**COMP2270/6270 – Theory of Computation**
**Tenth Week**

**School of Electrical Engineering & Computing**
**The University of Newcastle**

**Exercise 1)** Construct a standard 1-tape Turing machine *M* to compute each of the following functions:

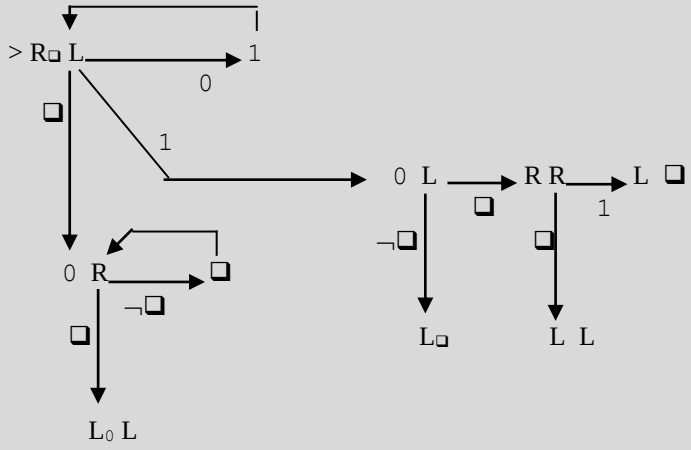
*a)* The function $sub_3$, which is defined as follows:

$$sub_3(n) = \begin{cases} n-3 & \text{if } n > 2 \\ 0 & \text{if } n \leq 2. \end{cases}$$

Specifically, compute $sub_3$ of a natural number represented in binary. For example, on input `10111`, *M* should output `10100`. On input `11101`, *M* should output `11010`. (Hint: you may want to define a subroutine.)

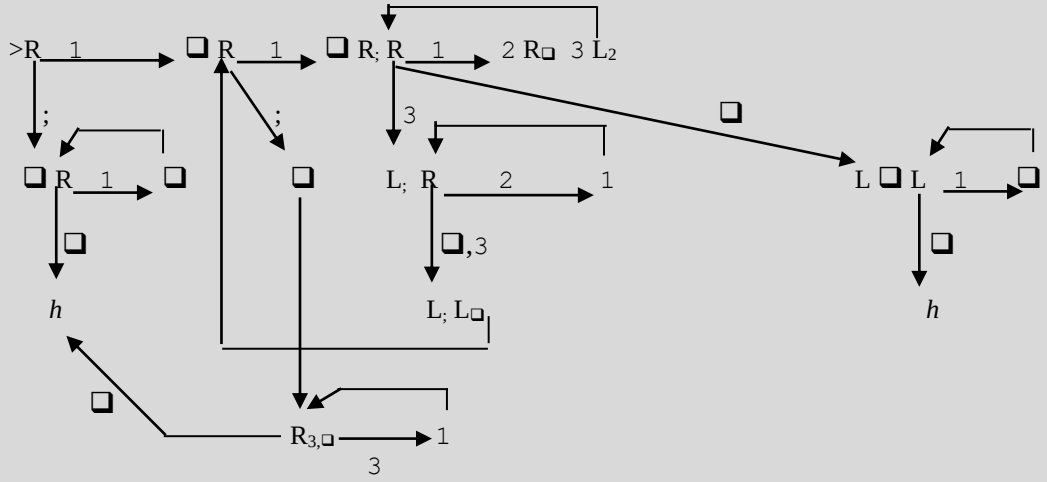
The design of *M* would be simpler if we could assume that it will be given as input a binary number without leading zeros. In that case, we could easily take the input strings `0`, `1`, and `10` as special cases. But it's harder to treat 0, 1, and 2 as special cases if leading zeros are allowed. We won't make the no leading zeros assumption here, however. So the input string might be `0001`, for example.

We first define a subroutine that we will call *S*. On input > 0, it will subtract a single 1. On input 0, it will simply return 0. It will leave leading zeros alone, but it won't introduce them if there were none.




Now we can define *M* as: > *S S S*

b) Multiplication of two unary numbers. Specifically, given the input string <*x*>;<*y*>, where <*x*> is the unary encoding of a natural number *x* and <*y*> is the unary encoding of a natural number *y*, *M* should output <*z*>, where *z* is the unary encoding of *xy*. For example, on input `111;1111`, *M* should output `111111111111`.




This machine first erases the first `1` in *x*. Then it uses each of the others to create a new copy of *y*. Each time it uses a `1` in *x*, it writes over it with a blank. Once all the ones in *x* have created their copies of *y*, the ; is erased.

**Exercise 2)** Define a Turing Machine *M* that computes the function *f*: {`a`, `b`}* → *N*, where:

*f*(*x*) = the unary encoding of *max*($\#_{\texttt{a}}(x)$, $\#_{\texttt{b}}(x)$).

For example, on input `aaaabb`, *M* should output `1111`. *M* may use more than one tape. It is not necessary to write the exact transition function for *M*. Describe it in clear English.

We can use 3 tapes:
Tape 1: input
Tape 2: write `1` for every `a` in the input
Tape 3: write `1` for every `b` in the input

- Step 1: Move left to right along tape 1. If the character under the read head is `a`, write `1` on tape 2 and move one square to the right on tapes 1 and 2. If the character under the read is `b`, write `1` on tape 3 and move one square to the right on tapes 1 and 3. When tape 1 encounters a blank, go to step 2.
- Step 2: Move all three read heads all the way to the left. Scan the tapes left to right. At each step, if there is `1` on either tape 2 or 3 (or both), write `1` on tape 1. Move right on all tapes except that as soon as either tape 2 or tape 3 (but not both) gets to a blank, stop moving right on that tape and continue this process with just the other one. As soon as the other of them (tape 2 or 3) also hits a blank, go to step 3. At this point, tape 1 contains the correct number of `1`'s, plus perhaps extra `a`'s and `b`'s that still need to be erased.
- Step 3: Scan left to right along tape 1 as long as there are `a`'s or `b`'s. Rewrite each as a blank. As soon as a blank is read, stop.

**Exercise 3)** Encode the following Turing Machine as an input to the universal Turing machine:

$M = (K, \Sigma, \Gamma, \delta, q_0, \{h\})$, where:
$K = \{q_0, q_1, h\}$,
$\Sigma = \{\mathtt{a}, \mathtt{b}\}$,
$\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, ❑\}$, and
$\delta$ is given by the following table:

| $q$ | $\sigma$ | $\delta(q, \sigma)$ |
|---|---|---|
| $q_0$ | A | $(q_1, \mathtt{b}, \rightarrow)$ |
| $q_0$ | B | $(q_1, \mathtt{a}, \rightarrow)$ |
| $q_0$ | ❑ | $(h, ❑, \rightarrow)$ |
| $q_0$ | C | $(q_0, \mathtt{c}, \rightarrow)$ |
| $q_1$ | A | $(q_0, \mathtt{c}, \rightarrow)$ |
| $q_1$ | B | $(q_0, \mathtt{b}, \leftarrow)$ |
| $q_1$ | ❑ | $(q_0, \mathtt{c}, \rightarrow)$ |
| $q_1$ | C | $(q_1, \mathtt{c}, \rightarrow)$ |

We can encode the states and the alphabet as:

| | |
|---|---|
| $q_0$ | q00 |
| $q_1$ | q01 |
| $H$ | q10 |
| A | a00 |
| B | a01 |
| ❑ | a10 |
| C | a11 |

We can then encode $\delta$ as:

(q00,a00,q01,a01,→),(q00,a01,q01,a00,→),(q00,a10,q10,a10,→),
(q00,a11,q00,a11,→),(q01,a00,q00,a11,→),(q01,a01,q00,a01,←),
(q01,a10,q00,a11,→),(q01,a11,q01,a11,→)

**Exercise 4)** What is the minimum number of tapes required to implement a universal Turing machine?

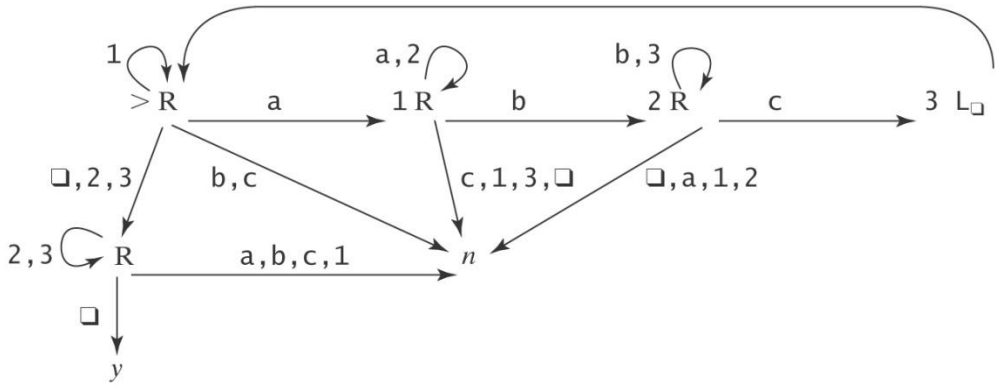
One. It can be implemented with three tapes as described in the book. But for every three-tape Turing machine, there is an equivalent one-tape one.

**Exercise 5)** In Example 17.9, we showed a Turing machine that decides the language W$\mathtt{c}$W. If we remove the middle marker $\mathtt{c}$, we get the language WW. Construct a Turing machine $M$ that decides WW. You may exploit nondeterminism and/or multiple tapes. It is not necessary to write the exact transition function for $M$. Describe it in clear English.

$M$'s first job is to find the middle of the string. It will shift the first half of the string one square to the left, then it will deposit # between the two halves. So $M$ moves the first character one square

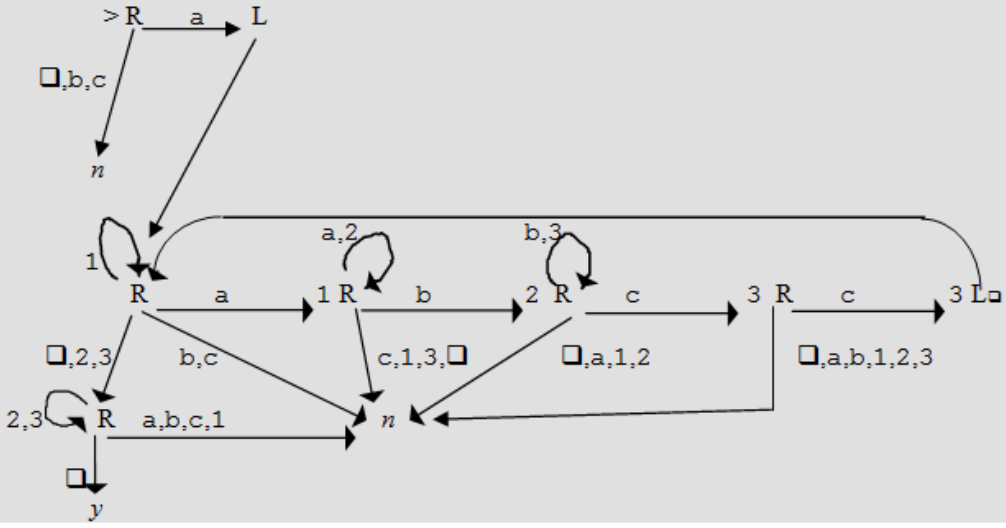
to the left. Then it nondeterministically decides whether it has moved half the string. If it decides it has not, it moves one square to the right and shifts it one square to the left, and continues doing that until it decides it's halfway through. At that point, it writes # between the two halves. Now *M* must compare the two halves. It does this by bouncing back and forth, marking off the first character and the first character after #. Then the second, and so forth. If it comes out even with all characters matching, it accepts. If either there is a mismatch or one string is longer than the other, that path rejects. If the # is in the middle, then *M* operates like the *wcw* machine described in the book.

**Exercise 6)** Consider the following Turing Machine *M*, taken from the book, that decides the language $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$:




Modify *M* so that it accepts $\{a^nb^nc^{2n} : n \geq 1\}$.

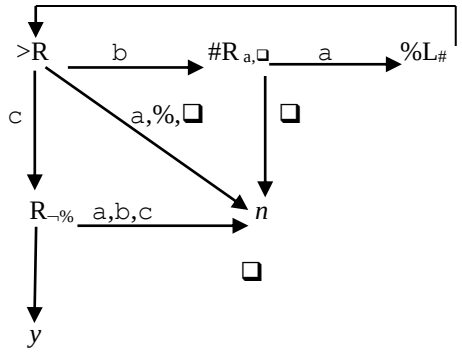
We need to make two changes: We'll add a test at the beginning to make sure that there is at least one a. And we'll modify the machine so that, when it finds a c, it must immediately find another one also.



**Exercise 7)** Consider a three-tape Turing machine $M$, where $\Gamma_M = \{\square, \mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Suppose that we want to simulate $M$ with a one-tape Turing machine $T$ using the technique described in Section 17.3.1. How large must $\Gamma_T$ be?

$$\begin{aligned}\Gamma_T &= \Gamma_M \cup (\Gamma_M \times \{0, 1\})^3 \\ |\Gamma_T| &= |\Gamma_M| + (2 \cdot |\Gamma_M|)^3 \\ &= 4 + 8^3 \\ &= 516\end{aligned}$$

**Exercise 8)** Give a clear formal description of language accepted by each of these Turing machines:
$\Sigma M = \{a, b, c\}$. $M =$




$\{\mathtt{b}^k\mathtt{c}\mathtt{a}^k : k \geq 0\}$

REFERENCES

[1] Elaine Rich, Automata Computatibility and Complexity: Theory and Applications, Pearson, Prentice Hall, 2008.