TURING MACHINES AND COMPUTABILITY

KAMAL LODAYA

1. Algorithms

Intuitively, we will define an algorithm as a "method" for answering any one of a class of questions. We will only look at questions about the natural numbers $N = \{1, 2, \dots\}$. We will also associate a function (over N) with the algorithm.

Question 1. *Is n a prime number?*

Algorithm 1. *Divide by $2, 3, \dots$ upto k such that $k^2 \geq n$.*

Function 1. *The algorithm computes $f : N \to \{1, 2\}$ given by:*

$$f(n) = \begin{cases} 1 & \textit{if } n \textit{ is prime} \\ 2 & \textit{otherwise} \end{cases}$$

Question 2. *What is the nth prime number?*

Algorithm 2. *The first prime number is 2. Having found the mth prime p, the $(m+1)$th can be found by testing $p+1, p+2, \dots, p!+1$ for primality.*

Function 2. *$g : N \to N$ is given by the function where $g(n)$ is the nth prime number.*

For each question in the class, an algorithm must return an *answer*, that is, it must operate in a *finite* number of steps. If we have an algorithm for a class of questions, the associated function is said to be *computable*. In 1936, Alan Turing analyzed the concept of algorithm to come up with a notion of "machines" which "compute" functions.

Turing's analysis of algorithms. A. What are "steps?" Things we do mentally? Calculations we work out on paper? Well, let us say we use paper. We write in an *alphabet*: a set of letters A. For our purposes just two letters will suffice, *blank* and *marked*, which we represent by # and 1.

B. What do we do in the "step?" We focus our attention on the paper, look at some letters and manipulate them. Let the paper be divided into *squares*, each of which may contain a letter.

C. The squares are "arranged" in some fashion. We let them be *linearly* ordered from left to right. So instead of sheets of paper, let us say we have a *tape* divided up into squares. At any moment, attention is focussed on only *one* of these squares.

D. We might use our memory too. Turing says: "the number of states of mind which need to be taken into account is *finite* ... if we admitted an infinity of states, some of them will be 'arbitrarily close' and confused." We postulate a set of states $Q = \{q_1, \dots, q_n\}$. We will also use a special state

Prepared for the *Orientation Course on Foundational Aspects of Computer Science*, 24 May–12 June 1999. Based on George Boolos and Richard Jeffrey, *Computability and logic*, Cambridge University Press.

q_0 outside Q: the machine will begin computation in the state q_1 and halt in the state q_0. Once a machine reaches q_0, it cannot do anything further.

E. The "moments" of computation are numbered $0, 1, 2, \ldots$. At any moment, the machine "situation" is a printing on the tape (a *finite* string of letters), the tape square being scanned, and the state in which the machine is. Initially, we assume the machine is scanning the first letter of the input string. If there is no input, the machine can be scanning any square.

F. From moment t to moment $t+1$, the machine chooses which letter to overwrite on the square it is scanning, shifts its attention to the square on the left or on the right, and chooses its new state. That is, we have a *transition* function $Act : Q \times A \to A \times \{L, R\} \times (Q \cup \{q_0\})$.

2. Computing

Let us look at a sample machine W_n. Its set of states is $\{q_1, q_2, \ldots, q_{n+1}\}$. Its transition function is shown in the figure on the left. Clearly, this $n+1$ state machine writes out n marks. Then it moves back to the leftmost mark.

For a more interesting machine, consider the machine S, whose transition function is described by the right hand diagram.

Next we look at a sample execution of this machine on input 11. That is, the tape initially looks like 11#####.... The machine works its way and halts with the tape 11#111#....

How does this machine compute a number-theoretic function? We feed it a tape with the unary representation of a number a, that is, a marks surrounded by blanks. Initially, the machine is scanning the first mark of a. The machine places to the right of a, separated by a blank, the (representation of) number b, and halts. We say that $S(a) = b$. On the next page, we have an execution showing $S(2) = 3$.

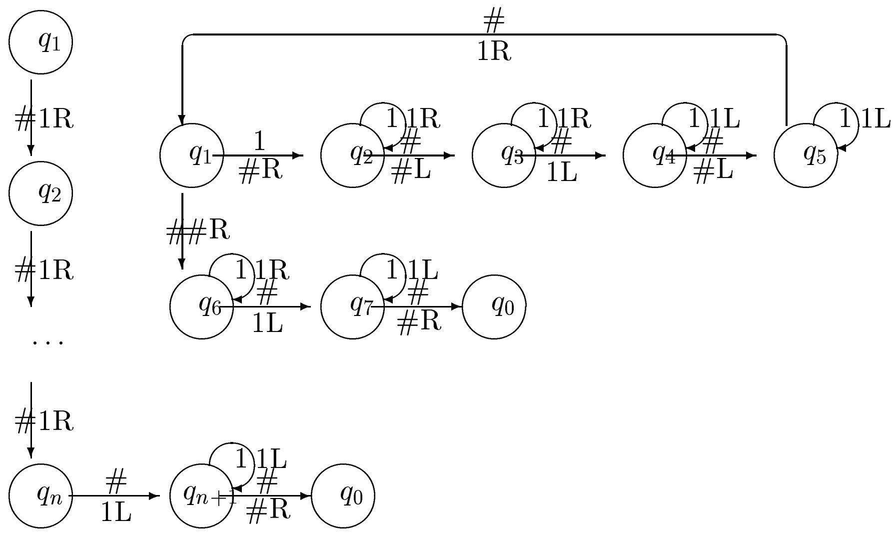

FIGURE 1. Turing machines W_n and S

Moment	Situation
0	$\dots \underbrace{1}_{q_1} 1\#\#\#\#\# \dots$
1	$\dots \#\underbrace{1}_{q_2} \#\#\#\#\# \dots$
2	$\dots \#1\underbrace{\#}_{q_2} \#\#\#\# \dots$
3	$\dots \#1\#\underbrace{\#}_{q_3} \#\#\# \dots$
4	$\dots \#1\underbrace{\#}_{q_4} 1\#\#\# \dots$
5	$\dots \#\underbrace{1}_{q_5} \#1\#\#\# \dots$
6	$\dots \underbrace{\#}_{q_5} 1\#1\#\#\# \dots$
7	$\dots 1\underbrace{1}_{q_1} \#1\#\#\# \dots$
8	$\dots 1\#\underbrace{\#}_{q_2} 1\#\#\# \dots$
9	$\dots 1\#\#\underbrace{1}_{q_3} \#\#\# \dots$
10	$\dots 1\#\#1\underbrace{\#}_{q_3} \#\# \dots$

Moment	Situation
11	$\dots 1\#\#\underbrace{1}_{q_4} 1\#\# \dots$
12	$\dots 1\#\underbrace{\#}_{q_4} 11\#\# \dots$
13	$\dots 1\underbrace{\#}_{q_5} \#11\#\# \dots$
14	$\dots 11\underbrace{\#}_{q_1} 11\#\# \dots$
15	$\dots 11\#\underbrace{1}_{q_6} 1\#\# \dots$
16	$\dots 11\#1\underbrace{1}_{q_6} \#\# \dots$
17	$\dots 11\#11\underbrace{\#}_{q_6} \# \dots$
18	$\dots 11\#1\underbrace{1}_{q_7} 1\# \dots$
19	$\dots 11\#\underbrace{1}_{q_7} 11\# \dots$
20	$\dots 11\underbrace{\#}_{q_7} 111\# \dots$
21	$\dots 11\#\underbrace{1}_{q_0} 11\# \dots$

Claim 1. *S computes the successor function $S(a) = a + 1$.*

Proof. On each iteration of the loop through q_1, the machine transfers a mark from the left number to the right number. It then adds one to the end of the copied number. Hence the output number b is $a + 1$. $\square$

Moment	Situation
0	$\dots \underbrace{1}_{q_1} 1\#\#\#\#\# \dots$
7	$\dots 1\underbrace{1}_{q_1} \#1\#\#\# \dots$
14	$\dots 11\underbrace{\#}_{q_1} 11\#\# \dots$

Notice that both S and W_n end with the machine scanning the first letter of the output number. So we can *sequence* them: by considering the halt state of W_n to be the initial state of S, we obtain a machine which will write n and then add one to it, i.e. an $n + 8$ state machine which will write $n + 1$. On the other hand, we have the $n + 2$ state machine W_{n+1} which will also write $n + 1$. Thus two Turing machines can compute the *same* function.

Now consider the machine C obtained from S using only $q_1, \dots, q_5$ of Q, and having the "final" transition $q_1 \frac{\#}{\#R} q_0$. C will simply *copy* a number from the input to the output. Suppose we add a state q to C, and replace the transition $q_3 \frac{\#}{1L} q_4$ by $q_3 \frac{\#}{1R} q$ and $q \frac{\#}{1L} q_4$. This 6 state machine D will *double* a given number. D computes the doubling function $D(a) = 2a$.

Now we can even *define* computable functions as those that are computable by Turing machines. This definition is justified by the (empirical) fact that all other definitions which people have tried out yield the same class of functions. In fact, Alonzo Church conjectured:

Thesis 1. *All algorithms for number-theoretic functions are programmable on Turing machines.*

3. Beyond computability

Now it is easy to check that the computable functions are countable. Georg Cantor showed that the set of functions from N to N is uncountable, so there must exist uncomputable functions. However, this precise definition of computability enabled Turing to *find* functions which are uncomputable. Here is a function discovered and proved uncomputable by Tibor Rado in 1962. Warning: it is quite tricky.

Think of the "productivity" of a Turing machine as the length of its output.

Question 3. *What is the productivity of the most productive n state Turing machine?*

Function 3. *$BB : N \to N$ is the maximum number an n state Turing machine can write before it reaches its halt state.*

This is clearly a well-defined function, since the number of n state Turing machines is finite. An n state Turing machine which computes $BB(n)$ (is "most productive") is called a *busy beaver.*

It is easy to see that $BB(1) = 1$ and that $BB(n+1) > BB(n)$. It has been proved that $BB(2) = 4$, $BB(3) = 6$ and $BB(4) = 13$. No further values of BB are known. However, it has been shown that

$$BB(5) \geq 4098,\ BB(6) \geq 136612,\ BB(8) \geq 3(7 \times 3^{92} - 1)/2.$$

Theorem 1. *For any Turing machine computable function T, there is an m such that $BB(m) > T(m)$.*

Proof. Suppose the Turing machine computing T has n states. Now consider the following sequence of Turing machines:

An $n + 15$ state machine writing $n + 14$ (W_{n+14});

a 6 state machine doubling its input (D);

an n state machine computing f;

a 7 state machine adding one (S).

By sequencing these together we get a $2n+28$ state machine computing $T(2n+28)+1$. Since we have demonstrated a $2n + 28$ state machine which can write out something more than $T(2n + 28)$, $2n + 28$ state machines can produce more than T can on that input. That is, $BB(2n + 28) > T(2n + 28)$. ☐

T was an arbitrary Turing machine computable function, so Rado's busy beaver function grows faster than (and differs from) all such functions. By Church's Thesis, these are *all* the computable functions. Hence BB is uncomputable.

4. Simulation

That went by a bit fast! Let us think again a little bit about why BB is not computable. Given n, how would we intuitively go about computing $BB(n)$?

There are only finitely many different graphs of n state machines. Set all the n state machines going, starting in state q_1 on a blank tape. When a machine halts, we see if it is scanning the leftmost of a string of marks. If it is not, its productivity is zero. Otherwise, we count the number of marks and keep track of the maximum reached by any machine. Can't we do all this by *simulating* all the n state machines?

To simulate a Turing machine, we have to first write it out. How do we do this?

Clearly each machine can be specified by a finite string over the alphabet $A = \{L, R, \#, 1, q_0, q_1, \dots\}$. This needs to be coded into a finite alphabet (say $\{1, 2\}$). We map the ℓ'th letter of A to 12^ℓ.

For example, here is a Turing machine: it is specified by the transitions $q_1 \frac{\#}{\#R}\, q_0$ and $q_1 \frac{1}{1L}\, q_1$. When started in a standard initial configuration, it will halt leaving its tape unchanged. So it computes the identity function $i(x) = x$. This is specified as

$$1222222\ 1222\ 1222\ 122\ 122222\ 1222222\ 12222\ 12222\ 12\ 1222222.$$

Given such a string, we can construct the Turing machine.

Working with these numbers, we can now figure out what the simulated Turing machine would do in the first step, second step and so on. So now, can't we compute $BB(n)$ by simulating all the finitely many n state Turing machines?

5. Halting

The catch is that some Turing machines may never halt. In general, Turing machines compute *partial* functions. The productivity of these non-halting machines is zero. We should not keep simulating them, but directly assign zero to their productivity. Then we would be able to compute $BB(n)$.

But that contradicts Rado's theorem. Where did we go wrong? We assumed the *halting problem* below had an algorithmic solution:

Question 4. *Given a Turing machine, does it halt when started on a blank tape?*

Function 4. $h : N \to \{1, 2\}$ *gives, when a Turing machine description is encoded as a number n, the value (say) 1 if this machine halts on a blank tape, and 2 if it does not.*

Theorem 2. *If Church's Thesis is true, the halting function h is uncomputable.*

Proof. If there was an algorithm for identifying the halting Turing machines, and therefore for the non-halters, the method described above yields an algorithm for the busy beaver function, so using Church's Thesis, BB is computable. Since this contradicts Rado's theorem, the halting function h cannot be computable. □

6. Colouring

Perhaps there is something a little unsatisfactory about the two uncomputable functions we have seen so far; both of them refer to Turing machines. Here is a function discovered and proved uncomputable by Hao Wang in 1964, which makes no mention of machines of any kind. Wang's formulation was in terms of systems of tiles, instead colours are used here.

Question 5. *Given a nonempty finite set of colours $C = \{c_1, \dots, c_k\}$ and two relations R, U on C, can one assign colours to the lattice $N \times N$ satisfying three constraints:*

- *the origin is coloured c_1,*
- *if a point coloured c_r is to the right of a point coloured c, then $c\ R\ c_r$,*
- *if a point coloured c_u is above a point coloured c, then $c\ U\ c_u$?*

Function 5. $Col : N \to \{1, 2\}$ *takes a number encoding the problem instance (C, c_1, R, U) and returns (say) 1 if this can colour the lattice and 2 if it cannot.*

Theorem 3. *The function Col is uncomputable.*

Proof. The basic idea is to bring in the halting problem, by representing an execution of a Turing machine (like the one we saw for the successor function earlier) on the lattice. The n'th row describes the situation at the $n-1$'th moment, showing the symbols on the tape and also encoding which square is being scanned and what the state of the Turing machine is. The first row describes the initial situation: the tape is blank and the machine is in the initial state. The actions of the Turing machine are encoded into the two functions R and U.

Any Turing machine can be encoded in this fashion. Now the existence of a colouring corresponds to an infinite run of the machine on a blank tape, and its non-existence to a finite run. So a computable Col would make the halting function h computable, contradicting the previous theorem. □

7. Indexing

We have shown how, to each Turing machine which deals with its input and output in a standard format, there corresponds a string over an alphabet A. Conversely, a string over A specifies some Turing machine provided it satisfies some easily characterizable properties – for example, it must be in blocks of 5, corresponding to $Q, A, A, \{L, R\}, Q \cup q_0$. Such strings are certainly countable; they can be enumerated. The first such string which encodes a Turing machine is 1222222122212221212222. So we can enumerate the Turing machines: $U_1, U_2, \ldots$ and so on. The first Turing machine U_1 computes the identity function. U is called an *indexing*.

We now use a trick called *diagonalization*, which Cantor devised for indexings. (He used it to prove that the functions from N to N are uncountable.) The *diagonal* of the indexing U is the function

$$Diag(n) = \begin{cases} 1 & \text{if } U_n(n) \text{ is defined} \\ U_n(n)+1 & \text{otherwise} \end{cases}$$

Theorem 4. *$Diag$ is not computable.*

Proof. Suppose $Diag$ were computable. Say it is U_m, the m'th in the enumeration we have fixed. Then $Diag(m) = U_m(m) = U_m(m) + 1$, which is a contradiction. □

Since U_1 is the identity function, $U_1(1) = 1$ and $Diag(1) = 2$. So $Diag$ has perfectly well-defined values, the contradiction lies in assuming that it is computable. It might seem that $Diag$ *is* computable, if we painfully work our way through the indexing U. If we were to do so, we would again come to this question: given a machine U_n and input n, will it halt with an answer or not? If *that* were routine, there would be an algorithm for $Diag$. And if Church's thesis is correct, we again come to the conclusion that the halting problem is algorithmically unsolvable.

Using indexing, we can give a diagonal proof to show that the halting function h is not Turing computable. Suppose h is computed by a Turing machine H. Build the sequence:

A 6 state machine copying its input; then the machine H; then a 4 state machine which halts iff the output of H is 1.

For all n in N, this composite machine halts if and only if the machine U_n does not. Let this composite machine be U_m in the indexing U. Set $m = n$ to see that U_m cannot exist. Thus H cannot exist: h is not Turing computable.