Turing Machines

Computation has been around a very long time. After all, computer programs are a rather recent invention. So we shall take what seems like a brief detour and examine another system (or model) of computation. We shall travel back a little in history to do this. Not back too far though, merely to the mid 1930's. (After all, one could go back to Aristotle, who was possibly the first person to develop formal computational systems and write about them.)

Well before the advent of modern computing machinery, a World War II codebreaker and logician named A. M. Turing developed the system we shall examine. In the 1930's (before the construction of the first electrical computer), he was joined by several mathematicians (including Church, Markov, Post, and Turing) who also considered the problem of specifying a system in which computation could be defined and studied.

Turing focused upon human computation and thought about the way that people compute things by hand. After this examination of human computation, he designed a system in which computation could be expressed. He claimed that any nontrivial computation required:

- a simple sequence of computing instructions,
- scratch paper,
- an implement for writing and erasing,
- a reading device, and
- the ability to remember which instruction is being carried out.

Turing then developed a mathematical description of a device possessing all of the above attributes. We would recognize the device that he defined as a special purpose computer. In his honor it has been named the *Turing machine*.

This heart of this machine is a *finite control box* which is wired to execute a specific *list of instructions* and thus is precisely a special purpose computer or computer chip. The device records information on a *scratch tape* during computation and has a *two-way head* that *reads* and *writes* on the tape as it moves along. Such a machine might look like that pictured in figure 1.

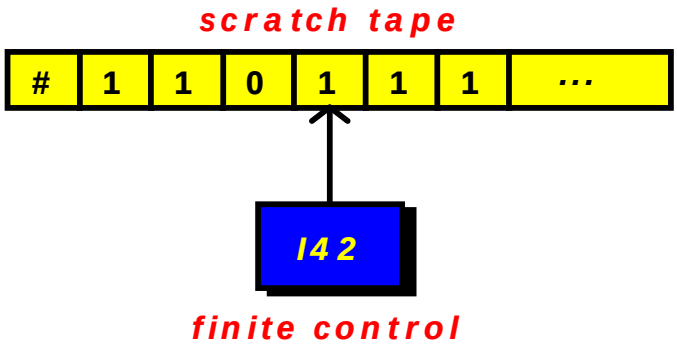


Figure 1 - A Turing Machine

A *finite control* is a simple memory device that remembers which *instruction* should be executed next. The *tape*, divided into *squares* (each of which may hold a *symbol*), is provided so that the machine may record results and refer to them during the computation. In order to have enough space to perform computation, we shall say that the tape is *arbitrarily long*. By this we mean that a machine never runs out of tape or reaches the right end of its tape. This does *NOT* mean that the tape is infinite - just long enough to do what is needed. A *tape head* that can *move* to the left and right as well as *read* and *write* connects the finite control with the tape.

If we again examine figure 1, it is evident that the machine is about to execute instruction *I42* and is reading a 1 from the tape square that is fifth from the left end of the tape. Note that we only show the portion of the tape that contains non-blank symbols and use three dots (. . .) at the right end of our tapes to indicate that the remainder is blank.

That is fine. But, what runs the machine? What exactly are these instructions which govern its every move? A *Turing machine instruction* commands the machine to perform several steps, namely:

a) to *read* the tape square under the tape head,
b) *write* a symbol on the tape in that square,
c) *move* its tape head to the left or right, and
d) proceed to a *new instruction*

depending upon what symbol appeared upon the tape square being scanned before the instruction was executed.

An instruction shall be described in a chart that enumerates all of the possible combinations of the four actions outlined above. Here is an example of an instruction for a machine which uses the symbols 0, 1, #, and blank.

	symbol read	*symbol written*	*head move*	*next instruction*
I93	**0**	**1**	**left**	**next**
	1	**1**	**right**	**I17**
	b	**0**	**halt**	
	#	**#**	**right**	**same**

This instruction (*I93*) directs a machine to perform the actions described in the table below according to the tape symbol being read. In this description we merely wrote out things in a bit more detail.

Read	Action
0	print a 1, move one tape square left, and go to the next instruction *(I94)*
1	print a 1, move right one square, and go to instruction *I17*
blank	print a 0 and stop immediately
#	print #, move to the right, and execute this instruction once more

Now that we know about instructions, we need some conventions concerning machine operation. *Input strings* are written on the tape prior to computation and will always consist of the symbols 0, 1, and blank. Thus we may speak of inputs as binary numbers when we wish. This may seem arbitrary, and it is. But the reason for this is so that we can describe Turing machines more easily later on. (Besides, we shall discuss other input symbol alphabets in a later section.) When several binary numbers are given to a machine they will be separated by blanks (denoted as *b).* A sharp sign (#) always marks the left end of the tape at the beginning of a computation. Usually a machine is never allowed to change this marker. (This is so that it can always tell when it is at the left end of its tape and thus not fall off the tape unless it wishes to do so.) Here is an input tape with the triple <5, 14, 22> written upon it.

#	1	0	1		1	1	1	0		1	0	1	1	0	···

When we depict this tape as a string we will write: #101*b*1110*b*10110 and omit the blank fill on the right.

Turing machines are designed by setting up sequences of instructions. So, let us design and examine an entire machine. The sequence of instructions in figure 2 describes a Turing machine that receives a binary number as input, adds one to

it and then halts. Our strategy will be to begin at the lowest order bit (on the right end of the tape) and travel left changing ones to zeros until we reach a zero. This is then changed into a one.

One small problem arises. If the endmarker (#) is reached before a zero, then we have an input of the form 111...11 (the number $2^n - 1$) and must change it to 1000...00 (or 2^n).

sweep right to end of input

	read	*write*	*move*	*goto*
I1	**0**	**0**	**right**	**same**
	1	**1**	**right**	**same**
	#	**#**	**right**	**same**
	b	**b**	**left**	**next**

change 1's to 0's on left sweep, then change 0 to 1

I2	**0**	**1**	**halt**	
	1	**0**	**left**	**same**
	#	**#**	**right**	**next**

input = 11...1, so sweep right printing 1000...0 (print leading 1, add 0 to end)

I3	**0**	**1**	**right**	**next**

I4	**0**	**0**	**right**	**same**
	b	**0**	**halt**	

Figure 2 - Successor Machine

In order to understand this computational process better, let us examine (or trace) a computation carried out by this Turing machine. First, we provide it with the input 1011 on its tape, place its head on the left endmarker (the #), and finally turn it loose.

Have a peek at figure 3. It is a sequence of snapshots of the machine in action. One should note that in the last snapshot (step 9) the machine is *not* about to execute an instruction. This is because it has halted.

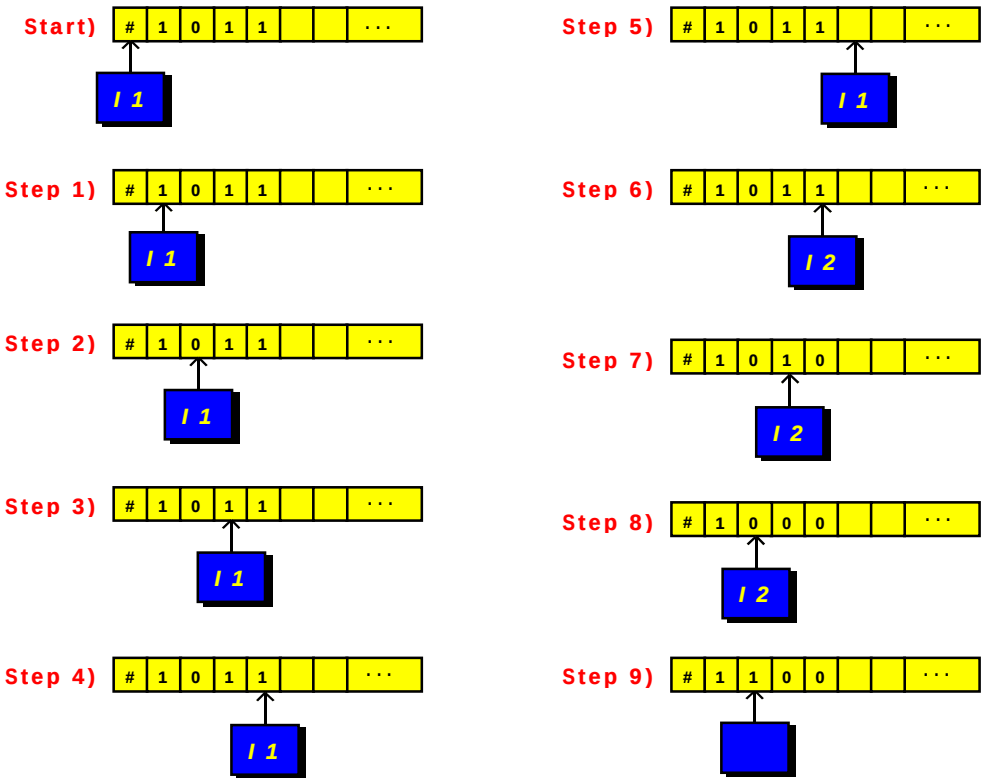


Figure 3 - Turing Machine Computation

So, now we have seen a Turing machine in action. Let's note some of the properties of these machines.

a) There are no space or time constraints.
b) They may use numbers (or strings) of any size.
c) Their operation is quite simple - they read, write, and move.

In fact, Turing machines are *merely programs written in a very simple language.* Everything is finite and in general rather simple. There is not too much to learn if we wish to use them as a computing device. (Well, maybe we should wait a bit before believing that!)

For a moment let's return to the previous machine and discuss its efficiency. If it receives an input consisting only of ones (for example: 111111111), it must:

1) Go to the right end of the input,
2) Return to the left end marker, and
3) Go back to the right end of the input.

This means that *it runs for a number of steps more than three times the length of its input.* While one might complain that this is pretty slow, the machine

does do the job! One might ask if a more efficient machine could be designed to accomplish this task? Try it as an amusing exercise.

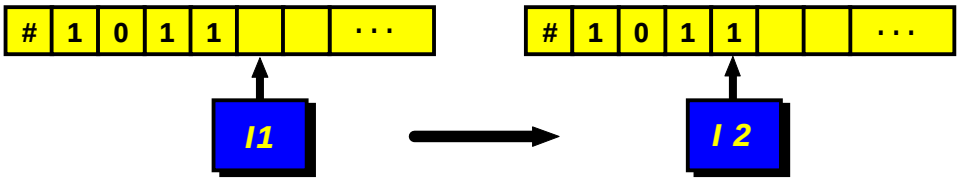
Another thing we should note is that when we present it with a blank tape it runs for a few steps and gets stuck on instruction I3 where no action is indicated for the configuration it is in (reading a blank). Thus it cannot figure out what to do. We say that this in an *undefined computation* and we shall examine situations such as this carefully later.

Our discussion of Turing machines has been quite intuitive and informal so far. This is fine, but if we wish to study them as a model of computation (and perhaps even prove a few theorems about them) we must be a bit more precise and specify exactly what it is we are discussing. Let us begin.

A *Turing machine instruction* (or just an *instruction*) is a box containing *read-write-move-next* quadruples. A *labeled instruction* is an instruction with a label (such as *I46*) attached to it. Now for the machine.

> **Definition.** *A **Turing machine** is a finite sequence of labeled instructions with labels numbered sequentially from I1.*

Now we know precisely what Turing machines are. But we have yet to define what they do. Let's begin with pictures and then describe them in our definitions. Steps five and six of our previous computational example (figure 3) were the *machine configurations:*



If we translate this picture into a string, we can discuss what is happening in writing - which we must if we wish to define what Turing machines accomplish. So, place the instruction to be executed next to the symbol being read and we have an encoding of this change of configurations that looks like:

$$\#1011(I1)b... \rightarrow \#101(I2)1b...$$

This provides the same information as the picture. It is almost as if we took a snapshot of the machine during its computation. If we omit trailing blanks from the description, then we have

$$\#1011(I1) \rightarrow \#101(I2)1$$

From now on we shall assume that there is an arbitrarily long sequence of blanks to the right of any Turing machine configuration.

> **Definition.** *A **Turing machine configuration** is a string of the form x(In)y or x where n is an integer and both x and y are strings of symbols used by the machine.*

So far, so good. Now we need to describe how a machine goes from one configuration to another. This is done (as we all know) by applying the instruction mentioned in a configuration to that configuration thus producing another configuration. An example should clear up any problems with the above verbosity. Consider the instruction:

I17

0	**1**	**right**	**next**
1	**b**	**right**	**I3**
b	**1**	**left**	**same**
#	**#**	**halt**	

and observe how it transforms the following configurations.

a) #1101(I17)01 → #11011(I18)1
b) #110(I17)101 → #110b(I3)01
c) #110100(I17) → #11010(I17)01
d) (I17)#110100 → #110100

(Especially note what took place in (c) and (d). Case (c) finds the machine at the beginning of the blank fill at the right end of the tape. So, it jots down a 1 and moves to the left. In (d) the machine read the endmarker and halted. This is why the instruction disappeared from the configuration.)

> **Definition.** *If C_i and C_j are Turing machine configurations then C_i **yields** C_j (written $C_i \rightarrow C_j$) if and only if C_j results from applying the instruction in C_i to C_i .*

In order to be able to discuss a sequence of computational steps (or an entire computation) at once, we need additional notation.

> **Definition.** *If C_i and C_j are Turing machine configurations then C_i **eventually yields** C_j (written $C_i \Rightarrow C_j$) if and only if there is a finite sequence of configurations $C_1, C_2, \ldots, C_k$ such that:*

$$C_i = C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_k = C_j .$$

At the moment we should be fairly comfortable with Turing machines and their operation. The concept of a machine going through a sequence of configurations should also be in place in our computational definition set. Let us turn to something quite different. What about configurations which do not yield other configurations? They deserve our attention also. These are called *terminal configurations* because they terminate a computation). For example, given the instruction:

I3	0	1	halt	
	1	b	right	next
	#	#	left	same

what happens when the machine gets into the following configurations?

a) (I3)#01101
b) #1001(I3)b10
c) #100110(I3)
d) #101011

Nothing happens - right? If we examine the configurations and the instruction we find that the machine cannot continue for the following reasons (one for each configuration).

a) The machine moves left and falls off of the tape.
b) The machine does not know what to do.
c) Same thing. A trailing blank is being scanned.
d) Our machine has halted.

Thus none of the configurations lead to others. Furthermore, any computation (or sequence of configurations) which contains configurations like them must terminate immediately upon reaching any of these kinds of configurations.

(By the way, configuration (d) is a favored configuration called a *halting configuration* because it was reached when the machine wished to halt. For example, if our machine was in the configuration #10(*I3*)0011 then the next configuration would be #101011 and no other configuration could follow. These halting configurations will pop up later and be of very great interest to us.)

From now on we shall name individual machines so that we know exactly which machine we are discussing at any time. We will often refer to them as M_1, M_2, M_3, or even M_i or M_j. We shall use the notation $M_i(x)$ to mean that Turing machine M_i has been presented with x as its input. We shall also use the name of a machine for the function it computes. If x is a binary number, z is a string of symbols used by the Turing machine M_i and (I1)#x eventually yields #z through the execution of M_i's instructions, then we say that:

$$M_i(x) = z.$$

Thus, if the Turing machine M_i is presented with x as its input and eventually halts (after computing for a while) with z written on its tape, we think of M_i as a function whose value is z for the input x.

erase x, find first symbol of y

I1	#	#	right	same
	0	b	right	same
	1	b	right	same
	b	b	right	next

get next symbol of y - mark place

I2	0	&	left	next
	1	&	left	I5
	b	b	halt	

find right edge of output - write 0

I3	b	b	left	same
	#	#	right	next
	0	0	right	next
	1	1	right	next
I4	b	0	right	I7

find right edge of output - write 1

I5	b	b	left	same
	#	#	right	next
	0	0	right	next
	1	1	right	next
I6	b	1	right	next

find the & and resume copying

I7	b	b	right	same
	&	b	right	I2

Figure 4 - Selection Machine

The Turing machine in figure 4 is what we often call a *selection* machine. These machines receive several numbers (or strings) as input and select one of them as their output. An example is the function:

$$M(x, y, z) = y$$

which selects the second of its three inputs. (This generalizes to any number of inputs, but let us not get too carried away.) This machine receives two inputs

and selects the second. It computes the function M(x, y) = y. (Recall that the input string consists of two binary numbers separated by a blank.)

Looking carefully at this machine, it should be obvious that it:

1) erases x, and
2) copies y next to the endmarker (#).

But, what might happen if either x or y happens to be blank? Figure it out! Also determine exactly how many steps this machine takes to erase x and copy y. (The answer is about n^2 steps if x and y are each n bits in length. But, why not find a more precise answer yourself.)

find the right end of the input

I1	**0**	**0**	**right**	**same**
	1	**1**	**right**	**same**
	#	**#**	**right**	**same**
	b	**b**	**left**	**next**

is low order bit is 0 or 1?

I2	**0**	**b**	**left**	**next**
	1	**b**	**left**	**I5**
	#	**#**	**right**	**I6**

erase input and print 1

I3	**0**	**b**	**left**	**same**
	1	**b**	**left**	**same**
	#	**#**	**right**	**next**
I4	**b**	**1**	**halt**	

erase input and print 0

I5	**0**	**b**	**left**	**same**
	1	**b**	**left**	**same**
	#	**#**	**right**	**next**
I6	**b**	**0**	**halt**	

Figure 5 - Even Integer Acceptor

A very important family of functions are those which describe relations (or predicates) and membership in sets. These are called *characteristic functions,* or *0-1 functions* because they only take values of zero and one. An example is the characteristic function for the set of even integers. It may be denoted:

$$\text{even}(x) = \begin{cases} 1 \text{ if } x \text{ is even} \\ \\ 0 \text{ otherwise} \end{cases}$$

and is computed by the Turing machine of figure 5.

This machine leaves a one upon its tape if the input ended with a zero (thus an even number) and halts with a zero on its tape otherwise (for a blank or odd integer). It should not be difficult to figure out how many steps it takes for an input of length n.

Now for a quick recap and a few more formal definitions. We know that Turing machines *compute functions.* Also we have agreed that if a machine receives x as an input and halts with z written on its tape, or in our notation:

$$(I1)\#x \Rightarrow \#z$$

then we say that M(x) = z. When machines never halt (that is: run forever or reach a non-halting terminal configuration) for some input x we claim that the value of M(x) is *undefined* just as we did with programs. Since output and halting are linked together, we must precisely define halting.

> **Definition.** *A Turing machine **halts** if and only if it encounters a halt instruction during computation and **diverges** otherwise.*

So, we have machines that always provide output and some that do upon occasion. Those that always halt compute what we shall denote the ***total functions*** while the others merely compute ***partial functions***.

We now relate functions with sets by discussing how Turing machines may characterize the set by deciding which inputs are members of the set and which are not.

> **Definition.** *The Turing machine M **decides membership** in the set A if and only if for every x, if $x \in A$ then M(x) = 1, otherwise M(x) = 0.*

There just happens to be another method of computing membership in sets. Suppose you only wanted to know about members in some set and did not care at all about things that were not in the set. Then you could build a machine which halted when given a member of the set and *diverged* (ran forever or entered a non-halting terminal configuration) otherwise. This is called *accepting* the set.

> **Definition.** *The Turing machine M **accepts** the set A if and only if for all x, M(x) halts for x in A and diverges otherwise.*

This concept of acceptance may seem a trifle bizarre but it will turn out to be of surprising importance in later chapters.