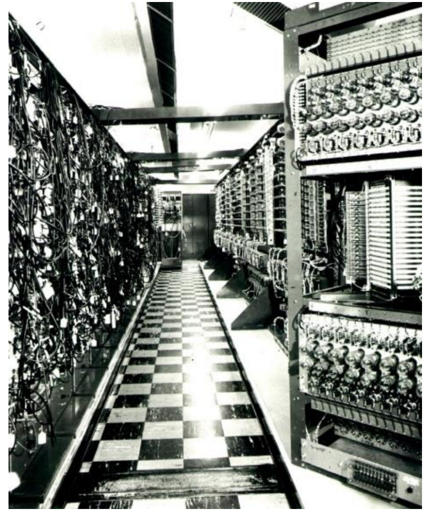
7. THE UNCOMPUTABLE

§7.1. Conceptual Models for the Computing Process



If we want to investigate in detail what computers can or cannot do, we need a precise conceptual model for the computing process. Computers, their operating systems and their programming languages can be very complex. But this complexity has to do with practicality and efficiency, not possibility. One can use a very primitive computing device and still, given enough time and patience, be able to do anything that the most advanced 'state-of-the-art' computer can do. So, in setting up a model for computability we should set up an abstract machine which is as simple as possible.

But what we must insist on, with our conceptual model computer, is unlimited memory. Those who drive powerful computers with terabytes of memory are still conscious of the limitations placed upon them by how much computer memory they have. They'd always like more. Having a fixed amount of storage places artificial

limitations on computability, even with something as straightforward as multiplying two whole numbers.

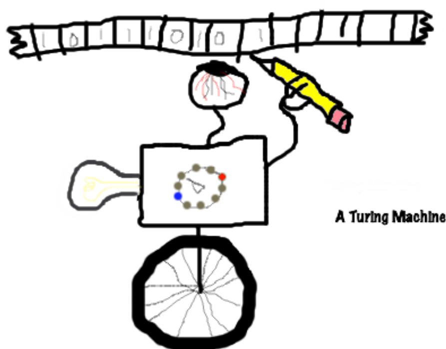
No computer in the world will ever be able to multiply any two arbitrarily large numbers. The process isn't difficult, and computers can be programmed to do this. But with limited memory, even if that limit is huge, we may not even be able to store the input, and even if we just managed to store the two numbers we may not have any memory left over to store the intermediate calculations. Yet we know, in principle, how to multiply any two numbers no matter how large they are. So we say that the multiplication function is computable. The abstract computer that's usually used to explore computability is the **Turing Machine**.

§7.2. Turing Machines

So what is a Turing Machine? Alan Turing was a mathematician who worked in Cambridge in the 1940's. He was fascinated by the concept of computability. Remember this was at a time just before the first actual computers were built. His conceptual model of a computing machine was based on the English public service. He imagined a large room filled with clerks. These clerks would make marks on a paper tape or erase marks with an eraser, according to certain instructions. Being clerks in the English public service

they were not expected to show any initiative – they had to simply follow orders. The paper tape was infinitely long (so avoiding any artificial limitations due to limited memory). And the tape came in one hatch and out another. Input was written on the tape, the tape was pulled through the hatch and when the process terminated the tape would be pushed out with the output written on it.

Dispensing with the unnecessary imagery of a room filled with public servants, we can describe a Turing Machine as having an infinitely long paper tape, ruled up into squares. A small device runs up and down this tape, capable of writing and reading marks on the tape. At each moment of time this read/write head is scanning a single square.



.he <peter@cs.uoregon.edu>

There's only one symbol that can be written onto the tape. It doesn't much matter what it is. We'll represent this mark by the symbol 1. Those squares which don't contain a 1 are said to be blank. Throughout the calculation the head writes 1's or erases them.

Now because of its invisibility, a blank is a difficult symbol to represent. For convenience we'll use the

symbol 0 to represent a blank. When the machine begins, we assume that there are only finitely many 1's on the tape, representing the input data. Sometimes we begin with the tape completely blank, represented by a two-way infinite sequence of 0's.

In addition to this infinite external memory a Turing Machine has a finite amount of internal memory. There's a gear wheel that can rotate and it can be in any one of a finite number of positions. We call these various positions the 'states' of the machine. If the machine has n states, they're labelled 0, 1, 2, ... , $n - 1$.

At any given moment the machine is in one of its states and the head is scanning one of the squares. There's a program, or set of instructions, which regulates the behaviour of the machine. Depending on the current state of the machine and the symbol being scanned, the machine writes to the square, moves either left or right one square, and the gear wheel rotates to a new state (or perhaps it stays in the same state). Then the process starts all over again.

The instructions in a Turing program are written in a table. The table has two columns, one labelled 0 and the other labelled 1. The symbol that the head is currently scanning, either a 1 or a blank, that is a 0, determines which column we take the next instruction from.

The rows of the table are labelled 0, 1, 2, ... $n - 1$ and represent the states. The current state of the machine determines the row for the next instruction. So if the machine is currently in state 3 and the head is reading a 1, the next instruction comes from the row labelled 3 and the column labelled 1.

Now what do these instructions look like? There's just one type of instruction, which is why it's so easy to learn the Turing Machine language. Suppose that the machine is in state 3, reading a 1 and that the instruction in the appropriate cell of the table is 0L5. This highly cryptic instruction says "print 0, move left and go to state 5". The symbol on the current square is erased (print 0), the read/write head moves one square left and the gear wheel rotates to position 5. One step in the calculation has just occurred.

Just two more comments are needed to complete the description of the Turing Machine – how does it start and how does it stop? The Turing machine always begins in state 0. It stops whenever it's told to go to a non-existent state. For an n-state machine, with states 0, 1, 2, ... , $n - 1$, an instruction which tells the machine to go to state n has the effect of halting the machine, indicating that the computation has been completed. What appears on the tape at this stage represents the output of the machine. So if a machine has 5 states, numbered 0, 1, 2,

3, 4 the instruction 1R5 has the effect of printing a 1, moving the head one square to the right, and then halting.

This then is the Turing Machine. It's a wonderful tool in theoretical computing science, but it only exists in the mind. Nobody has ever built such a machine. Infinitely long paper tapes are hard to come by! But since it would be highly impractical for practical purposes this is no loss. The mind is the appropriate place for it.

That's not to say that Turing Machines haven't been simulated on actual computers. It's a very easy exercise to program an actual computer to act like a Turing Machine with a very long tape, which is the nearest one can get in reality to the infinitely long tape.

You may wonder why we need an infinitely long tape if, in the course of a finite number of steps between starting and halting, only finitely many squares are visited. The reason is not that we need infinitely many squares. But we *do* need an arbitrarily large number. We may not know in advance how many squares will be visited so we have to have infinitely many to be on the safe side. We want our uncomputability results to be absolute, and not simply because we've run off the end of the tape.

Some descriptions of Turing Machines use finite tapes which can be extended if the head is about to fall off

the end. But since the real purpose of these machines is conceptual, not practical, we may as well have infinitely many squares and be done with it. After all, an infinitely long tape is no more difficult to imagine than an infinitely long line in geometry or an infinite collection of numbers in arithmetic.

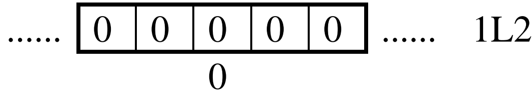
§7.3. Turing Programs

We're now ready for our first Turing Machine program. This one has 3 states and doesn't do anything particularly useful.

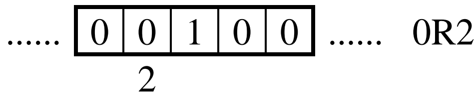
	0	**1**
0	1L2	1R1
1	1L0	0L3
2	0R2	1R1

Let's run this program on our Turing Machine. The best way to describe what happens, step by step, is to draw a picture of the tape, or at least a portion of it – as long as the portion includes all the 1's on the tape. So we can assume that everything to the left or the right of the portion that's shown, is blank. We then mark the position of the head and the state of the gear wheel by putting the number of the state underneath the square being scanned. Finally, as an extra aid to following what is going on, we put the next instruction to be performed at the right of the picture.

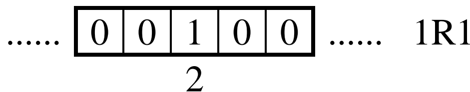
Suppose we start the machine with a blank tape. Being in state 0, reading a 0, the first instruction is 1L1.



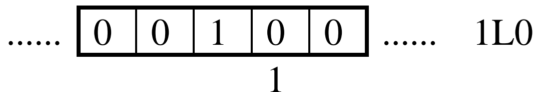
Carrying out this instruction writes a 1, moves the head one square left, and causes the machine to go into state 2. A description of the machine at the end of this machine cycle is:



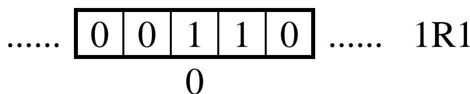
After the next step we have:



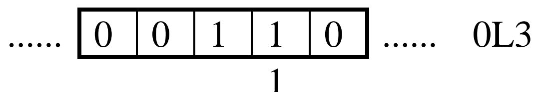
And then:



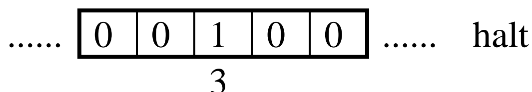
And then:



And then:



And finally:



The machine halts in state 3 with a single 1 on the tape.

Here's another Turing Machine.

	0	**1**
0	0L1	1R0
1	0R2	1L1

Suppose we start with the data 11111 on the tape, with 0's to the left and right of these five 1's, and suppose the head starts scanning the left-most 1. The first instruction to be obeyed is 1R0. This leaves the symbol 1 as it is, but moves right. The machine stays in state 0. The second 1 is encountered, and the same thing happens. We have a loop, with instruction 1R0 being performed over and over again, until the first 0 is reached to the right of all the 1's.

At this stage the instruction 0L1 is encountered. Nothing is changed on the tape, but the head moves left and the gear wheel changes to state 1. Now it is the turn of 1L1 to be performed over and over, while the head moves progressively left and the gear wheel stays in state 1. The head returns, past all five 1's until the 0 to their left is reached. The instruction now changes to 0R3. This moves the head back to where it started, and being sent to the non-existent state 3, the machine halts. The net effect is to return to exactly the same situation as existed at the beginning.

This machine hasn't resulted in any useful computation, but it has performed a little bit of mildly amusing animation, simulating a train which starts at a station, goes down the line till it reaches a blank, returns, overshoots the station, backs up and finally stops at the station.

The next machine behaves in a fundamentally different way to any machine so far.

	0	**1**
0	0L1	1R0
1	0R0	1L1

A quick examination of the instructions will show you that no matter what the initial data is, this machine

will never halt. There is simply no halting instruction in the whole table.

The fundamental problem in the Theory of Computation is to find a way of deciding whether or not a given Turing program will halt when we use certain input data. Now we did solve the problem very easily in this particular case (no halting instruction). We don't need to run the machine to see that it will never halt. But the problem is to devise a method which will work in *all* cases.

Certainly if, when you scan the instructions in a Turing program you find nowhere for it to halt, then you can say "it doesn't halt!" But the problem is that the converse doesn't work. Here's a program which provides a halting instruction in the bottom right hand corner. But, if we start it with a blank tape the blighter just ignores it!

	0	**1**
0	0R1	0R0
1	0R1	1L2

After one step, the machine finds itself in state 1, reading 0's, with the head moving continually to the right, for ever and ever.

If we started the above program with input consisting of a finite string of 1's with the head commencing on the one at the left, the behaviour of the

machine is easy to predict. It moves to the right, wiping out each 1 as it goes until it reaches a 0. By now the tape is completely blank and the machine then behaves as before, moving forever to the right. This time the only one of the four instructions not to be reached is the halting one.

§7.4. A Program For Locating a 1

Have you ever run out of petrol on an isolated road in the country and had the difficult job of deciding whether you should walk back the way you came, or walk on. You may remember how far it is back to the last town but what if the next town is just around the corner?

There's an interesting problem like this with Turing machines. Suppose you had the usual infinitely long tape, and you were told that there is a single 1 on the tape – all the other squares are blank. The machine starts somewhere, but you don't know whether the 1 is to the left or the right. The problem is to design a Turing Machine program that will locate this 1 and halt on that square.

It would be no good going left until you hit the 1 because the 1 might be to the right and this strategy would have you going left forever. Similarly it would not do to just go right. The only strategy is to alternately search to the left and to the right. Each time you move to the left

you'll need to go further than you did last time and similarly when you search to the right.

So you might go one square to the right, then back to where you started and go one square to the left. Then back to where you started and go two squares to the right. In this way you alternate between left and right searches, and each time your search goes one square further than last time. Sooner or later you will reach the 1 and you then halt.

Such a strategy is fairly obvious, but the primitive nature of the Turing Machine means that we must employ a bit of ingenuity to implement that strategy. All blank squares look the same. You would need some mechanism of counting so that you knew when you had reached the point you had reached previously in that direction so that you could go just one step further.

States can be used for counting in Turing Machines in a limited way. Each time we move to the right we could go to a new state. The problem is that every Turing Machine, by definition, has a *finite* number of states and the number of squares we might need to move might exceed this. Remember the one program has to work in all cases, no matter how far away the 1 is from our starting position.

If you decided to adopt this alternating left/right strategy on the long, straight, featureless road across the Nullarbor Plains in Australia, you might hit on the idea of marking the furthest point you have reached in each direction with a chalk mark on the road. You wouldn't need to mark the starting point, just the furthest point in each direction.

So, you move east and west alternately. When moving east you continue till you find the chalk mark, erase it, walk a certain distance further and mark the road. Then you walk west till you come to the chalk mark, erase it, walk a certain distance further west and then mark the road. This way you'll eventually reach a petrol station!

How do we adapt this to the Turing Machine problem? A chalk mark would simply be a 1 that we write on a blank square. But we have to be careful we don't confuse the 1's we are writing with the 1 we're looking for. On the Nullarbor we're not likely to confuse a chalk mark with a petrol station, but on a paper tape any 1 looks exactly like any other.

Here's a solution to the problem. Note that the beginning is a little different to the subsequent steps in that we have to put down the two 1's to begin with. Note too that if we are just about to write a 1 on a square that already has a 1 we know that we've found what we were looking for. Oh, and being tidy programmers, we go back

and erase the 1's we made and halt on the located 1, leaving the rest of the tape completely blank.

	0	1	
0	1R1	1L7	Put down left marker for the first time
1	1L2	1L8	Put down right marker
2	0L2	0L3	Move left - erase left marker
3	1R4	1R5	Put down left marker in new position
4	0R4	0R1	Move right – erase right marker
5	0R5	0L6	Tidy up right marker
6	0L6	1L7	Return to the found "1"
7	0R10		Halt on the "1"
8	0L8	0R9	Tidy up left marker
9	0R9	1L7	Return to the found "1"

§7.5. The Longest Running Turing Machines

The Turing Machine Olympics is a great occasion. Turing Machines from all round the world compete in many events. But as with any Olympics the star event is the marathon. Actually the Turing Marathon is an endurance race. Speed is a non-issue because all Turing machines run at the same speed. The gold medal goes to the one which runs longest when started with a blank tape.

Now larger Turing machines have the potential for running longer so there are marathon events for each size of machine. So the winner in the 20 state division will be

the 20 state Turing machine that runs longest when started with a blank tape.

Imagine the excitement of this great event. Countless Turing Machines are all lined up around a huge stadium. Each of them is loaded with a blank tape. The starting pistol fires and these machines spring into action. Heads fly left and right across each infinite tape as the machines operate furiously.

The machines all run at the same speed so that after a while each machine has run 1000 steps. By now many machines have halted and so are out of the race. Attention focuses on those still running. After a time there are only a few machines left, each showing no sign of tiring. Finally there is just one competitor left, and eventually he halts. Or perhaps there are several who halt at the same time – they are declared joint winners.

But what if a particular program runs forever? It's easy to write a Turing machine program that doesn't halt when started with a blank tape. Apart from tying up the stadium and preventing the next event from taking place, it isn't fair. Programs which loop have to be disqualified at the outset. Only those that will eventually halt are allowed to compete. The stewards have to examine the machines before the race begins and disqualify those that will never halt.

So for each number of states the prize goes to the Turing Machine with that number of states that for the longest number of steps *and eventually halts*, when started with a blank tape. Of course, since the number of states in a Turing program can be arbitrarily large, there are infinitely many programs competing. But there are only finitely many in each division because there's only a finite number of ways you can fill out any specific table.

The fact that there are only finitely many competitors in each race is important because if there were infinitely many the race may still never finish, even if each competitor does. Suppose there were infinitely many competitors C1, C2, C3, … and that C1 halted after one step, C2 after 2 steps and so on. Even though each competitor eventually halts there would never be a stage when they had *all* halted. But as there are only finitely many Turing machines with a given number of states this problem never arises.

§7.6. The Busy Beaver

When the problem we're about to discuss was first described it was called the Busy Beaver Problem. Beavers are industrious little animals, found in North America, who chop down small trees with their huge teeth and use the timber to construct small dams. The apparent tirelessness of the beavers has inspired such phrases as "as busy as a beaver" and the thought of Turing Machines

"beavering away" suggested the name "Busy Beaver Problem".

As we've seen, with our fanciful Turing Marathon Race, for a given number of states, *n*, there are only finitely many *n*-state machines. Of these some will never halt when started with a blank tape and so are disqualified. If the remaining ones are run, there will eventually be a winner, or at least some joint winners. There will be a certain number of steps at which these winners finally stop.

Let's call this number B(*n*). It's a function of *n* in that you need to know the value of *n* before you can work out B(*n*). It's much like the function $f(n) = n^2 + n$, except that we don't have a neat formula for it.

The Busy Beaver Function
B(*n*) is the largest number of steps that an *n*-state Turing Machine can run for, starting with a blank tape, and still halt.

The Busy Beaver Problem
The Busy Beaver Problem is to write a program that will calculate the Busy Beaver Function.

If there was a formula for B(n), even a complicated one, it would be a simple matter to write such a program. But programs can be written even when there isn't a formula. If there is *any* systematic procedure for working out B(n) in all cases, one can write such a program. In what computer language do we want such a program to be written? It doesn't matter because any such program, in any computer language, can be converted to a Turing program.

In what format do we want our answer? Do we want the B(n) to be written in normal notation, or in binary or in some other form. Binary notation is the system used for expressing numbers with just with 0's and 1's and it's the way numbers are actually stored inside a real computer.

But again it doesn't matter because it's a routine programming task to convert from one system of notation to another. Binary might seem to be one that's very suitable for Turing Machines, but don't panic if you don't know about binary notation. There's a much simpler system we can use.

§7.7. Unary Notation for Numbers

There have been many systems of notation for the numbers 1, 2, 3, ... The Babylonians had a system based on counting in 60's. The Romans had their Roman numerals. The Arabic system we use is quite efficient.

Binary is particularly suitable for computers. But the simplest system of notation is **unary**.

In unary we represent the number 7 by 1111111. This is the system prisoners use to mark off the number of days of their imprisonment. It's a system which is also used to score in various sports like cricket. Sometimes the strokes are grouped into 5's or 10's to make them easier to count, but the basic system just has 1's. What's attractive about it is the simplicity of adding 1. None of this fuss about carrying 1 that you get when you have to add 1 to 99. Just put down an extra stroke.

Of course the unary system would be totally impractical. Imagine the date 13-5-2022 expressed in unary:

1111111111111/11111/1111111111111111 .... 1.

But we're not concerned with practicalities here, just whether or not something is possible.

§7.8. Turing Programs with One or Two States

We'll calculate the value of B(1) to illustrate what would be needed in a program for calculating the Busy

Beaver Function. If a Turing Program has just one state, the initial state 0, the whole program can be written in a table with one row and two columns:

	0	**1**
0		

Each of the two cells in this table will contain an instruction. In this case there are only eight possible instructions: 0L0, 0L1, 0R0, 0R1, 1L0, 1L1, 1R0, 1R1. With eight possibilities for each cell, there are 8 × 8 = 64 possible Turing Machines. In terms of our story of the Turing Marathon, there will be 64 possible entrants in the 1-state division. But some of these will be disqualified.

Let's focus our attention on the first instruction. We can sort these 64 programs into eight groups of eight according to their first instruction.

A	**0**	**1**
0	0L0	

B	**0**	**1**
0	0L1	

C	**0**	**1**
0	0R0	

E	**0**	**1**
0	1L0	

F	**0**	**1**
0	1L1	

G	**0**	**1**
0	1R0	

Each of these partially completed tables represents eight programs corresponding to the eight different possibilities for the second instruction.

Now let’s examine these eight tables in turn. Starting with a blank tape the machines in groups A, C, E, G will loop. The second instruction will never be reached. Machines in group A, for example, move continually to the left, leaving the tape blank. Machines in group E move continually to the left, leaving behind a trail of 1’s. Machines in groups C and G exhibit similar behaviour to the right.

The remaining four groups, representing 32 programs in all, will halt at the very first step. So the longest number of steps that a 1-state program can run for, starting with a blank tape, and still halt, is 1. Thus B(1) = 1.

It’s much more difficult to calculate B(2). For a start there are 12 possible instructions now: 0L0, 0L1, 0L2, 0R0, 0R1, 0R2, 1L0, 1L1, 1L2, 1R0, 1R1, 1R2. And there are now four cells in which to put them. So there are 12 × 12 × 12 × 12 such programs in all. That's 20736 programs to consider. Even putting them into groups it’s a tedious job.

I once set this as a problem for a post-graduate course on the Theory of Computation. With the help of a computer program that they had to write, they analysed these cases and concluded that B(2) = 6. No 2-state Turing machine program will halt after more than 6 steps, but there are some 2-state programs that halt after exactly 6

steps. They are the joint winners in the 2-state division of the Turing Machine Marathon. Here is one of them.

	0	**1**
0	0L1	1L2
1	1R0	1L1

Not only were my students able to come up with 2-state programs that halted after 6 steps, they also had to show, by an analysis of all the others, that any program that was still going after six steps would go on forever.

A two-state analysis was difficult enough so it would appear that it would be very difficult to write a program that would handle the general case. Very difficult, but is it actually impossible? To say that it is would seem to limit the ingenuity of man (or woman). Yet the ingenuity of the human mind has limits – a rather humbling thought. At least we are ingenious enough to recognise our own limitations.

We can never write a program to compute the Busy Beaver Function, but at least we can prove that we can't, which is perhaps the next best thing.

The Busy Beaver Function can never be computed. It might be possible to find the values of B(3), B(4) and so on, but the methods would be forever changing. No one set of ideas can handle all B(n)'s. Why not? Read on!

§7.9. Why B(*n*) is uncomputable

The Busy-Beaver function is uncomputable. That is, there is no Turing Machine which computes B(n) for all n. Nor could one ever be found. It's a logical impossibility. What's more, the fact that no such Turing Machine can exist means that no program can ever be written in *any* computer language on *any* computer – not now, not ever.

For, if anyone is ever clever enough to do so, such a program can be converted to a Turing Machine and he or she will have created a logical impossibility. Our whole world of logical reasoning will collapse!

Our proof will be a proof by contradiction. We suppose that there *is* a Turing program **BEAVER** which computes B(n). That is, if we input the number n by writing n 1's on the tape, the output will be B(n) 1's. Both input and output will be in unary notation.

With this supposedly-existing program, together with two other programs that *do* exist, we construct another program. The two auxiliary programs are **INCREMENT** and **DOUBLE**.

INCREMENT is a 2-state program that computes the function $F(n) = n + 1$. In unary notation, this is very easy to do. We simply put down one extra 1.

INCREMENT	0	**1**
0	1 L 1	1 L 0
1	0 R 2	

The other auxiliary program is **DOUBLE**. It's a 9-state program that computes the function G(n) = 2n. It takes a string of 1's, representing the input n, and joins a second copy onto it, making a string of twice the length. This is quite tricky, because after we've copied a 1 we have to mark it in some way to avoid copying it again. We do this by temporarily changing the 1 to a 0. After the head has move across to put down the copy and comes back, it can recognise where the 1 came from. It then reinstates the 1, moves to the right and proceeds to copy the next 1.

If you have the patience it's interesting to work through this program, say with an input of 3. That is, the tape consists of 111 on an otherwise blank tape and the head begins on the left-most 1.

If you can't be bothered working through it you can just accept that such a program is possible.

DOUBLE	**0**	**1**
0	0 L 5	0 R 1
1	0 R 2	1 R 1
2	1 L 3	1 R 2
3	0 L 4	1 L 3
4	1 R 0	1 L 4

continued	**0**	**1**
5	0 R 6	1 L 5
6	0 R 9	0 R 7
7	1 L 8	1 R 7
8	0 R 9	1 L 8

We now take as many copies of **DOUBLE** as we like and build up a Turing program called **OMEGA**.

OMEGA
INCREMENT
DOUBLE
DOUBLE
....
DOUBLE
BEAVER

How many states will this program have? Well, that depends on how many copies of **DOUBLE** we're taking. Suppose we take n copies. Each copy has 9 states, so that's $9n$ states, plus 2 for **INCREMENT** plus how ever many states this mythical **BEAVER** has. Since we don't have a **BEAVER** program, we can't count them, but if **BEAVER** exists its number of states must exist. Let's suppose there are b states in **BEAVER**. So **OMEGA** with n **DOUBLE**'s will have $9n + b + 2$ states.

Now what will **OMEGA** do with a blank tape? Well, first it will add 1, to get 1, and then it will double that n times. At this point there will be 2^n 1's on the tape. If $n = 4$ we'll have doubled the 1 four times to get $2^4 = 16$.

At this stage **OMEGA** hands over control to **BEAVER**, which will take the 2^n as input and proceed to compute $B(2^n)$. So at the end of the day, starting with a blank tape, **OMEGA** will halt, leaving $B(2^n)$ 1's on the tape.

In doing so it must have run for at least $B(2^n)$ steps because it takes one step to put down each 1. Suppose it runs for s steps. Then s is at least as big as $B(2^n)$.
So $s \geq B(2^n)$

Now **OMEGA** is itself a Turing program, with $9n + b + 2$, states, and it halts. So it can't run longer than the maximum for all programs of its size. Hence s, the number of steps that **OMEGA** runs for must be less than or equal to $B(9n + b + 2)$, the maximum for programs in the same class as **OMEGA**.
So $s \leq B(9n + b + 2)$

Perhaps you need a breather at this point. We're establishing a number of inequalities which are probably more easily considered using symbols. Let's recap.

We have:
s = # steps that **OMEGA** (with n doubles) runs for.
$B(2^n)$ = number of 1's that **OMEGA** prints.
$B(9n + b + 2)$ = maximum # steps that any program as big as **OMEGA** can run for (and still halt).

These are connected by the following inequalities:

$$\mathbf{B(2^{\boldsymbol{n}}) \leq \boldsymbol{s}}$$

$$\mathbf{\boldsymbol{s} \leq B(9\boldsymbol{n} + \boldsymbol{b} + 2)}$$

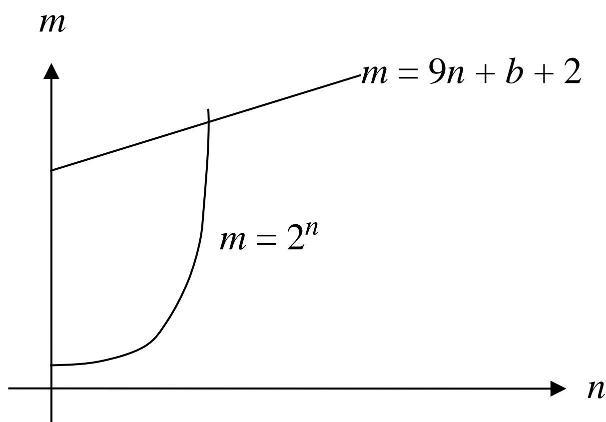
Combining these inequalities together we get:

$$\mathbf{B(2^{\boldsymbol{n}}) \leq B(9\boldsymbol{n} + \boldsymbol{b} + 2)}$$

and the final contradiction is just around the corner.

Up till now we've not been particular about the size of n. Any n would have done. But now we want n to be large. How large? Well, we want n to be large enough so that 2^n is bigger than $9n + b + 2$. Unless we had a specific value for b we could never say explicitly how large we'd need n to be. But 2^n grows *exponentially*, and no matter how large b is, eventually 2^n would exceed $9n + b + 2$.

For example if $b = 100{,}000{,}000$ a value of $n = 27$ would be large enough. The important thing is not to calculate how big n would need to be, but in recognising that no matter how big b is, there will always be a suitably large value of n.



OK, so we choose a value of n that makes 2^n bigger than $9n + b + 2$. This will mean that $B(2^n)$ is bigger than $B(9n + b + 2)$. (Remember the more steps available the longer one can make the program run.)

$$\mathbf{B(2^{n}) > B(9n + b + 2)}$$

But, and here's the contradiction, we showed above that no matter how big n is, $B(2^n) \leq B(9n + b + 2)$. But now we've shown that $B(2^n) > B(9n + b + 2)$. The only way to resolve this contradiction is to deny the only unsubstantiated assumption we've made — the existence of **BEAVER**. Therefore no such program can possibly exist and so the Busy Beaver function is uncomputable. For each n there must be a value of $B(n)$ and we may be able to find out what some of these other values are. But

a uniform, systematic procedure, that will work for *all n*, has just been proved to be impossible.

§7. 10. Busy Beaver and the Halting Problem

We've given independent proofs of the Unsolvability of the Halting Problem and the Uncomputability of the Busy Beaver function. Actually, each could have been proved from the other. If the Halting Problem did have a solution, that is if we had a program like **PREDICTOR**, we could calculate B(n) very simply, as follows:

(1). Go through the n-state programs, one by one and run **PREDICTOR**. This will tell us which machines to disqualify.
(2). Then we simulate the remaining ones, keeping a track of how long each one runs for. Since we can guarantee that these remaining candidates will all eventually halt, this procedure will terminate in a finite time.
(3). Finally we run through our record of how long each machine lasted, and take the maximum. This will be B(n).

The fact that we've shown that it's impossible to compute the Busy Beaver function shows that our assumption that we had solved the Halting Problem must be false.

Now suppose that we did have a program which could calculate the Busy Beaver function. We would then be able to solve the Halting Problem as follows:

(1). Given a program, count the number of states, n.
(2). Use **BEAVER** to compute B(n).
(3). Simulate the program for the first B(n) steps.
(4). If it halts within the first B(n) steps the answer is that the program will halt.
(5). If it hasn't yet halted by the B(n)'th step we'll know that it can never halt, because B(n) is the maximum number of steps for halting programs of that size.

The fact that we've shown that the Halting Problem is unsolvable gives us a contradiction and hence proves that **BEAVER** could not exist. So we've just shown that **BEAVER** exists if and only if **PREDICT** exists. Proving that either one cannot exist is sufficient to show that neither exists. In fact we've given independent proofs for each, so in a sense, each is doubly proved.

Not that a second proof increases the reliability of our claim. A proof is a proof is a proof. But the different methods employed in these two independent proofs are interesting and instructive.

There are many other computer programs you'd be wise not to waste time trying to write. A program that takes as input any two programs and determines whether or not they are equivalent, is such an impossibility. It would be nice to have such a program, particularly when

marking students work in computing classes. If your program is equivalent to the tutor’s then it’s correct. We could leave it to the computer to decide. Such computer marking of programs is actually used to some extent, but they are all quite limited.

No program can possibly test equivalence in all circumstances. Why? Because it has been shown that such a program, if it existed, would lead to a solution of the Halting Problem. And since there is no solution to the Halting Problem there cannot exist a solution to the Equivalence Problem.

SCIENTIFIC ARTICLE: AMITERMES LAURENSIS

[This scientific article begins as an accurate account of a species of termite but, towards the end, it becomes somewhat fanciful in order to tie in with the material of the previous chapter. The reader must decide where fact gives way to fantasy.]

Termites are found in many countries of the world, notably in Africa and Australia. The aboriginal word for termite is ‘ngartdan’.

The Amitermes Laurensis is a species of termite that builds mounds in the Northern Territory of Australia. They occur in Cape York Peninsula and eastern Arnham Land. In Queensland, north of the township of Laura (hence the name of the species), these mounds are built as thin flat plates, oriented in a north-south direction. South of Laura the mounds are conical.

Termites are sometimes referred to as “white ants” though ants and termites come from quite different insect groups. In fact termites are more closely related to cockroaches than to ants. Ants have elbowed antennae and a waist while termites have antennae like strings of beads and no waist.

However, like ants, termites are social insects and have a caste system. There are the reproductives, the workers and the soldiers. The latter two castes have neither eyes nor wings.

The mounds, which can be up to ten metres tall, are highly organised "cities" with areas for different activities. The reason for the distinctive north-south shape of the Amitermes Laurensis species is to maintain a comfortable interior temperature. In the mornings the large, flat, eastern face gets the sun while the western face remains several degrees cooler. The majority of the colony is to be found on the cooler side in the mornings. In the hottest part of the day the sun shines directly only on the northern edge, helping to keep the mound cool.

Most termites eat wood. They can hollow out large branches and this is the source of the hollow tubes from which the aborigines make didgeridoos. However the Amitermes Laurensis feed on grass and a single colony of them can process more grass than a large grazing animal. Moreover they are much more efficient in processing biomass than cows or sheep. They are probably the most efficient life-form on the planet for extracting energy from plant material. It is estimated that termites can turn a single sheet of paper into two litres of hydrogen.

The lignocellulose polymers are firstly broken down into simple sugars and hydrogen by fermentation in the termite's gut and then other bacteria transform these sugars into energy. Because of their efficiency in producing so much energy from a single kilogram of biomass they may one day help to solve the world's energy problems.

The height that a termite mound can reach is determined by the number of termites in the colony, but it is not a simple linear relationship. A colony twice as large as another would not produce a mound twice as tall. Hence there is a mathematical function, called the Busy Termite Function. T(n) is the height in millimetres of a mound that a colony of n termites can build.

Though we can determine T(n) for specific values of n, by measuring termite heights and estimating the size of the colony, it has not been possible to obtain a mathematical formula for it.

This information is of interest not only to scientists studying termites, but also to the termite colonies themselves. They need to know, for example, whether they should continue to grow as one colony or to split into two. What is surprising is that termite colonies can compute the termite function, in a crude way.

Every termite mound is, in effect, used as a computer for this purpose just as Stonehenge was a computer for making simple astronomical predictions. Indeed a termite mound could be called “Sandhenge” in view of the fact that the mound is constructed from particles of sand, held together by termite saliva. It is ironic that such silicon based computing was going on long before the invention of the silicon chip.

Each termite can hold eight grains of sand in its mandible, or bite, and this can represent a number from 0 to 255. The exact process by which the termites cooperate to perform the Busy Termite program has yet to be discovered. What we do know is that the steps, which must be genetically programmed in their DNA, cannot compute the Busy Termite function, T(n), for *all* values of n because it has been proved that this function is non-computable!