

5. THE UNCOMPUTABLE

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 megabytes 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 multiplication of numbers.

No computer in the world will ever be able to multiply any two arbitrarily large numbers. The process is not difficult, and computers can be programmed to multiply numbers thousands of digits long. But if the numbers are so big that they take up all the available memory just to store them there would be no room left over for the intermediate calculations.

Yet we know how to multiply any two numbers of any size. So we say that the multiplication function is computable.

The abstract computer that is usually used to explore computability is the Turing Machine.

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 with the concept of computability. This was at a time just before the first electronic computers were built. He took his inspiration from work done earlier in the century by Church.

A Turing Machine has 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.

There is only one symbol which can be written onto the tape. It doesn't much matter what it is. We shall represent this mark by the symbol 1. Those squares which do not 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 shall 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 that is completely blank, represented by an 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 is a gear wheel which can rotate and 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 are labelled $0, 1, 2, \dots, n - 1$.

At any given moment the machine is in one of its states and the head is scanning one of the squares. There is 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 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. Whether the head is currently scanning a 1 or a blank, that is a 0, determines which column the next instruction comes from.

The rows are labelled 0, 1, 2, ... $n - 1$ in accordance with the states. The current state of the machine determines which row the next instruction comes from. 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 is just one form for each instruction, which is why it is so easy to learn the Turing 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 tape square is erased, the read/write head moves one square left and the gear wheel rotates to position 5. One step in the calculation has 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 is 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 to the right and then halting the machine.

This then is the Turing Machine. It is 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 a 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 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 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.

Suppose we start with the data 11111 on the tape, with 0's to the left and right of these 5 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 unsolvable problem 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 in the above particular case, but the problem is to devise a method which works 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 "no can 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	0L1	1R0
1	0R0	1L1

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

If we started the above program with input consisting of a 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.

4. The Longest Running Turing Machines

The Turing Machine Olympics is a great occasion. Turing Machines from all round the world compete 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 — for the largest number of steps.

Now since Turing machines are basically identical. The only difference is the program they're running. Each competitor chooses some Turing program with which to compete. So the event is all in the strategy. The winner is the one whose program runs longest.

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

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 focusses on those still running. After a time there are a few machines left, each showing no sign of tiring. Finally they all halt simultaneously and they are declared joint winners.

Now there is not just winner, because there are numerous divisions in the event. It wouldn't be fair for a Turing program with only 2 states to have to compete with one with 1000 states. Clearly machines with more states can run longer.

Of course it wouldn't be fair for a program to compete if it runs for ever. Programs which loop are disqualified. Only those which will eventually halt are allowed to compete. So for each number of states the prize goes to the Turing Machine with that number of states which runs 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 is only a finite number of ways you can fill out any specific table. After you remove those who are disqualified you are still left with a finite number in each division.

As each race is run, the machines drop out one by one until you are left with a winner. Of course what can happen is that you get a dead heat with all the machines left in the race halting at the same step. (In fact it can be shown that this is always the case.)

5. The Busy Beaver

When the problem we are 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, using 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 have 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. But 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.

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 $8 \times 8 = 64$ possible Turing Machines with one state. In terms of our story of the Turing Marathon, there will be 64 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 come in eight groups of eight according to their first instruction.

A 0 1 0 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px;">0L0</td><td style="padding: 2px;"> </td></tr></table>	0L0		B 0 1 0 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px;">0L1</td><td style="padding: 2px;"> </td></tr></table>	0L1		C 0 1 0 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px;">0R0</td><td style="padding: 2px;"> </td></tr></table>	0R0		D 0 1 0 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px;">0R1</td><td style="padding: 2px;"> </td></tr></table>	0R1	
0L0											
0L1											
0R0											
0R1											
E 0 1 F 0 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px;">1L0</td><td style="padding: 2px;"> </td></tr></table>	1L0		0 0 1 0 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px;">1L1</td><td style="padding: 2px;"> </td></tr></table>	1L1		G 0 1 0 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px;">1R0</td><td style="padding: 2px;"> </td></tr></table>	1R0		H 0 1 0 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px;">1R1</td><td style="padding: 2px;"> </td></tr></table>	1R1	
1L0											
1L1											
1R0											
1R1											

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 is 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 $12 \times 12 \times 12 \times 12$ such programs in all. That's 20736 programs to consider. Even putting them into groups it's a tedious job because unlike the 1-state case you can't ignore any cell.

I once set this as a problem for a post-graduate course on the Theory of Computation. With the help of a computer program they had to write, they analysed these cases and concluded that $B(2) = 6$. There are 2-state programs that halt after exactly 6 steps. Here is one of them.

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

Not only were they able to come up with 2-state programs which halted after 6 steps, but they showed, by an analysis if all the others, that no program halted after six steps. Any of these programs that was still going after six steps would go forever.

Still, although it appears that it would be difficult to write a program that would handle the general case, it doesn't appear to be impossible. And yet, that is exactly what it is. The Busy Beaver function can never be programmed.

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!

8. 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 is 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 clever enough to do so he or she will have created a logical impossibility and our whole world of logical reasoning will collapse!

Our proof will be a proof by contradiction. We suppose that there exists 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.

Out of this supposedly-existing program and two other programs which do exist, we construct some other programs. The two auxiliary programs are *INCREMENT* and *DOUBLE*.

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

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

The other auxiliary program is *DOUBLE*. It is 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 is 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
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 however many states this mythical **BEAVER** has. Since we don't have a **BEAVER**, we can't count them, but if **BEAVER** exists it must be some number. Let's suppose there are b states in **BEAVER**.

So an n copy version of **OMEGA** 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 will have double the 1 four times to get 16.

At this stage **OMEGA** hands over control to **BEAVER**. **BEAVER** 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 $B(2^n)$.

Now **OMEGA** is itself a Turing program, with $9n + b + 2$, and it halts. So it cannot run longer than the maximum for all programs in its class. 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**.

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

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

These are connected by the following inequalities:

- $B(2^n) \leq s$ [**OMEGA** must run for at least as many steps as the number of 1's it prints]
- $s \leq B(9n + b + 2)$ [$B(9n + b + 2)$ is the maximum for programs in the same class as **OMEGA**]

Combining these together we get $B(2^n) \leq B(9n + b + 2)$ 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 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 the value of b , there will always be a suitable large value of n .

OK, so we choose a value of n which 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.)

But, and here's the contradiction, we showed above that no matter how big n is, $B(2^n)$ cannot exceed $B(9n + b + 2)$. On the one hand $B(2^n) \leq B(9n + b + 2)$ and on the other hand $B(2^n) > B(9n + b + 2)$, a contradiction.

The only way to resolve this contradiction is to deny the only unsubstantiated assumption we've made — 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 values are. But a uniform, systematic procedure, that will work for all n , has just been proved to be impossible.

9. Busy Beaver and the Halting Problem

We have just given independent proof 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 have shown that it is 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 can calculate the Busy Beaver function. We could now 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!
4. If it hasn't yet halted by the $B(n)$ 'th step we will know that it can never halt (because $B(n)$ is the maximum number of steps for halting programs).

The fact that we have shown that the Halting Problem is unsolvable gives us a contradiction and hence proves that **BEAVER** could not exist.

So we have just shown that **BEAVER** exists if and only if **PREDICT** exists, so that proving that either one cannot exist is sufficient to show that neither exists. In fact we have 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 which takes as input any two programs and determines whether or not they are equivalent, for example.

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 is 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. But there is no solution to the Halting Problem. Quite so. And hence there is no solution to the equivalence problem.