11.2.2 Multitape Turing Machine

Another important modification of the basic Turing Machine model is a *multitape Turing Machine*. A Multitape Turing Machine has multiple tapes, each of which has its own tape head (See Figure 11.6).

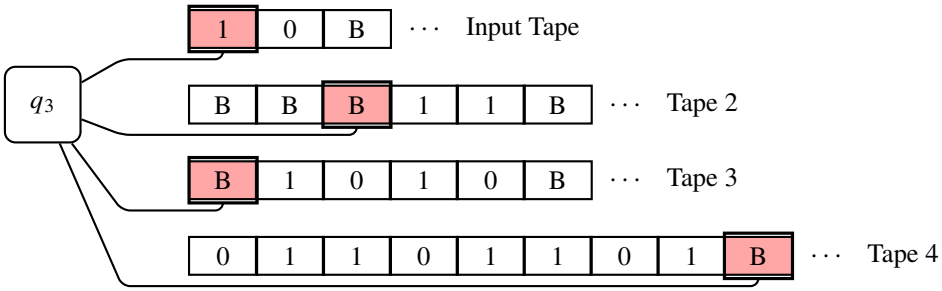


Figure 11.6: A 4-Tape Turing Machine

The machine is in one state at a time. In each move, the machine reads all the symbols in the cells scanned by type heads, overwrites each of these symbols with new symbols, moves each of the tape heads and goes to the new state. The transition function for the multitape Turing Machine with n tapes is of the form

$$\delta(q_0, X_1, X_2 \ldots, X_n) = (p, [Y_1, D_1], [Y_2, D_2], [Y_3, D_3], \ldots, [Y_n, D_n]).$$

This means that the machine in the state q and scanning the symbol X_1 on the first tape, X_2 on the second tape and so on, moves to the state p, replaces the symbol X_1 with Y_1 on the first tape and moves the tape head of that tape in the direction D_1, replaces the symbol X_2 with Y_2 on the second tape and moves the tape head of that tape in direction D_2 and so on.

One tape of the machine is the input tape. The multitape Turing Machine starts the computation on string w with string w on its input tape, and with all the other tape containing just blank symbols. The string is accepted if the machine at some point during its processing enters one of the accepting states.

Theorem 11.2: Equivalence Between Multitape Turing Machines and Basic Turing Machines

The class of languages accepted by the Multitape Turing Machines is the same as the class of languages accepted by basic Turing Machines.

The process of simulating a multitape Turing Machine by a basic, single-type Turing Machine is conceptually simple, but quite tedious to describe precisely. It is described in some detail in the "Multitape Turing Machines" video. We are going to omit the details here.

11.3 Turing Machines as Computers of Integer Functions

So far, we have seen Turing Machines as language acceptors, in that they would receive a string as an input, and would produce "yes" or "no" as an answer (depending on whether the input string is accepted or not), or would alternatively loop forever. But, thanks to the fact that the Turing Machine can write on its tape, Turing Machines can also be seen as computers of functions with integer arguments. Here we are going to focus only on functions that accepts non-negative integers and return a non-negative integer, but this is easily extended to negative integers too. The traditional (and very inefficient) approach is to represent integers in unary representation - the integer number $i \geq 0$ is encoded by the string 0^i. Thus, 0 is encoded by ϵ, 1 is encoded as 0, 2 is encoded as 00 and so on. In this way, we can encode every integer number as a string of 0's. For functions that accept tuples of integer numbers, e.g. $(i_1, i_2, \ldots, i_k)$, we can encode each element of the tuple and then concatenate all these encodings together, separating them by a symbol 1. Thus, the tuple $(i_1, i_2, \ldots, i_k)$ can be encoded as $0^{i_1}10^{i_2}1 \ldots 10^{i_k}$. For example, the pair $(3, 2)$ can be encoded as 000100. In this way, we encode tuples of integer numbers as strings over the alphabet $\{0, 1\}$. Encoding of an input to some integer function can be placed on the input tape of a Turing Machine, thus serving as an input of that TM.

If the Turing Machine halts on such an input, regardless of whether it does so in an accepting or non-accepting state, with the tape consisting of 0^m for some m, then we say that $f(i_1, i_2, \ldots, i_k) = m$, where

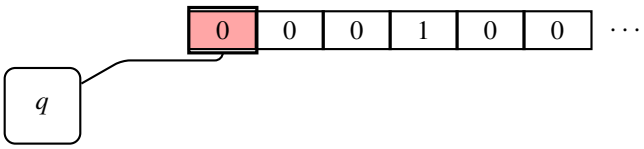
f is the function of k arguments computed by this Turing Machine. We do not require the m 0's to be left-aligned on the tape. The tape can have some blank symbols at the start, followed by m 0's, followed by blanks - this is still interpreted as an encoding of m. Note that we here do not require for the Turing Machine to halt on every input that is encoding of a tuple of k integer numbers. If a function f of k arguments is computed by a Turing Machine that halts on an encoding of every k-tuple of integers, then f is called *totally recursive function*. If f is computed by a Turing Machine (regardless of whether it halts on all encodings of k-tuples of integers or not), f is called a *partially recursive function*. Totally recursive functions are, in a sense, equivalent to totally decidable languages (also called recursive), whereas the partially recursive functions are equivalent to partially decidable (or recognisable) languages. It is worth noting that all common arithmetic functions on integers, such as addition, multiplication and factorial, are total recursive functions.

We will now see two examples of Turing Machines for computing arithmetic functions - TM for addition and TM for proper subtraction.

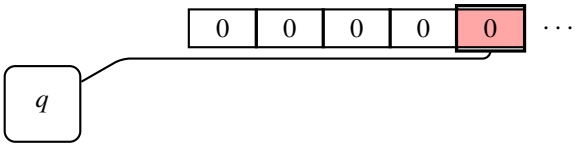
Example 11.2: Turing Machine for Addition

Find a Turing Machine that computes the addition function on positive integers. That is, find a Turing Machine that computes the function $f(i_1, i_2) = i_1 + i_2$.

Solution. This example is fairly easy. The input tape of our Turing Machine at the start of the computation contains i_1 0's, followed by 1, followed by i_2 0's, followed by B's (Figure 11.7a). At the end of the computation, it has to contain $i_1 + i_2$ 0's, followed by B's (Figure 11.7b). Therefore, all we have to do to make this happen is, starting from the initial configuration, go over all the 0's, replace the 1 with 0, then go over all the following 0's until we encounter a B. Note that now we have one extra 0 on the tape, so we need to delete one (i.e. replace it with B). For this, we simply move the tape head one cell to the left of the first B, and replace the symbol there with B. This Turing Machine is so simple that we can easily give its complete transition table, which is shown in Table 11.1. Note that we also have to deal with corner cases when one (or both) arguments are 0, which is represented by the empty string ϵ.



(a) Addition Turing Machine at the Start of Computation for $i_1 = 2$ and $i_2 = 3$.



(b) Addition Turing Machine at the End of Computation for $i_1 = 2$ and $i_2 = 3$.

Figure 11.7: Start and End Configuration of an Addition Turing Machine

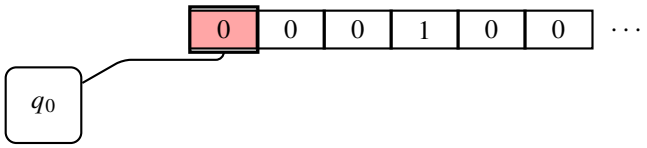
δ	0	1	B
$\rightarrow q_0$	$(q_o, 0, R)$	$(q_1, 0, R)$	-
q_1	$(q_1, 0, R)$	-	(q_2, B, L)
q_2	(q_3, B, R)	-	-
$*q_3$	-	-	-

Table 11.1: Transition Table for an Addition Turing Machine

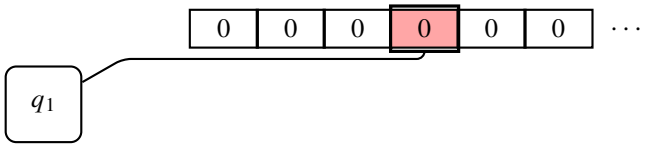
It is pretty clear what the states in our Turing Machine represent. q_0 is the state in which the tape head goes over all 0's of the representation of i_1. When it encounters 1, it means we have reached the separator between i_1 and i_2. We change that 1 to 0 and go to the state q_2. In the state q_2, the tape head goes over all the 0's of the representation of i_2, until it encounters B. When this happens, the Turing Machine moves to the state q_2 and the tape head returns one cell to the left. In the state q_2, a single 0 is replaced by B and the machine enters the accepting state. It is worth analysing what happens for the corner cases, e.g. when $i_1 = 0$ and

$i_2 = 0$. In this case, the tape will contain just a single 1, followed by B's. That 1 will be replaced by 0, and then changed to B, therefore the tape will at the end contain only B's. This tape represents the string ϵ, which is the representation of the integer 0, which is indeed the result we expect. Analysis of other corner cases, when $i_1 = 0$ or $i_2 = 0$ is left to the reader. □

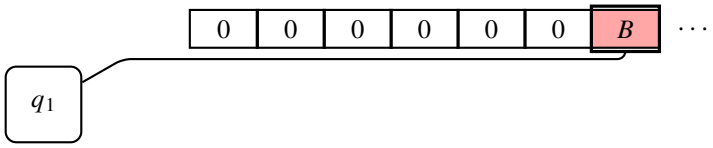
One thing to note is that in the example above, we do not analyse the input to make sure that it is in the right format. We do not care what happens when the input on the tape is not an encoding of a pair of integers (i_1, i_2). Our Turing Machine can return anything we want in these cases (or even not halt at all). The important thing is to halt with the appropriate content of the tape when the input indeed contains an encoding of a pair of integers. Figure 11.8 shows a few snapshots of the computation of our Turing Machine on input 000100.



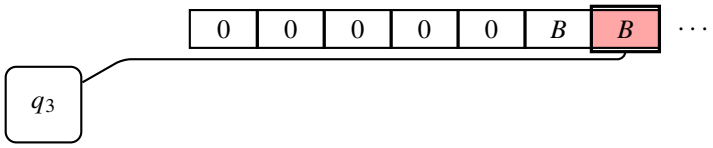
(a) Addition Turing Machine at the Start of Computation for $i_1 = 2$ and $i_2 = 3$.



(b) Addition Turing Machine after the First 1 has been replaced by 0.



(c) Addition Turing Machine when Passing Over 0 of i_2 in State q_1.



(d) Addition Turing Machine in its Final Configuration for $i_1 = 3$ and $i_2 = 2$.

Figure 11.8: Various Configurations of an Addition Turing Machine for Input 000100.

Example 11.3: Turing Machine for Proper Subtraction

Find a Turing Machine that Computes the proper subtraction function. The proper subtraction for i_1 and i_2 is $i_1 - i_2$ if $i_1 \geq i_2$, and 0 otherwise.

Solution. This Turing Machine is a bit trickier to design, but also not overly problematic. Given input of i_1 0's, followed by 1, followed by i_2 0's, the tape at the end of computation needs to be either completely blank (in case that $i_2 > i_1$), or to contain $i_1 - i_2$ 0's, followed by blanks. The idea is to erase, for every 0 of the encoding of i_2, a single 0 in the encoding of i_1. In this way, we will in the end erase i_2 0's from the encoding of i_1, leaving $i_1 - i_2$ 0's in it - exactly what we need. The transition function of our Turing Machine is given in the Table 11.2.
The states in the transition function have the following purposes:

- q_0 is the start state, and also the state where we need to erase one 0 from the encoding of i_1. If 0 is scanned in this state, it is erased (replaced with B), and the Turing Machine goes to the state q_1. Alternatively, if 1 is encountered, it means all the 0's from the encoding of i_1 have been erased. In this case, the Turing Machine goes into the state q_4, where it needs to erase the rest of the tape, as the final result should be 0.

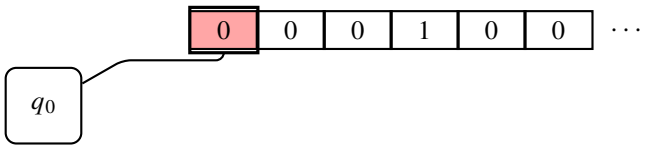
δ	0	1	B
$\rightarrow q_0$	(q_1, B, R)	(q_4, B, R)	-
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_2, B, L)
q_2	(q_3, B, L)	$(q_F, 0, R)$	-
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	(q_4, B, R)	-	(q_F, B, R)
$*q_F$	-	-	-

Table 11.2: Transition Table for a Proper Subtraction Turing Machine

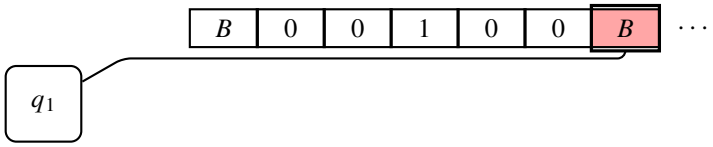
- q_1 is the state where one 0 has been erased from the encoding of i_1. We now need to erase one 0 from the encoding of i_2. We choose to erase the last 0 from it. We simply pass over all the 0's and 1's until we encounter B. Then the tape head moves to the left and the state q_2 is entered.
- q_2 is the state where we need to erase the symbol 0 which the tape head is possibly scanning. If the scanned symbol is indeed 0, it is erased and the Turing machine goes to the state q_3. If there is no more 0 from the encoding of i_2, then all is done - all the 0's from the encoding of i_2 have been erased and the tape head is scanning the symbol 1. Note that we have previously erased one 0 from the encoding of q_0, so the tape now has one fewer 0 than needed. We solve this by replacing the symbol 1 on the tape with B and moving to the accepting state.
- q_3 is entered when we have erased one 0 from the encoding of i_2. We then need to go back to the remaining symbols of the encoding of i_1 and erase the next 0 from it. So, in the state q_3, the tape head simply goes over all the 0's and 1's to the left until it encounters B (which is the position of the last erased symbol 0 from the encoding of i_1), then moves one place to the right and goes back to the state q_0.
- q_4 is the state where we have erased the last symbol 0 of the encoding of i_1. Therefore, the result needs to be 0. We know that all the cells to the left of the currently scanned cell are B, so we simply go over all the non-blank cells to the right and erase them. The tape will then comprise of only B's, which is the encoding of 0.
- q_F is the accepting state

Figure 11.9 shows a few configurations of this Turing Machine when processing the input 000100 (that is, computing the proper subtraction for $i_1 = 3$ and $i_2 = 2$) while Figure 11.10 shows a few configuratins when processing the input 0100, computing the proper subtraction for $i_1 = 1$ and $i_2 = 2$. Note that for this Turing Machine, the encoding of the result is not aligned to the left on the tape. □

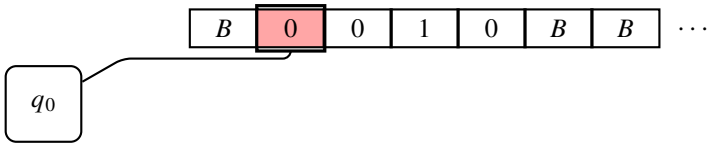
In a similar way and with some effort, we can find Turing Machines that will compute the other common arithmetic functions over integers, such as multiplication, integer division, division remainder, exponent and so on.



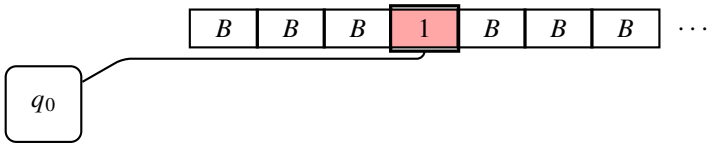
(a) Subtraction Turing Machine at the Start of Computation for $i_1 = 2$ and $i_2 = 3$.



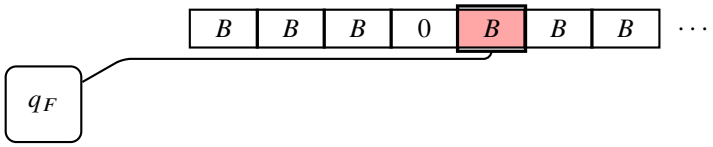
(b) Subtraction Turing Machine after the first 0 of i_1 has been erased and the rest of the tape is scanned for the last 0 of i_2.



(c) Subtraction Turing Machine after the first 0 of i_2 has been erased and the next 0 of i_2 to be erased has been found.

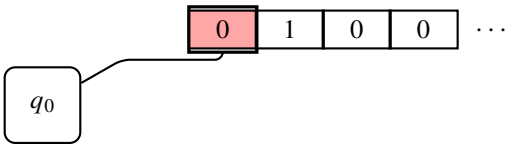


(d) Subtraction Turing Machine after all the 0's of i_2 has been erased, and one more than needed 0 of i)2 has been erased.

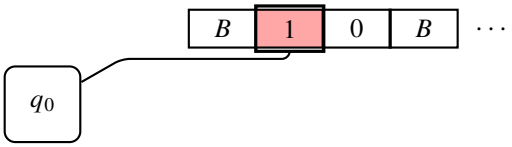


(e) Final configuration of Subtraction Turing Machine for input $i_1 = 3$ and $i_2 = 2$.

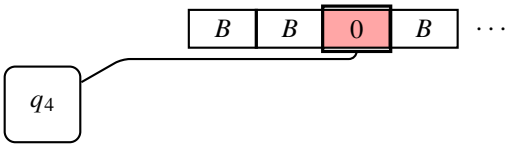
Figure 11.9: Various Configurations of the Subtraction Turing Machine for Input 000100.



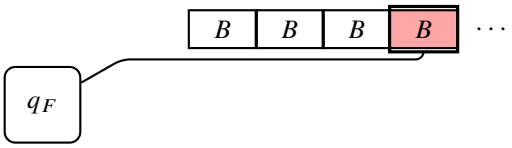
(a) Subtraction Turing Machine at the Start of Computation for $i_1 = 1$ and $i_2 = 2$.



(b) Subtraction Turing Machine after the first 0 of i_1 and one 0 of i_2 have been erased.



(c) Subtraction Turing Machine after all the 0's of i_1 have been erased. The machine is in the state q_4 and it needs to erase every non-blank symbol on the tape.



(d) Final configuration of Subtraction Turing Machine for input $i_1 = 1$ and $i_2 = 2$.

Figure 11.10: Various Configurations of the Subtraction Turing Machine for Input 0100.