols that are part of the output, which are read-only, and those which are just "rough notes", which are read-write. This distinction isn't important in terms of the expressive power of the machine: the standard modern presentation usually assumes all tape symbols are read-write.

Section 2: Definitions (pp. 232-233)

Some fairly standard definitions here. The *c* -machines are those which need some user interaction, and Turing will not consider them further.

Turing's machines print results in binary format: the other "rough note" symbols can be of any kind. He assumes the sequence of digits produced by the machine can always be interpreted as a real number by imagining a 'decimal' point in front of it For example, a machine producing *01010101...* would be the (binary) equivalent of one-third. Similarly we could then stick an integer in front of the decimal point to get any real number.

Turing introduces another definition of the word configuration: this time a " *complete* configuration" is a triple consisting of the current state, the current position on the tape, and the whole contents of the tape. Thus, if we saved the "complete configuration" for a machine at any stage of its processing, we could restore it later and continue on with the computation.

The definition of *circular* and *circle-free* machines is important here. Turing's machines print out a real number - even if this is actually a rational number or an integer, Turing sees his machine as continuing to print zeros at the end. Thus these *circle-free* machines that print an infinite number of symbols are actually the desirable ones. The other kind of machine, the *circular*machine is one that gets stuck (and halts, or continues to work without printing any more output symbols).

Note the difference here with modern presentations: there is no "halt with success" option and, moreover, non-halting machines may be successful or not, depending on whether they are still capable of generating output.

Section 3: Examples of computing machines (pp. 233-235)

Turing uses the Fraktur font here for state names (and the typography isn't so good): the states of his machine are actually *b, c, k* and *e* (and should look like 𝔟, 𝔠, 𝔨 and 𝔢).

The machine at the bottom of page 233 skips every second square as these are reserved for "rough work", and not needed in this machine (this is the purpose of states c and k).

**[pg 234]**

His second machine has states *o, q, p, f* and *b*. If we remember that the output will only consist of the figures 1 and 0 (on every second square) we can see that the other symbols ə and *x* are just for rough work. Actually, the əə symbol-sequence is just used to mark the start of the tape, and *x* is the only non-output symbol written by the machine after this.

The upside-down *e* symbol ə is called a [schwa](http://en.wikipedia.org/wiki/Schwa#The_term).

Before reading further you should try to figure out the operation of this machine by yourself.

For confirmation, we could summarise the operation of the machine as:

* 𝔟: only used once, at the start, to set up the initial əə0\_0 sequence
* 𝔬: mark each symbol of the previous 111..11 sequence with an *x* in the blank just after it
* 𝔮: go right to the end of the tape and write an extra 1
* 𝔭: go back left, find the next *x* and erase it (if there are no more, go to state 𝔣)
* 𝔣: go right to the end of the tape and write an extra 0

Thus the (𝔭-𝔮) combination forms an inner loop that adds a 1 each time it erases an *x*. Note that state 𝔬 always goes to state 𝔮 before we move on to state 𝔭, thus ensuring an extra 1 gets printed each time around the loop. The outer loop is roughly of the form 𝔬-𝔮-(𝔭-𝔮)-(𝔭-𝔮)-...-(𝔭-𝔮)-𝔣, and is run once for every extra sequence of 1s appended to the end of the tape.

**[pg 235]**

Turing takes us through a sample run of this machine. Here, there are 14 steps of the machine, each represented by a complete configuration.

For example, when Turing writes the complete configuration  this shows three things: the whole contents of the tape, the current state (𝔮) and the position of the tape-head (the square above the (𝔮)).

The linear form of the tape he then presents is less easy to follow, but he will use this later when he comes to encoding his universal machine - see the reference to this description "(C)" on page 242.

Finally in this section Turing explains his alternate-squares policy (this might have been useful earlier in the discussion). Note that he marks a figure by writing a symbol to its *right*, as in the machine he has just presented.

Section 4: Abbreviated tables (pp. 235-239)

In this section Turing introduces a shorthand notation for his machines, and gives us some more examples, building up a library of machines that he will use later in the paper.

His 'skeleton tables' simply have *m*-functions instead of states. These functions are really just macros that can be instantiated to give actual states of the Turing machine. Thus when he writes *𝔣(ℭ, 𝔅, α)* he intends that the user will supply values for the parameters: a state for each of ℭ and 𝔅, and a symbol to use in place of α.

The example he gives is of a skeleton table that, when expanded, would be a machine that:

* first scanned left until it found a ə (denoting the leftmost end of the tape),
* then scanned right until either it found an α or two blanks (denoting the rightmost end of the tape),
* and goes to either state ℭ (found) or 𝔅 (not found) respectively.

In the narrative that follows this on page 236, Turing takes some care to explain that these are not a new programming mechanism: in modern terms we would call these "macros" rather than "functions". This is important, since Turing wants to make sure that his machine descriptions always have *finite* length, resulting from having just a finite set of states and symbols. Thus you should think of a two-phase operation:

* a "compile time" where each use of an *m*-function is expanded to an actual state, so we get an ordinary machine,
* and then a separate "run-time" for this machine, as before.

**[pg. 237]**

Turing now gives us an example to show the power of his *m*-functions. His notation changes a bit, and he allows himself a kind of no-op transition, for example from 𝔢 to 𝔣 here (this is the 𝔣 defined on the previous page). He also uses overloading: there's a two-argument and a three-argument version of 𝔢, which should be considered different *m*-functions. Really the main function being defined here is 𝔢(𝔅, α), which just erases all α symbols and goes to state 𝔅.

Again, Turing makes an effort to justify that this is not proper recursion, even in the definition of two-argument 𝔢. Think of it this way: the first time we see an actual instantiation of one of these, say 𝔢(𝔟,x) where 𝔟 and x is an existing, known state and symbol, we generate a new state for this, which he calls 𝔮 here. Then the apparently recursive definition turns into a relatively tame instruction of the form: 𝔮 --> 𝔢(𝔮, 𝔟, x). This simply stays in state 𝔮 erasing *x*s; when there are none left, it goes to state 𝔟.

This isn't a particularly *efficient* definition, since 𝔣 is called anew for each *x* (thus going all the way left of the tape and coming back again), but efficiency is not considered at all relevant here.

In the remainder of this section Turing gives a relatively long list of definitions, starting with 𝔭𝔢 on page 237 and going down to 𝔢 and 𝔢1 on page 239. These may seem a strange choice of functions to define but, of course, he intends to use them to build his universal machine in subsequent sections of the paper. They broadly serve to provide Turing with erase, copy and print functions that he will need later.

I've typed out the definitions from this section in a [separate sheet](https://2017.moodle.maynoothuniversity.ie/pluginfile.php/547197/mod_resource/content/3/turing-sec4.pdf) that might be a bit easier to read than the originals from the paper (I've also corrected a few minor typos).

Section 5: Enumeration of computable sequences (pp. 239-241)

In this section Turing's goal is to show that he can calculate a unique Gödel number for each of his machines. This will serve two purposes. First, it is an important ingredient in constructing his universal machine in the next section. Second, because such an encoding exists, he has shown that the set of all Turing machines much be enumerable.

He encodes a Turing machine as simply a list of the instructions it contains (i.e he's encoding the *instructions* in the machine). To make this easier he changes notation slightly so that each instruction becomes a five-tuple: the *E* (erase) operation is interpreted as just printing the same symbol as the one already there, and he introduces a *N* operation which means "no-move".

He now observes that we can list the (finite) set of five-tuples, separated by semi-colons. He encodes states and symbols by the letter "D" followed by some number of either "A"s or "C"s. Thus, state 3 is encoded as "DAAA" while symbol 4 would be encoded as "DCCCC". (Thus blank, 0 and 1 are "D", "DC", "DCC").

Since he now has only 7 letters in his encoding, he can translate these to digits, and concatenate them together to get a number (the *description number*) that represents the whole list of instructions for this machine.

Note the definition in the last paragraph of "satisfactory" number: the encoding of a circle-free machine.

Section 6: The universal computing machine (pp. 241-243)

In this short section Turing just extends his encoding to work for the run-time configuration of the Turing machine - i.e. the current state, the whole input tape, and the current position on that tape. He as already encoded such "complete configurations" on one line (the line marked "(C)" on page 235), so he just needs to use the encoding from the last section on this to get a description number (DN) for the configuration.

We can now see how Turing's "universal" machine will work. If we want the universal machine to simulate some other machine ℳ, then at any stage this universal machine will have two encodings on its tape: the SD for ℳ's instructions, and the SD for the current complete configuration of ℳ. As it runs, it will simply work out the next complete configuration for ℳ, and append the SD for this complete configuration to the end of the tape.

Section 7: Detailed description of the universal machine (pp. 243-246)

Now Turing proves that there *can* be a universal machine by showing us how to build one. There's not much you can do here other than go through the instructions and work them out.

Turing reminds us about how we should instantiate his *m*-functions - the function 𝔢 has already been described earlier (as Turing notes, on page 239) so this is just an example.

The overall layout on the tape:

* the F-squares will have the SD for the machine, then a "::", then followed by the sequence of SDs for its complete configurations,
* *states* are a D followed by A's (starting with "DA" for state 1),
* *symbols* are a D followed by C's (starting with "D" for a blank),
* *instructions* (in the SD for the machine) are separated by semi-colons,
* *complete configurations* are separated by colons,

The one extra auxiliary function that he defines is 𝔠𝔬𝔫 - nearly everything defined in the remainder of this section is a proper state, not an *m*-function. All 𝔠𝔬𝔫 does is to mark out the next configuration, i.e. the next DA...DC... sequence, with the letter α. Note that Turing gets maximum value out of this: the configuration could either be the first two parts of the five-tuple the makes up an instruction, or the sub-section of the (current) complete configuration where the tape head is.

**The table for U**

* 𝔟 (begin)   
  The start state: find the "::" and write the initial configuration, i.e. the start state (q0) and an empty tape.
* 𝔞𝔫𝔣 ("anfang" is German for beginning)  
  This is the start of the overall read-execute loop.   
  Call 𝔠𝔬𝔫 to mark the (state,symbol) in the current complete configuration with *y*.
* 𝔨𝔬𝔪 (kom = compare?)  
  Now moving left, find the next (thus rightmost) instruction to check: look for the semi-colon that starts it.  
  As instructions are checked, they are marked at the semi-colon with a *z*.  
  Call 𝔠𝔬𝔫 to mark the instruction's first (state,symbol) pair with an *x*.
* 𝔨𝔪𝔭 (kmp = compare?)  
  Now we must check the two (state,symbol) pairs to see if this instruction matches the current configuration.  
  If it doesn't, go back to 𝔨𝔬𝔪 and try the next instruction. if it does match, go to 𝔰𝔦𝔪. Turing typo: he hasn't actually defined a three-argument 𝔢, so we have to imagine it erasing all *x* and *y* markers, and going to 𝔨𝔬𝔪
* 𝔰𝔦𝔪 (simulate an instruction)  
  Find the *z* at the start of the instruction, erase it, and erase the *x* that mark the (state,symbol) pair following it.  
  𝔰𝔦𝔪2 marks the action with a *u*.  
  Turing typo: the second ("not A") action here should go left first, i.e. it should be *L,Pu,R,R,R*   
  𝔰𝔦𝔪3 marks the next (state,symbol) pair with an *y*.
* 𝔪𝔨 (mk = mark)  
  Turing typo: note that the rule for 𝔪𝔨 should be 𝔤(𝔪𝔨1, :)  
  First shuttle right to find the last complete configuration (after the rightmost :)  
  𝔪𝔨1: find the 'state' part of the complete configuration (i.e. DAAA...A)  
  𝔪𝔨2: The symbol before this is marked with an *x\_  
  𝔪𝔨3: the section of the tape before this (all symbols) is marked with an \_v*   
  𝔪𝔨4: the markers on the (state,symbol) pair of the current configuration are erased (these won't be part of the next complete configuration)  
  𝔪𝔨5: the remaining section (all symbols) to the right of this is marked with an *w*.
* 𝔰𝔥 (sh = show?)  
  Go back (left) to the instruction - marked with a *u*.   
  𝔰𝔥2/𝔰𝔥3: Go right to find the action part.  
  𝔰𝔥4: if it is DC print a 0 at the end, 𝔰𝔥5: if it is DCC print a 1 at the end.  
  Turing typo: in 𝔰𝔥2, the first option (on "D") should go to 𝔰𝔥3 not 𝔰𝔥2
* 𝔦𝔫𝔰𝔱 (inst = instruction)  
  The goal here is to work out the next complete configuration based on the L/R/N action and the current marked configuration. First go back (left) to the instruction (marked with a *u*) and find out what the action is.   
  Then use 𝔠𝔢5 to copy the five parts of the next complete configuration to the end *in the right order*.  
  Note that this always starts with (the section marked) *v* and ends with *w*. Also *y* marks the next state, *x* the old symbol (before the one we're replacing) whose position has to be decided, and *u* marks the new symbol being printed on the tape as per the instruction.
* 𝔬𝔳 (ov = over?)  
  Erase all the remaining (u,v,w,x,y) markers and go back to 𝔞𝔫𝔣 to start the next instruction.

A few typos have been spotted since Turing published this paper (some noted above). Emil Post went through some of them in a paper called *Recursive Unsolvability of a Problem of Thue* in (Journal of Symbolic Logic, Vol. 12, No. 1. 1947). Here are his main corrections (most noted above):

If you really want to get into the details, Donal Davies (who worked with Turing at the National Physical Laboratory in the late 1940s) has published a paper on *Corrections to the Universal Computing Machine* in a book by Jack Copeland called *The Essential Turing*.

Section 8: Application of the diagonal process (pp. 246-248)

Before Turing gives his diagonalisation argument he first rules out a "fake" version. This fake version tries to redo Cantor's proof (of the uncountability of the reals) except using *computable*reals. The idea, as with Cantor, is to change the *n*th digit of the *n*th real number: since these are computable reals they are in binary, and *1-φn(n)* serves to flip the *n*th bit of the *n*th real number.

The problem with this (the "fallacy" as Turing calls it) is that this new number can't really be *computed*, since we would have to have some way of enumerating only those machines that were circle-free (i.e. that didn't halt). This in turn would mean that we could recognise such machines without running them (to infinity), which **we** know we can't do - though Turing now has to prove this.

**[pg. 247]**

This is the core result here, all on this page (what we would nowadays call the undecidability of the *halting problem*).

This is a proof by contradiction: Turing supposes the existence of a machine 𝒟 that can decide whether or not another machine is circle-free. He then goes on to describe a machine ℋ which will combine 𝒟 with the universal Turing machine. It works just like in the "fallacy" argument Turing described earlier. We get an enumeration of the natural numbers going, and for each one of these we use the machine 𝒟 to check if the corresponding Turing machine is circle-free or not. Here, *R(N)* is just the number of machines between 1 and *N* that have been found to be circle-free. If it is circle-free, say the *R(N)-th* circle-free machine, then we generate its *R(N)-th* digit. (Turing doesn't bother to flip the bit here: we can do this later once we get this part working).

Assuming 𝒟 works as expected, giving a yes or no answer in finite time, then the machine ℋ he's just described is circle-free.

The last part then is to close the trap: we feed ℋ its own encoding (we assume this has some DN, say *K*) - that is, we ask ℋ to decide if it is circle-free. Since we know ℋ is circle free, then the answer can't be 'u'. However, if the answer is 's' then the process we described will call on the universal machine and set it going with the SD corresponding to *K* to generate its first *R(K)*outputs. This will be fine for the first outputs between 1 and *R(K)-1*. However, when we try to generate *R(K)*th output, the universal machine will follow this whole process again, and set ℋ going to get its *R(K)*th output. This process is circular and will never finish, contrary to what we have said. Thus the original assumption, that 𝒟 could exist, is not true.

**[pg. 248]**

In this last part of section 8 Turing presents a corollary to his main theorem: not only can we not have a machine that decides on circle-freeness, we can't even have a machine that answers simple questions like "Does a machine ever produce a 0"?

The proof is by contradiction so Turing assumes that there exists some machine ℰ,and builds several others that will eventually contradict the result he's just proved (on page 247). The three machines all work by taking the S.D. of any machine, call it ℳ, as input.

* ℰ is a machine that decides whether a machine ℳ ever produces a 0.
* ℱ is a machine that enumerates machines like ℳ, but with the first, first two, first three (etc.) 0s in the output deleted.  
  Turing uses 0 with a bar over it here, which I think we can interpret as "anything but 0".
* 𝒢 combines ℰ and ℱ. Given any version of ℳ, say ℳk, generated by ℱ, we can then use ℰ on this to see if it produces any 0s at all. Since we're assuming ℰ works, we can generate a 0 if ℳk never produces a 0.

Now that we've built 𝒢, we can feed it (or rather its S.D.) in to ℰ. If ℰ says 𝒢 never produces a 0, then ℳ must have produced infinitely many of them; otherwise ℳ must have stopped producing 0s at some (finite) point, causing 𝒢 to produce a 0 when it got this far.

Applying the same argument to the production of 1s would then allow us to solve the question of whether ℳ produces infinitely many 0s and 1s, i.e. whether ℳ is circle-free. This contradicts the result on page 247, and so ℰ cannot exist.

Section 9: The extent of the computable numbers (pp. 249-254)

In this section Turing elaborates on the justification for his approach that he mentioned in the first section. When Turing uses the word "computer" here (e.g. on page 250) he is referring to a **person** who computes: i.e. calculates things with pen and paper - not a machine.

**[pg. 252-253]**

In subsection II here Turing gives a "logic" based justification for his choice of machine, where logic is expressed in terms of what he calls the *Hilbert functional calculus*, which we would nowadays call the predicate calculus. The basic idea here is that we can mimic the operations of the logical calculus on the tape of a Turing machine, including the successor function *F(x,y)* so that we can build up a basic set of axioms that includes the Peano axioms. Turing calls this basic set of axioms 𝔘, and he uses the term *F(n)* simply to mean that all the numbers up to *n* are defined.

Turing asserts that, given any predicate calculus statement *that is provable*, he can construct a Turing machine 𝒦 to churn out a proof of that statement. There's no magic here: Turing just envisages his machine generating all possible proofs and waiting for the right one to come up.

As a special case, if we are given some predicate Gα(x) about the *x*th element of some sequence α, such that either Gα(x) or its negation is provable from the Peano axioms, then Turing can construct a macine 𝒦 α that will eventually churn out either the proof of Gα(x) or its negation.

There are two important caveats here with the proof-enumerating machine 𝒦:

* It only works for *provable* formulas, so it is not circumventing Gödel's theorem in any way
* It can't decide *in advance* if a formula is provable, so we still haven't solved the *Entscheidungsproblem*

Turing adds an interesting note at the end of subsection II, distinguishing between computable numbers and the larger class of well-defined numbers (or *definable* numbers). Thus, even though the set of all satisfactory numbers is a subset of the natural numbers, and is well-defined, we cannot have a general process for deciding if any number is actually in this set. We can, of course, have specific processes for easy cases (e.g. those machines with no loops), but no *general* process.

Section 10: Examples of large classes of numbers which are computable (pp. 254-258)

Thus far, Turing has only defined the concept of a computable real number (or a "computable sequence"), and in this section he wants to investigate what other mathematical concepts the word "computable" can be attached to: functions, limits of sequences, sums of power series etc. Note that throughout this section Turing uses the phrase "integral variable" to mean a variable whose values are natural (i.e. whole) numbers. Such variables can be extended to cover (what we call) integers and rational numbers using standard encodings.

His first extension is to define what it means to compute a function from the natural numbers to the natural numbers. Given some such function φ, Turing just computes *φ(1)*, *φ(2)*, *φ(3)*, ... on the tape, separated by 0s. At each stage a natural number is represented by that many 1s on the tape. Thus, for example, *doubling* function that mapped *x*to *2x* would be represented as:

0110111101111110111111110.....

Turing also gives an encoding in terms of the predicate calculus: if φ is such that you can prove that *φ(x)=y*, then his proof-enumerating machine 𝒦 can emit the corresponding proof, and the function is defined in this way. Turing assumes that you use some set of axioms, conjoined together to give one big axioms 𝔘φ, that contains at least *P*, the Peano axioms.

To widen the class of functions further, Turing considers the case where the variables can range over computable (real) numbers (not just natural numbers). At the moment he is stuck with computing numbers between 0 and 1 so, to widen this further he uses an old trick. The *tan* function maps values in the open interval (-π/2, +π/2) to the range (-∞, +∞). Hence, Turing's trick is to take his number γn which in the range 0 to 1, subtract 1/2 and multiply by π to get it in the right interval, and the apply *tan* to get a value over the whole range of computable real numbers.

There now follows a sequence of ten theorems (i) through (x), of which Turing will only prove (ii) and (iii) in detail. Since we assume that we can get a Turing machine to do division, all the rational numbers must be computable. However, we know that *not* all the real numbers are computable. Thus the set of computable numbers must lie somewhere between the rationals and the reals. In what follows Turing is trying to see how many of the standard mathematical concepts from the real numbers (convergence, limits etc.) also work for the computable numbers.

In summary the first six of these are:

1. The *composition* of two computable functions is computable.
2. Defining a function inductively in terms of other computable functions gives you a computable function.  
   Here Turing uses the term "recursive" in the Gödel 1931 sense (we'd call this *primitive* recursion).
3. This seems a curious idea: defining a one-argument function in terms of a two-argument function.
4. A computable *predicate* is just as computable function that returns 0 or 1.
5. One of the principal formal definitions of real numbers is in terms of a [Dedekind cut](https://en.wikipedia.org/wiki/Dedekind_cut), and here Turing is giving a version of this process for his *computable* real numbers. Basically, if we have any non-trivial predicate *G* that is true up to a point and false afterwards, then this will cut the numbers into two sets at some computable real number ξ.  
   Richard Dedekind defines the real numbers, along with natural numbers, induction, infinite sets etc. in a famous little book called *Was sind und was sollen die Zahlen?* ("What are numbers and what should they be?") published in 1888.
6. If a computable increasing continuous function crosses the x-axis between some values α and β then it must cross it at a computable point γ. Presumably we could use this (at least) to solve polynomial equations with real roots.

Just before points (vii) through (x), Turing extends the notion of the *convergence of a sequence* to computable numbers. This is similar to the usual definition, but note that the function *N* has to be a *computable* function. With this, the "computable" concept extends to power series (and this π and *e*) in the usual way.

**[pg. 257]**

The [algebraic numbers](https://en.wikipedia.org/wiki/Algebraic_number) are those real numbers that are roots of polynomials (with integer coefficients), and thus would include all the rationals but also square roots, cube roots etc. Adding the zeros of Bessel functions gives us access to the trigonometric functions (sin/cos/tan), logs etc.

Thus we can see that the set of computable numbers, which not including *all* real numbers, is nonetheless quite large.

*Proof of (ii)*

To prove that all inductively-defined functions are computable, Turing will encode them in the predicate calculus: it's not pretty, but it's not as complicated as it first looks.

We're assuming that we have some inductively-defined function η whose base case is the number *r* and whose inductive case is defined by some function φ. The axioms 𝔘φ and *P* are just whatever logic is needed to code up the auxiliary functions for φ, along with the Peano axioms.

Turing defines *H* and *K* to represent η and φ in the predicate calculus. From page 252, Turing recycles two-argument *F* as the successor function and one-argument *F* to just introduce a number. He neglects to tell us that *G(x,y)* is the grater than relation, and is true when y > x. Thus the equation from lines 7-9 on page 257 has three parts:

* The first line defines the greater-than relation (as the closure of the successor relation)
* The second line shows how H (representing η) is defined in terms of a base case and recursive case.  
  Remember that here *u* is 0, and *u(r)* is just the number *r*.
* The third line makes sure that η is a function (and not a relation): i.e. just one result for any argument.

Turing only ever uses his greater-than function in this format: "G(x,y) ∨ G(y,x)", essentially to mean that "(x < y) ∨ (y < x)", or, more simply, "x ≠ y".

The proof is broken into two parts.

* In the first part (lines 14-26) Turing needs to show that the predicate *H* actually delivers the right result, mapping any *n* to *η(n)*, or, as Turing puts it, that *H(u(n), u(η(n)))*.   
  The proof is by induction on *n*, so we assume (on line 15) that *H(u(n-1), u(η(n-1)))*. Then, since we know that *n > 0*, η(n) must have been defined in terms of φ(n, η(n-1)). Re-expressing this in terms of its encoding *K* gives Turing the

ingredients he needs to deduce the result from line 8.

* The second part of the proof starts on the last two lines of pg 257, and continues for the first four lines of pg 258. Here Turing just wants to show that it is a *function* - i.e. the logical relation *H* doesn't hold between *n* and anything other than *η(n)*, for example some other value *m*. The proof is a straightforward application of the formula on line 9.

Turing typo: on the second-last line of pg 258 we should have η(n), not η(u).

**[pg. 258]**

*Proof of a modified form of (iii)*

I find this a bit perplexing: Turing goes to a lot of trouble to prove something that seems reasonably intutive anyway. I think what's happening here is that he's taking the opportunity to follow up his discussion on the (uncomptable) function β that he discussed on page 246. Here he defines a tamer version: a function that will take a given computable sequence γ and print out the *n*-th digit of the *n*-th element of this sequence, which he calls φn(n).

Since the original sequence γ is computable we can assume that we have a machine 𝒩 for it. This machine 𝒩 operates by reading an input on the tape which is a number *n* represented by a sequence of *n* "F" symbols. It then proceeds to generate a sequence of 0s and 1s representing γn; Turing is interpreting the function φn(m) as being the *m*th digit printed.

Turing now defines a new machine 𝒩' to compute the sequence φn(n) for all *n*. The algorithm is just:

* for (h=1; h < ∞; h=h+1)
  + write the number h on the input tape as a sequence of "F"s
  + run machine 𝒩 to generate γh; when you get to the *h*th digit in this sequence stop, and copy that digit to the end.

To implement this Turing defines 𝒩 ' by modifying 𝒩. First he edits the instructions for 𝒩 to make sure it won't print any 0s or 1s itself, and to make it think the start of the tape is marked by ΘΘ instead of əə.

At any stage the letters *h* will be our counter, representing the digit we're after. You can see that he's modified the transition to state ℬ to replace one *h* by a *k* each time 𝒩 prints a 0; when there are none left he switches to state 𝔲. In state 𝔲 we print an actual 0 at the end, and then ourselves up for the next running of 𝒩. This involves first printing an extra *k*, then printing the ΘΘ pair, and then a bunch of "F" symbols (one more than the last time) to act as its input. In the process, the *k*s get changed back to *h*s, and we're ready to go again.

We assume the process is similar for printing 1s, with a set of 𝓋 states doing the work like to the 𝔲 states.

Turing typos:

* The rule for 𝔲2 is wrong: when it runs out of *k*s it should get out of the loop: thus it should go to 𝔯𝔢(𝔲3,b,k,h).
* Also the action part of the rule for state 𝔢 won't do any good; it would be better as R, Pθ, R, Pθ.  
  The Ξ doesn't seem to be used anywhere else.

Section 11: Application to the Entscheidungsproblem (pp. 259-263)

To finish off, Turing now moves on to the *Entscheidungsproblem* - i.e. the decision problem for the predicate calculus (or "functional calculus") as Turing calls it.

The overall argument is this: Gödel showed that there are undecidable propositions 𝔘 - i.e. we can't generate a proof of either 𝔘 or its negation. Turing argues that if we were able to recognise these propositions then we would have a decision procedure. This would work as follows:

* First decide if 𝔘 is one of the "undecidables"; if so, answer "don't know".
* Otherwise generate all proofs. Sooner or later a proof ending with either 𝔘 or its negation will appear: when it does, answer either "provable" or "not provable".

Thus, if a Turing machine could recognise undecidable propositions, then it could solve the *Entscheidungsproblem*.

Turing notes that the proofs that follow are a bit messy (and "somewhat lengthy") because of his encoding of integers. In what follows, I'll largely ignore the functions that make the integers work like they're supposed to: these have the form *N(x)* and *F(r).*

To show that it isn't possible to recognise the undecidables, Turing will construct a proposition of the form Un(ℳ) and argue that there is no general method that will, for any ℳ, decide if it is provable or not. The trick is to construct the formula Un(ℳ) so that it says "ℳ will print a 0", which we know to be uncomputable (from page 248).

**[pg. 260]**

This proposition is presented on the bottom-half of page 260: this is a formula in the predicate calculus that simply says "ℳ will print a 0" - i.e. it contains a description of what machine ℳ can do, and asserts that in some configuration, on some square, a 0 appears.

Turing first shows us on lines 7-9 how one instruction from ℳ can be encoded. Here *x* and *y* are the current complete configurations and square, *x'* is the next complete configuration and *y'* is the square to the left of *y*. Basically lines 7-9 say that:

* If we're reading symbol Sj on square y, and the tape-head is at square y, and the current state is qi,
* then we move the tape head (left) to square *y'*, print symbol Sk on square y and move to state ql,
* and the symbols in all other squares *z* stay the same.

Turing typo: he messed up the last bit here: instead of using Sj and Sk on the last line, he should really list a formula like this for all symbols Si and say

RSi(x,z) ⇒ RSi(x',z)

Anyway, given this encoding for *one* instruction, we can run through and encode all the instructions in a machine and then take their conjunction (not "logical sum" as Turing says): Turing calls this proposition Des(ℳ).

The formula Un(ℳ) is just a statement that

* if there exists some *u* (which we can read as the number zero) such that in the configuration numbered *u* every square *y* is blank (symbol S0), the tape head is at position 0 and the state is state q1
* then there exist some configuration *s* and square *t* such that there is a 0 (symbol S1) on *t* in configuration *s*.

Turing abbreviates the first part of this as A(ℳ) since this will be the standard set-up and starting configuration for any machine ℳ.

The use of propositions *F* and *N* in this long formula simply set up the numbers to work as expected. Actually Turing left some parts out here (e.g. asserting that successor was an injective function) and had to publish a correction later.

Having defined the formula, Turing now must prove an if-and-only-if statement to show that this problem is "as hard" as deciding if 0 appears, i.e.

* (a) If 0 appears then the proposition Un(ℳ) is provable. This is Lemma 1, from the top of pg 261 to midway down pg 262.
* (b) If the proposition Un(ℳ) is provable then 0 appears. This is Lemma 2, proved in the bottom half of pg 262.

**[pg. 261]**

**Lemma 1**: We suppose that ℳ eventually prints a 0.

Turing defines functions *r*, *i* and *q* to mimic the propositions *R*, *I* and *K*, and then prepares a formula (on lines 8-10) to assert that they do, in fact, mimic these. The last line of this formula has some misplaced brackets, but basically states that, for any *y*, either *y < n* or square *y* contains a blank.

This formula, which he calls CCn, simply asserts that his functions *r*, *i* and *q* faithfully mimic the corresponding propositions for the complete configuration numbered *n*.

Turing then forms CFn by adding in A(ℳ) as a premise: this is the formula from pg. 260 that states the instructions and initial configuration for ℳ (but not the part that ℳ eventually prints a 0). Note that CCn works for any machine, but the purpose of forming CFn is to tie the definitions of *R*, *I* and *K* to be those specific to machine ℳ.

He then shows that CFn is provable for all *n*, by induction over *n* - this really just involves plugging in the definitions.

Turing typo: he mixed up the definitions of *a, b, c, d* on line 29, but corrected them later; they should read:

* a = k(n)
* b = r(n, i(n))
* c = k(n+1)
* d = r(n+1, i(n))

Thus *a* and *b* are the (indexes of the) configuration and symbol before the instruction is executed, and *c* and *d* are the (indexes of the) configuration and symbol afterwards.

**[pg. 262]**

Now that we know we can prove each CFn, we can basically prove any (true) statement about any configuration *n* of the machine. In particular, since we are assuming that the machine ℳ eventually prints a 0, there must be some specific configuration *N* where this happens, and thus the statement of this fact must be contained in the conjunction that is CCN. Thus he can assert that CCN implies "there is a zero on square *K* of configuration *N*".

Since we can prove CFN and this implies CCN for our machine, we can them prove that ℳ eventually prints a 0.

The reminder of the proof is really just housekeeping to tie these together, and assert the existence of a suitable sequence of numbers between 0 and N' (and that they behave like numbers).

**Lemma 2:** This is the easy part - or, rather, we already did the work when we defined Un(ℳ). We assume that Un(ℳ) is provable. But then it must be true, and since it asserts that "ℳ will print a 0", this proves the lemma.

**[pg. 263]**

In the final paragraph Turing notes the general format of his Un(ℳ) by noting the order and number of the quantifiers. There is a sub-part of the theory of computation concerning the decidability of different formats of propositions, and Turing is just adding his particular undecidable proposition to this list.

Appendix: Computability and effective calculability (pp. 263-265)

I won't go through the appendix in detail. Once Turing had become aware of Church's paper, he needed to relate his machines to Church's lambda calculus, and that is the purpose of this section. Turing does this in two parts:

* He shows how to build a machine that can simulate the reduction of a formula in the lambda-calculus.
* He shows how to define a formula in the lambda-calculus that simulates the operation of a Turing machine.

With a bit of hindsight, we could see these as being the first examples of a *compiler* and a *virtual machine* respectively.

Note that Turing signs off his paper by giving his address as "The Graduate College, Princeton University". By the time this paper was published, Turing had left Cambridge and was working with Church in Princeton on his PhD thesis.