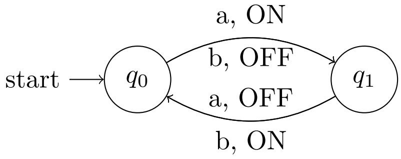
Lecture 4

Gidon Rosalki

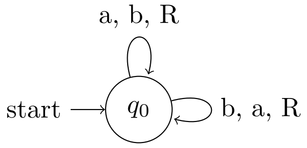
2025-04-20

We have extensively discussed DFAs, and mostly solved the problem (not entirely, but for our purposes). These could only take input, and would change state according to them, and only them. However, what if there is a state machine, that not only takes inputs, but also takes input from the environment? Let us consider for example a state machine that takes in the input from some sensor (temperature, air pressure, etc), and outputs light or dark depending on the input. Since we are also dependent on external values, and we want more options for outputs, we do not want to simply have true or false as our output, but more options. So we shall have states, and letters that move us from state to state, and for these transitions, the transitions will also be annotated with the output. For example:

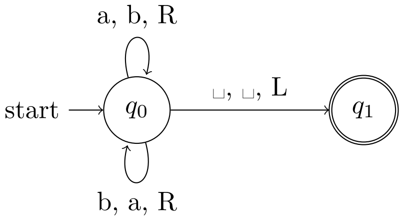


1 Turing machines (TMs)

A Turing machine has a tape, comprised of cells, and the read/write head which points at a cell. In each cell of the tape there is inscribed a letter. Additionally, the TM has a state machine, and it is currently located in a state. Every time it reads a letter from the tape, it transitions to a state, and emits both a letter, and a direction (R/L). Emitting a letter means rewriting the current cell's letter (**before** moving on the tape). For example:

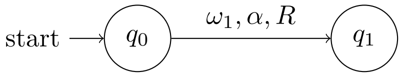


So here, when the TM reads an a, replaces it with b and moves right on the tape. For when we reach the end of the tape, we will define the letter "␣", which is an empty letter. We can additionally then add on to our state machine a final state as follows:



1.1 Runs on TMs

Let there be a series of cells $\omega_1, \dots, \omega_k$ on the tape. We shall state that ω_1 is the starting cell, so at ω_1 we know that the machine is in the initial state (generally q_0). We can call this entire definition the **configuration**. Should we consider the following machine:



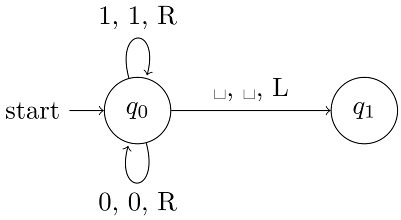
Then the configuration C_0 will be the series of states $q_0 \to \omega_1, \dots, \omega_k$. Should we take the configuration of $\alpha, q_1 \to \omega_2, \dots, \omega_k$, then we are in configuration C_1, which is the subsequent configuration to C_0. The final configuration is a configuration that finishes in a final state.

Now that we have configurations, we may define a run on a TM: A partial run on the TM T, with input ω is a series (finite or infinite) of configurations $C_0, C_1, \dots$ such that $C_0 = q_0 \to \omega_1, \dots, \omega_k$ is the starting configuration in the run of T on ω, and for all $i > 0$, C_i is the following configuration of C_{i-1}. The run is **full** if it is finite, and finishes in a final configuration, or if it is infinite.

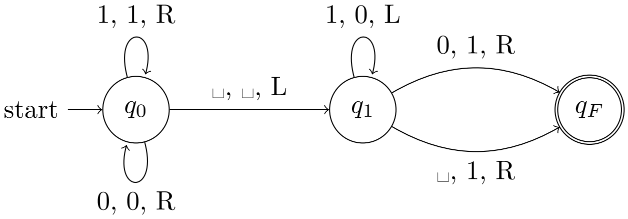
The **runtime** of a TM T on the input ω is the length of the full run on ω minus 1.
The **output** is the content of the tape at the end of the full run (if it is finite). **Configuration**: This can be shown graphically as $\alpha_1, \ldots, q \to \alpha_i, \ldots, \alpha_k$, but we may represent it as well as follows: $\alpha_1 \ldots \alpha_{i-1} q \alpha_i \ldots \alpha_k$

1.2 Successor

Let us consider an problem with the input $x \in \{0,1\}^*$, and the output is $x+1$. We will note that adding 1 to a binary number is simply converting all the rightmost 1s that touch the rightmost bit to 0, and converting the first 0 to 1 (101011 → 101100). To create a TM that does this, we will do the following: Firstly, we will find the rightmost edge of the input:



Next, we will convert all the 1s to 0s until we find 0. We will convert it to 1, and finish.

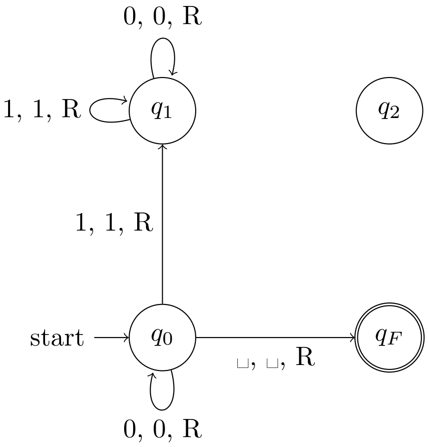


The runtime will be $O(2n) + O(2) (= O(n))$, since in the worst case we pass over the whole number twice, and off one space on the right, and one space on the left.

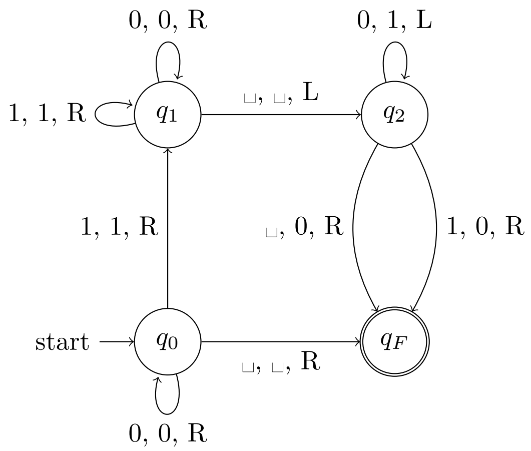
1.3 Predecessor

Input: $x \in \{0,1\}^*$
Output: $\begin{cases} x-1, & \text{if } x > 0 \\ x, & \text{otherwise} \end{cases}$

This algorithm can be seen as starting from the right, converting all the 0s to 1s, until we find a 1, at which point we convert it to 0, and finish (unless $x = 0$).
To do this, we will start by finding the rightmost edge of the input, but if the input is 0, we will stop.



Next is like above, but swapping 0s and ones



1.4 Formal definition

A Turing Machine T can be described as follows:

$$T = (\Sigma, \Gamma, \text{␣}, Q, q_0, F, \delta)$$

Where

- Σ is the alphabet of the input, a finite non empty set.
- Γ is the working alphabet, $\Sigma \subseteq \Gamma$ (and is naturally also finite)
- ␣ is the empty letter, and $\text{␣} \in \Gamma \setminus \Sigma$
- Q is the set of states (finite)
- $q_0 \in Q$ the initial state
- $F \subseteq Q$ is the set of final states (Sometimes $F = \{q_F\}$, and sometimes $F = \{q_{acc}, q_{rej}\}$)
- δ the transition function, where $\delta : \Gamma \times (Q \setminus F) \to Q \times \Gamma \times \{R, L\}$

1.5 Addition

Input: $x\#y$ where $x, y \in \{0,1\}^*$ (note, $\# \in \Sigma$)
Output: $x + y\#$
Implementation:

1. We will go right until we are one cell to the right of #.
2. We will subtract 1 from y, and if $y = 0$ we will go over all the letters of y from right to left and convert them to spaces, and stop when we arrive at #.
3. We will run left until 1 after the #
4. We will add 1 to x
5. We will return to the first step

This is horrifically inefficient, and runs in exponential time. Effectively, the runtime is the value of y.

1.6 Concatenation

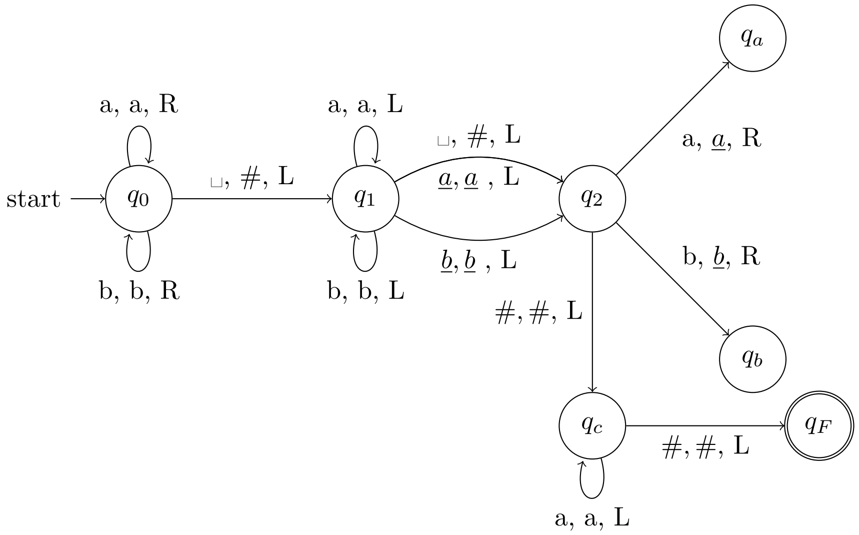
Input: $x \in \{a, b\}^*$
Output: $x\#x$
We could take the first letter, replace it with a space, and reinsert it after the #, but this will just leave us with the word moved to after the #. We shall add the symbols:

$$\Gamma = \Sigma \cup \Sigma \times \{\_, \hat{}\} \cup \{\text{␣}\} \cup \dots$$

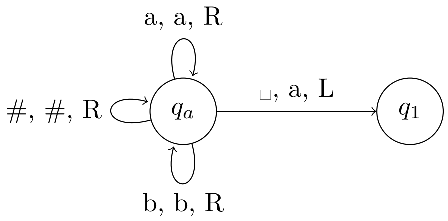
Technically, given $a \in \Sigma$, then it would be akin to $(a, \_) \in \Gamma$, but we will simply write $\underline{a}$.

1. Go right until the ␣, and replace it with #
2. We will go left until ␣ or until a diacritic letter, and then go one step right

3.



4.



1.7 Insertion

Input: $y\#x$
Output: $y\#\#x$
This is doable, and is left as an exercise to the reader (got to right until end, copy each letter right one until #, repeat and stop).

1.8 Multiplication

Input: $x\#y$
Output: $x \cdot y\#$
As in, multiplication of the binary numbers x and y. In short, we need to add x to itself y times.

To do this, we will copy the x to the left, so $x\#y \to x_c\#x\#y$, and then create $x_c\#x_c\#x\#y$. We then perform addition on the section $x_c\#x$, resulting in $x_c\#\left(x_c + x\right)\#y \to x_c\#2x\#\left(y - 1\right)$, and then return to the step where we copied over x_c.