

## Algorithms as Machines

Kamal Lodaya  
 The Institute of Mathematical  
 Sciences , C.I.T Campus  
 Chennai 600 113, India  
 Email:kamal@imsc.res.in

To solve problems occurring in the real world, computing scientists devise algorithms. These algorithms are programmed on computers and used to solve the problems. But computing scientists also talk about ‘machines’ of various kinds, such as Turing machines, Mealy machines and von Neumann machines. (These are quite different from actual computers.) This article examines how algorithms turn into machines. The problems considered are the very elementary ones of addition, multiplication and factoring.

### 1. Computing

#### 1.1 *Back to School*

We will start with very elementary problems, the kind you learn in school.

**Instance 1.** *What is  $1536 + 12800$  ?*

In computing science, this is called an *instance* of:

**Problem 1.** *Given integers  $i, j$  (of  $n$  digits each), what is  $i + j$  ? We say the input to the problem is of size  $n$ . The solution to the problem is found by:*

**Algorithm 1.** *for each digit from right to left {  
                   do single digit addition from  
                   memory, keeping track of the carry.  
                   }*

This is what we were taught in school, without using fancy words like ‘algorithm’.

Assume that each step (single digit addition) takes you a small amount of time, bounded by a constant  $c$ . The result can be computed in time  $cn$ . We say this is an  $O(n)$  (order  $n$ ) – or linear – time algorithm.

#### Keywords

Algorithm, computing machine, computation model, Turing machine, Mealy machine, Ptime.



Addition has an  $O(n)$  (order  $n$ ) or linear time algorithm, multiplication has an  $O(n^2)$  or quadratic time algorithm.

### 1.2 A Second Example

We continue our exploration of school arithmetic.

**Instance 2.** *What is  $256 \times 256$  ?*

**Problem 2.** *Given integers  $i, j$  (of  $n$  digits each), what is  $i \times j$  ?*

**Algorithm 2.**

```

    for each digit of the multiplier {
    for each digit of the multiplicand
    from right to left {
        do single digit multiplication
        from memory,
        keeping track of the carry,
        to get a temporary result
    } };
    add up the temporary results,
    shifting in one additional zero from
    the right.
    
```

This is an  $O(n^2)$  – or quadratic – time algorithm, where single digit multiplication is to be treated as one step. More generally this is called a PTIME (polynomial time) algorithm. (Naturally, a linear time algorithm is also a PTIME algorithm.) Here is the computation for our question:

$256 \times 6 = 1536$ ,  $256 \times 50 = 12800$ ,  $256 \times 200 = 51200$ ; adding, we get 65536.

Of course, once we have laid out the computation in front of you (which took quadratic space  $O(n^2)$ ), scanning it and verifying that the answer is correct takes time linear in the length of the computation, or quadratic in the size of the inputs.

**Exercise 1.** *Given an integer  $i$  (of  $n$  digits), give a PTIME algorithm to compute  $2^i$ .*



### 1.3 High School

**Instance 3.** Find a factor of 4294967297.

**Problem 2.** Given integer  $i$  of  $n$  digits, find a (non-trivial) factor of  $i$ .

Factorization has an  $O(10^{n/2} \times n^2)$  algorithm which takes time exponential in the size of the input.

**Algorithm 3.**  $\begin{array}{l} \underline{\text{repeat}} \{ \\ \quad \text{Guess a prime number } p < i \\ \quad \text{and try division} \\ \} \underline{\text{until}} \text{ successful or all smaller} \\ \text{primes are exhausted.} \end{array}$

Is this an algorithm?

How do we guess, by tossing a coin? In that case, how many trials lead to success?

**Algorithm 4.** Divide by 2, 3, ... upto  $k$  such that  $k^2 \geq i$ .

Now this looks like a proper *deterministic*  $O(\sqrt{n} \times n^2)$ , or PTIME algorithm, with each division taking quadratic time. It is *not*. The size  $n$  is the number of digits in the input  $i$ . The algorithm takes  $O(\sqrt{i} \times n^2)$  time, which is  $O(10^{n/2} \times n^2)$ , or a deterministic algorithm which takes time exponential in the size of the input.

Tossing a coin gives a *probabilistic* or *randomized* algorithm. Making a guess and verifying whether it is right is a *nondeterministic*  $O(n^2)$  time (more generally NPTIME) algorithm.

### 1.4 A Review

Let us step back and look at the above algorithms.

The first question we looked at, must have seemed pretty easy to you. The second one was not that easy (unless you have a good knowledge of the powers of two), but when we laid out the computation in front of you, it was pretty easy. The third one is not, unless, like the Swiss mathematician Leonhard Euler in 1732, you use



Verifying that a number is a factor can be done in quadratic time.

some number theory to find out that the smallest factor of 4294967297 is 641.

However, without using any number theory, by programming Algorithm 4 on a computer using any modern programming language, you can indeed solve Problem 3. But once we have told you the solution for an instance of the problem, verifying that it is indeed a factor is again not that difficult; you can do it in quadratic time.

## 2. Computing Science

Computing science began with modelling computation. We will do a quick recap of the history.

### 2.1 *The Late 19th Century*

The first idea was to model numbers and computations on them in *logic*. Richard Dedekind in Germany and Giuseppe Peano in Italy were two 19th century mathematicians who tried to write down a set of *axioms* from which everything about numbers could be computed. These followed the tradition of the axioms written by the Greek geometer Euclid around 300 BCE, who tried to capture everything about geometry. In the 19th century it was found that Euclid's axioms were not sufficient, and the German mathematician David Hilbert came up with an updated set of axioms.

### 2.2 *The Early 20th Century*

The British logicians Bertrand Russell and Alfred North Whitehead were ambitious enough to write a book called *Principles of Mathematics*, in which they suggested how all of mathematics could be reduced to its *logical principles* (or axioms). Hilbert and his student Wilhelm Ackermann wrote their book *Principles of Theoretical Logic* in 1928 in which they subjected these logical principles to great scrutiny, so that all mathematicians could be convinced that they were sound.



Within a few years, in 1930 and 1931, the Austrian logician Kurt Gödel put the brakes on Hilbert's programme by giving a proof (using just the axioms that Hilbert approved of) that it was impossible to list the theorems of mathematics about positive integers in the way Hilbert desired. The three questions that we saw above are instances of these kinds of theorems. The phrase 'impossible to list' can, in today's jargon, be written as 'there is no algorithm'.

For each instance of a problem, an algorithm must return an answer in a finite number of steps.

Two mathematicians, Emil Post, an American, and Alan Turing, a Britisher, decided to abandon the grand goal of capturing "all of mathematics" and more modestly tried to describe "all the algorithms" [1]. There may be no algorithm for all the theorems of mathematics, but at least there are algorithms for addition and multiplication and factorization, as we learn in school. Below we will try to arrive at their models (devised in the 1930s); however, we will try and handle the easier computations before we come to the harder ones.

### 2.3 Modelling Computation Circa 1936

For each instance of a problem, an algorithm must return an answer, that is, it must operate in a *finite* number of steps. Post and Turing defined 'machines' which follow the computation of an algorithm. We could go straight to these definitions, but it is interesting to see how Turing analyzed what a mathematician (for whom he used the word "computer") might do to solve a problem. Remember that when Turing's and Post's papers appeared in 1936, there were no computers like we have today!

**A.** What are 'steps'? Things the computing mathematician does mentally? Calculations worked out on paper? Well, let us say the computer uses paper. Then it must read and write in an *alphabet*: a finite set of letters  $\Sigma$  for input and another finite set  $\Delta$  for output. Both of



Turing argued that the number of states of mind which need to be taken into account is finite, otherwise some of them would be arbitrarily close and confused.

them could overlap or even be the same. For example, in doing the computations above we used the digits 0 to 9, and other signs like  $+$ ,  $\times$ ,  $=$  and blanks. Once the input and output alphabets are fixed, we can state what the problem to be solved is.

**B.** What does the computer do in the ‘step’? Focuses its attention on the paper, looks at some letters and manipulates them. Let the paper be divided into *squares*, each of which may contain a letter. The squares are ‘arranged’ in some fashion. Let them be ordered from left to right. So instead of sheets of paper, let us say the computer has one or more *tapes* divided up into squares. At any moment, attention is focussed on only *one* position on these tapes.

**C.** The computing mathematician might use memory too. Turing argued: “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.” He postulated a finite set of states  $S = \{s_1, \dots, s_n\}$ . His computing machine would begin computation in the initial state  $s_1$ .

**D.** Computation proceeds linearly through ‘moments’ 1, 2, .... From one moment to another, a machine with one input tape and one output tape does a ‘step’ which is described by a transition function  $f : (S \times \Sigma) \rightarrow (\Delta \times S)$ , that is, a function that maps a (state, input symbol) pair into a (state, output symbol) pair.

**E.** What does the computer do overall? It takes a string of input symbols over the input alphabet  $\Sigma$  and produces a string of symbols over the output alphabet  $\Delta$ . We say that the function computed is from  $\Sigma^*$  to  $\Delta^*$  (which denote the strings over the alphabets).

You may be a little confused that we are using ‘computing mathematician’, ‘computer’, ‘algorithm’ and ‘machine’ to mean the same thing. But this was Turing’s



brilliant idea! He realized that an algorithm for a problem is a sequence of mechanical steps which could very well be done by a machine. The American scientist John von Neumann realized this and built a physical machine which could do these steps mechanically, and this is how computers came to be physical machines rather than mathematical machines or mathematicians.

### 2.4 Mealy Machines

To begin with, we will consider the easier computations. For instance, to solve Instance 1 ( $1536 + 12800$ ), we are given the numbers on two tracks of a tape. Our computing machine (devised by George Mealy in 1955) has to produce its output on a second (answer) tape. In the first step, the machine reads the last digits 6 and 0 and outputs 6 on the answer tape. Then it reads 3 and 0 and outputs 3, reads 5 and 8 and outputs 3 and remembers that there is a carry . . . and so it will finally output 14336. In its memory, it will remember some temporary information, such as what the ‘carry’ is.

Fixing the set of states and the transition function turns the machine into an algorithm. When the input strings come to an end, the string produced on the output tape is the answer. That is, the transition function is lifted to a function from strings over the alphabet  $A$  to an answer string over the alphabet  $B$ .

Formally, a Mealy machine (with one input and one output tape)  $M = (S, \Sigma, \Delta, \delta, s_1)$  consists of:

- $S$  is a finite set of *states* ;
- $s_1 \in S$  is the *initial* state ;
- $\Sigma$  is a finite *input alphabet* ;
- $\Delta$  is a finite *output alphabet* (could be the same as  $\Sigma$ ) ;

Turing’s brilliant idea was that ‘computing mathematician’, ‘computer’, ‘algorithm’ and ‘machine’ mean the same thing as far as solving a problem is concerned.



Mealy machines can be used to do addition, but not to do multiplication.

- $\delta : S \times \Sigma \rightarrow \Delta \times S$  is the *transition function*.

A *run* of such a machine begins with the machine in state  $s_1$  at one end of the input string  $w$  and with the output tape blank. In every step, the machine moves one letter through the input string, writes one letter on the output tape and moves forward, and moves to a new state as dictated by the transition function. When the machine has read the last letter of the input and written the corresponding letter on the output, the string on the output is the answer. Thus, a Mealy machine computes a function  $f_M$  from  $\Sigma^*$  to  $\Delta^*$ , given by  $f_M(a_1 \dots a_n) = b_1 \dots b_n$ . For example, a Mealy machine for addition would have as  $\Sigma$  pairs of digits and as  $\Delta$  single digits, and would compute  $f_M(\begin{smallmatrix} 0 & 1 & 5 & 3 & 6 \\ 1 & 2 & 8 & 0 & 0 \end{smallmatrix}) = 14336$ .

**Exercise 2.** *Work out the details of how a Mealy machine with a two-track input tape can add any two numbers, and output the answer.*

### 2.5 The Limited Power of Mealy Machines

It is easy to see that Mealy machines cannot be used to do multiplication. When doing a multiplication computation, we need to write down the intermediate results where a number is multiplied by a single digit, and then add the intermediate results up. Of course, the machine could ‘remember’ the intermediate results by using its states, but since the number of states is fixed, the machine will not be able to do this for arbitrarily large inputs. The machine could write the intermediate results on its output tapes, but output tapes cannot be used as inputs.

### 2.6 Turing Machines

Turing devised a simple solution to this problem: His machine (1936) has a *single* tape, with a single *working* alphabet  $\Gamma \supset \Sigma$ , on which the inputs are given (separated by a *blank* symbol  $\square \in \Gamma \setminus \Sigma$ ), but the machine



can write as well as read on this tape, and it can move left and right on the tape (for instance, to find some new space to write intermediate results). Thus the tape is used for input, output, as well as an unbounded memory beyond the fixed number of states.

The transition function now will have the form  $(S \times \Gamma) \rightarrow (\Gamma \times S \times \{\text{moveleft}, \text{moveright}\})$ . That is, the machine in a state  $s$  reads the letter on the current square of tape it is positioned at, then goes to a new (or possibly the same) state, writes a new (or possibly the same) letter on the square and moves either left or right. Since the machine can go back and forth any number of times, Turing added a *halt* state  $s_0$  to his machine, after reaching which the machine could go no further.

Formally, a Turing machine is defined as  $M = (S, \Sigma, \Gamma, \delta, s_1, s_0)$  where

- $S$  is a finite set of *states*,  $s_1 \in S$  is the *initial* state,  $s_0 \in S$  is the *halt* state ;
- $\Sigma$  and  $\Gamma$  are alphabets as above ;
- $\delta : ((S \setminus \{s_0\}) \times \Gamma) \rightarrow (\Gamma \times S \times \{\text{moveleft}, \text{moveright}\})$  is the *transition function* .

A *run* of such a machine begins with the machine in state  $s_1$  at one end of the input string  $w$ . In every step, the machine moves one square on the tape, rewriting it (writing back the same letter means it is not changed), shifts to a new state and moves left or right one square, as dictated by the transition function. When the machine has reached the halt state  $s_0$ , the string left on the tape is the answer. Thus, a Turing machine computes a function  $f_M$  from  $\Gamma^*$  to  $\Gamma^*$ , given by  $f_M(a_1 \dots a_n) = b_1 \dots b_m$ .

Since the input is from the alphabet  $\Sigma$ , we usually require the output string to also be in the same alphabet  $\Sigma$  and a convention can be devised for ignoring the other letters.

Turing's solution was to have a single tape which can be used for input, output, as well as unbounded memory.



A Turing machine could start with the input instance  $256 \times 266$  written as a string on the tape, and keep rewriting these symbols to get the answer.

### 2.7 *Running a Machine*

For example, a Turing machine which is supposed to work on the multiplication problem could start with the input instance  $256 \times 256$  written as a string on the tape. After one single digit multiplication, the tape could look like

$$256 \times 25 \cancel{6} \rightarrow 1536.$$

Notice that the last digit of the second 256 has been cancelled out. Formally this means replacing a symbol like 6 by a new symbol  $\cancel{6}$ . After another single digit multiplication, the tape might look like

$$256 \times 2 \cancel{5} \cancel{6} \rightarrow 1536 + 12800.$$

Another digit has been cancelled out. After the third, the tape could be cleaned up to leave

$$\rightarrow 1536 + 12800 + 51200.$$

And the addition could now proceed to yield intermediate configurations as shown below. (This time the result is built up on the left.)

$$\begin{aligned} 6 &\rightarrow 153 \cancel{6} + 1280 \cancel{0} + 5120 \cancel{0} \\ 36 &\rightarrow 15 \cancel{3} \cancel{6} + 128 \cancel{0} \cancel{0} + 512 \cancel{0} \cancel{0} \\ 536 &\rightarrow 1 \cancel{5} \cancel{3} \cancel{6} + 12 \cancel{8} \cancel{0} \cancel{0} + 51 \cancel{2} \cancel{0} \cancel{0} \\ 5536 &\rightarrow \cancel{1} \cancel{5} \cancel{3} \cancel{6} + 1 \cancel{2} \cancel{8} \cancel{0} \cancel{0} + 5 \cancel{1} \cancel{2} \cancel{0} \cancel{0} \\ &65536 \end{aligned}$$

Again the Turing machine ‘cancels out’ digits as it goes along. Once the calculation is done, it replaces all the intermediate symbols by blanks. Getting the answer, the machine can proceed to its halt state.

What we showed above was the computation of the Turing machine for *one* instance of the problem. But from the states and transition function of the Turing machine we can also figure out what it will do on *any* instance of the problem. If we are convinced that what the Turing



machine is doing is correct, then the Turing machine is a specification of an algorithm which will solve *all* instances of the problem. Since the number of states, the alphabet – and hence the transition function – are all finite, this is a finite specification. That is, a Turing machine is one way of presenting an algorithm.

**Exercise 3.** *Work out the details of how a Turing machine can multiply any two numbers, and output the answer.*

### 3. The Power of Models

If you even tried doing the last exercise, you must be wondering why on earth Turing devised such tedious models as his machines for the purpose of representing algorithms. Programming something as simple as multiplication on a Turing machine is a real chore! Instead, writing such a program in a language like FORTRAN or C is a breeze. The compiler of such a language translates programs to code for the ‘real’ hardware of a computer, and computability theorists have shown that we could also translate these programs to Turing machines if we wanted to. Why not switch to using programming languages as models rather than the cumbersome machines?

Since machines are so simple, it is possible to give a formal proof that Mealy machines cannot do multiplication. Programming languages are good for practice, but machine models are good for theory. Clearly Turing machines can do anything Mealy machines can do, so we say they are a more *powerful* model.

But they are not all-powerful. The reason Turing defined his machines was to show that there *are* problems they *cannot* solve and hence, using the computability theory, there are problems that programs written in any programming language cannot solve. Such problems are said to be *undecidable* and *Resonance* has carried articles about them earlier.

A Turing machine is one way of presenting an algorithm, but it is a real chore! Writing a program in a language like Fortran or C is a breeze.



A Ptime algorithm can now be defined as a Turing machine ... We can then show that there are problems which Ptime machines cannot solve, but which can be solved by a machine which takes exponential time.

For multiplication, we were able to give a PTIME algorithm. This can now be *defined* as a Turing machine for which there is a polynomial  $p(n)$  in one variable such that, for any instance of the problem of size  $n$ , the number of steps taken before it halts with the solution is at most  $p(n)$ . We can then show that there are problems which PTIME machines cannot solve, but which can be solved by a machine which takes exponential time.

For our third problem, factoring, nobody has found a PTIME algorithm. We were only able to give an exponential time algorithm. But for this particular problem, we do not know if it is one which PTIME machines can solve. Either it is, and computing scientists have not been ingenious enough to find such an algorithm; or it isn't, and computing scientists have not been ingenious enough to prove that!

What computing scientists did do was to define different kinds of machines, those which employ guesses (*nondeterministic* machines) or probability (*randomized* machines). As we saw earlier, an NPTIME algorithm – one which takes polynomially many steps on a nondeterministic machine – is known for factoring. One of the breakthroughs of the last decade was Peter Shor's algorithm of 1994, which takes polynomial time for factoring on a machine equipped with special *quantum probabilistic* operations—we might call it a QPTIME algorithm. This is better than NPTIME because we do not know any way of building a machine which 'guesses correctly', but we can hope that with better knowledge of physics, the quantum machines can be built in the future.

A special case of factoring is *primality testing*. This is computing a function which, given an integer, returns an answer *prime* or *composite*. We are given a number, and if its smallest factor ( $\geq 2$ ) is the number itself, we know that the number is a prime; otherwise we return the answer *composite*. Notice that this is easier than



factoring, in the sense that if we have an algorithm for factoring we can easily modify it to do primality testing, but if we have an algorithm for primality testing it is not clear how to use it to quickly find a factor when the given number happens to be composite. In mathematical jargon, we say primality testing *reduces* to factoring.

A PTIME algorithm for the primality testing problem was found by Manindra Agrawal, Neeraj Kayal and Nitin Saxena of IIT Kanpur in 2002. *Resonance* has carried an article about this earlier [2]. So now we know that not only addition and multiplication but also primality testing can be done in PTIME. Is there an algorithm waiting to be found which will show that factoring falls into the same class of problems? The opinion among computing scientists is divided.

One of the breakthroughs of the last decade was Peter Shor's algorithm, which takes polynomial time for factoring on a machine equipped with special quantum probabilistic operations.

### Suggested Reading

- [1] *Resonance*, Vol.2, No.7, 1997.
- [2] N Kayal and N Saxena, *Resonance*, Vol.7, No.11, pp.77–79, 2002.

### Erratum

*Resonance*, Vol.14, No.3, p.308, March 2009

Please note the correct dates for the course given below:

#### **Refresher Course in Experimental Chemistry**

**June 15–29, 2009**

at School of Chemistry, University of Hyderabad,  
Hyderabad 500 046

See website for further details:

<http://www.ias.ac.in/resonance/March2009/p307-309.pdf>

