Machine Enhancement

We know now that Turing machines and programming languages are equivalent. Also, we have agreed that they compute all of the things that we are able to compute. But, maybe there is more we can discover about the limits of computation.

With our knowledge of computer science, we all would probably agree that not too much can be added to programming languages that cannot be emulated in either the *NICE* or the *SMALL* language. But, maybe a more powerful machine could be invented. Maybe we could add some features to these rather restricted Turing machines and gain computing power. This very endeavor did claim the attention of logicians for a period of time. We shall now examine some of their attempts to build a better Turing machine.

Let us begin with one of the most unappetizing features of Turing machines, namely the linear nature of their tapes. Suppose we had a work surface resembling a pad of graph paper to work upon? If we wished to add two numbers (such as 1011 and 1110) it would be nice to be able to write them one above the other like this:

#	1	0	1	1	. . .
	1	1	1	0	. . .
					. . .

and then add them together in exactly the manner that we employ when working with pencil and paper. If we sweep the tape from right to left, adding as we go, we produce a tape that looks like this:

#	1	0	1	1	. . .
	1	1	1	0	. . .
1	1	0	0	1	. . .

What we have been looking at is the tape of a *3-track Turing machine.* Instructions for a device of this nature are not hard to imagine. Instead of reading and writing single symbols, the machine reads and writes *columns* of symbols.

The 3-track addition process shown above could be carried out as follows. First, place the tape head at the right end of the input. Now execute the pair of instructions pictured in figure 1 on a sweep to left adding the two numbers together.

Add without carry

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

Add with carry

0 0 b	0 0 1	left	previous
0 1 b	0 1 0	left	same
0 b b	0 b 1	left	previous
1 0 b	1 0 0	left	same
1 1 b	1 1 1	left	same
1 b b	1 b 0	left	same
# b b	# b 1	halt	

Figure 1 - Addition using Three Tracks

If we compare this 3-track machine with the 1-track machine designed to add two numbers, it is obvious that the 3-track machine is:

- easier to design,
- more understandable, and
- far faster

than its equivalent 1-track relative. One might well wonder if we added more power to Turing machines by adding tracks. But alas, this is not true as the proof of the following result demonstrates.

> **Theorem 1.** *The class of functions computed by n-track Turing machines is the class of computable functions.*

Proof Sketch. Since n-track machines can do everything that 1-track machines can do, we know immediately that every computable function can be computed by an n-track Turing machine. We must now show that for every n-track machine there is an equivalent 1-track machine.

(Rather than show this result for all values of n at once, we shall demonstrate this for 2-track machines and state that extending the result to any number of tracks is an obvious extension of our construction. The major reason for this is that concrete examples are always easier to formulate and understand.)

Our strategy is to encode columns of symbols as individual symbols. For example, for two-track machine using zero, one and blank we might use the following encoding:

column symbols	0 0	0 1	0 b	1 0	1 1	1 b	b 0	b 1	b b
code	r	s	t	u	v	w	x	y	b

and then whenever we see a tape such as:

0	1	0	0		. . .
0	0	1		1	. . .

we merely encode it onto a 1-track tape like this:

r	u	s	t	y	. . .

Now all that is needed is to translate the instructions for a 2-track machine into equivalent instructions for the 1-track machine that uses the encoding. For example, the top instruction below may be replaced by the bottom one.

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

r u b	u v t	right left halt	next same

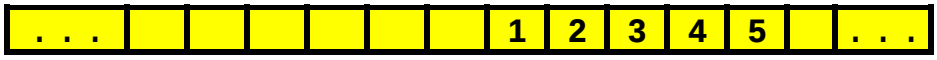
The remainder of the proof consists of the argument that as the original 2-track machine performs steps of a computation, the new 1-track machine will perform equivalent steps. And, for the final touch, we could very easily write a *decoding* routine which would write the answer (which appears on one of the tracks) on the tape before halting. Thus, the machines compute exactly the same function.

(**N.B.** We should note that the argument above and many of the arguments used to indicate proofs of theorems were not complete proofs. This is not because they're not needed in theoretical computer science - they are *very* important and will be provided for many theorems. But, proofs for simulation results are very long and tedious and so we have been using informal proofs or proof sketches thus far. It is expected that anyone with a bit of mathematical maturity could easily fill in the needed details and make the proof sketches complete.)

We will mention one more topic on k-track machines before moving along. Do we add complexity to computation by using 1-track machines? Are k-track machines faster, and if so, just how much faster are they? (Or, how much slower are 1-track Turing machines?) Try to get an estimate from the above construction.

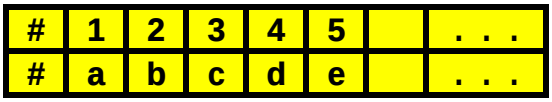
One of the ways in which a Turing machine could terminate its computation was to fall off the left end of the work tape. Often Turing machines are defined with tapes that stretch arbitrarily far in both directions. This means that the machine's tape head cannot leave the tape. As you might expect, this adds no power to a Turing machine. Let us explore a method for changing tapes with no ends into tapes with a left end.

Suppose a Turing machine began computation on a tape that looked like this:



and then moved left writing symbols until the tape resembled:

If we folded the tape in half we would still have all of the same information as before, but on a one-ended tape. The above example would look like the two-track tape pictured below.



Now that we have folded the *2-way* arbitrarily long tape onto a one-ended tape, the instructions of the old *2-way* machine need to be transformed into those befitting a *1-way* machine with two tracks. This is rather straightforward. We merely force the machine to use the top track until it wishes to move left off the end of the tape at the endmarker. Then we make our new machine transfer its attention to the lower track. On the lower track all moves are reversed. For example, if the *2-way* machine possesses the instructions *I1, I2, ... , In* and instruction *Ik i*s:

0	**1**	**right**	**next**
1	**0**	**left**	**same**

then the new machine would have two corresponding instructions, one for the top track (or right half of the tape), which looks like this:

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

and one with the directions reversed which writes on the lower track (or the left end of the tape). The second of these instructions will be labeled *I(n + k).*

We now have doubled the instructions for our machine. The *I1, I2, ... , In* instructions work on the top track and instructions *I(n+1), I(n+2), ... , I(2n)* work on the lower track.

All that remains is to coordinate the machine's moves so that it can switch between tracks. This happens when the machine hits the leftmost square - the one containing the endmarkers. We can make the machine bounce off the endmarkers by adding the pair of lines:

#	**#**	**right**	**I(n+k)**
#	**#**		

to each *Ik* in the instruction set *I1, ... , In* and the lines:

#	#	**right**	**Ik**
#	#		

to each of the instruction in *I(n+1), ... , I2n*.

Since the 2-track machine of the previous construction carries out its computation in exactly the *same* manner (*and in exactly the same time)* as the original *2-way* machine, we shall claim that they are equivalent. And, since we know how to change a 2-track machine into a 1-track machine, we shall state the following theorem without proof.

> **Theorem 2.** *Turing machines with 2-way arbitrarily long tapes compute exactly the class of computable functions.*

Thus far we have discovered that adding extra tracks or 2-way tapes to Turing machines need not increase their power. These additions were at some additional cost however. In fact, we were forced to introduce extra symbols. (This cost us some speed also, but we will save this for later.)

Our next question is to inquire about exactly *how many symbols a Turing machine needs.* Obviously no computation can be done on a machine which uses only blanks. Thus we need at least one additional symbol. In fact, one more will do. But, the proof is easier (and much more fun to read) if we show that any of the computable functions can be computed using only blanks, zeros, and ones. The demonstration that they can be computed using only blanks and one additional symbol will be reserved for readers to carry out in the privacy of their own homes.

> **Theorem 3.** *Turing machines that use n symbols are not more powerful than those that use two symbols (and blanks).*

> **Proof sketch.** We shall show how to simulate the operation of an n symbol machine with a machine that uses a binary alphabet. The special case of quaternary (n = 4) alphabets is presented below, but it is easily extended to any number of symbols.
>
> We must mimic a four-symbol computation on a binary tape. So, an encoding of the four symbols (0, 1, 2, 3, and *b*) into our binary alphabet is done according to the following chart

b	0000
0	1000
1	1100
2	1110
3	1111

The tape we use will be laid out in blocks of squares. We use one of these blocks for each square of the 4-symbol machine's tape. For example, consider a tape for the 4-symbol machine that contains this sequence of symbols:

Translating each symbol into a four-bit code with the above chart produces a tape like this:

for our new machine which operates on the encoded symbols.

Assuming that the 4-symbol machine receives its input as a binary number, the new binary machine must:

a) Change the binary input to encoded form,
b) Simulate the 4-symbol machine's computation, and
c) Translate the final tape to a binary number.

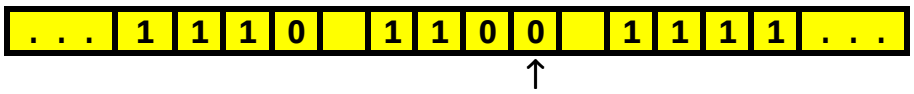
Steps (a) and (c) are left as exercises. Step (b), the simulation is done in four steps for each instruction of the original 4-symbol machine. These are:

a) Determine what symbol occupies the current block.
b) Write the appropriate symbol in this block.
c) Move the tape head to the correct neighboring block.
d) Transfer control to the appropriate next instruction.

(Note that we are merely doing for blocks what was done before for squares. It is really rather simple.)

Now for the sordid details. For each instruction *Ik* of the 4-symbol machine we shall have a group of instructions. One part of this group will read the encoded symbol and will begin with the instruction labeled *Ik-read*. Other parts of this group will contain instructions to do the writing, moving, and transfer for each different symbol read. These action parts will begin with instructions labeled *Ik-saw-b, Ik-saw-1,* etc. depending upon which symbol (*b*, 1, 2, 3) was detected by the *Ik-read* group.

Let's look at an example. We pick up our new machine's computation with its head at the right end of a block like this:



and execute the reading part (labeled *Ik-read*) of the group of instructions for the original machine's instruction *Ik*. This set of instructions just counts the 1's in the block and reports the symbol encoded by the block.

0	0	left	same
1	1	left	next
b	b	right	Ik-saw-b

1	0	left	next
b	b	right	Ik-saw-0

1	0	left	next
b	b	right	Ik-saw-1

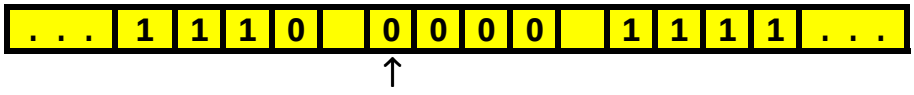
1	0	left	next
b	b	right	Ik-saw-2

b	b	right	Ik-saw-3

At this time our machine has

- read and decoded the symbol,
- erased the symbol in the block,
- placed it's head at the left end of the block,
- and transferred to an instruction to do writing, etc.

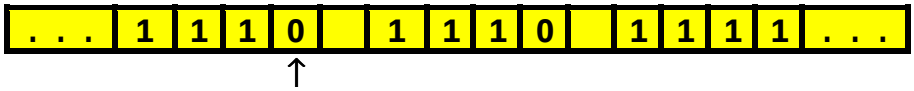
Now the tape looks like this:



We are ready to execute the instruction *Ik-saw-1*. *If Ik* of the original machine said to write a two, move to the left, and transfer to *Im* upon reading a one, then the instruction group *Ik-saw-1* consists of:

0	1	right	next
0	1	right	next
0	1	left	next
1 b	1 b	left left	same Im-read

and after executing them, our new machine ends up in the configuration:



A move to the right is similar in that it merely skips over all of the zeros in the block just written and the next block also in order to end up at the left end of the block to the right. If no block exists to the right then a new one must be written containing a blank, but this is a rather simple task.

To finish up the proof, we must show that for each configuration in the computation of the 4-symbol machine, there is an equivalent encoded configuration in the computation of our binary machine. Thus for each and every step of the 4-symbol machine there is an equivalent sequence of steps which the new binary machine executes. From this we can conclude that both machines compute the same function.

[**N.B.** *An interesting problem concerning the transformation of one type of Turing machine to another is the amount of added complexity or steps needed by the new machine. Try to develop some general formulas for this.]*

Due to this sequence of theorems, it will be possible to limit our attention to Turing machines that use a binary alphabet whenever we wish to prove something about the computable functions. There is no need to show things about machines with several tracks, or symbols, or even tapes. In other words, whenever something is true for the functions computed by these binary machines it is true for all Turing machines computable functions and for the class of computable functions. And, of course, when we wish to design a particular machine we shall have no fear about including extra symbols or tracks since we know that if we wished we could design an equivalent binary, one track machine.

Other variations or options have been proposed and used since Turing's day. Features such as additional heads, extra tapes, and even machines which

compute in several dimensions have been defined and proven to be equivalent to what we shall now call our *standard* (one track, one tape, one head, binary alphabet) Turing machine.