Computability and Computational Complexity
Academic year 2019–2020, first semester
Lecture notes

Mauro Brunato

Version: 2019-10-12

Caveat Lector

The main purpose of these very schematic lecture notes is to keep track of what has been said during the lectures. Reading this document is not enough to pass the exam. You should also see the linked citations and footnotes, the additional material and the references provided on the following webpage, which will also contain the up-to-date version of these notes:

`https://comp3.eu/`

To check for new version of this document, please compare the version date on the title page the one reported on the webpage.

Changelog

2019-10-12

- Universal Turing machines, uncomputability results.
- Busy beaver functions and results.
- Turing reductions
- Self-assessment questions and exercises.

2019-10-01

- First lectures: basic definitions, Collatz example, Turing machines, computational power of Turing machines.

Contents

Part I

Lecture notes

Chapter 1

Computability

1.1 Basic definitions and examples

In computer science, every problem instance can be represented by a finite sequence of symbols from a finite alphabet, or equivalently as a natural number. In the following, let Σ denote a finite set of *symbols*. Σ will be the *alphabet* we are going to use to represent things. Pairs, triplets, n-tuples of symbols are represented by the usual cartesian product notations:

$$\Sigma^2 = \Sigma \times \Sigma = \{(s,t)|s,t \in \Sigma\}, \quad \Sigma^3 = \Sigma \times \Sigma \times \Sigma, \dots, \Sigma^n = \overbrace{\Sigma \times \Sigma \times \cdots \times \Sigma}^{n \text{ times}}$$

As a shorthand, instead of representing tuples of symbols in the formal notation $(s_1, s_2, \dots, s_n)$ we will use the simpler "string" notation $s_1 s_2 \cdots s_n$. As a particular case, let $\varepsilon = ()$ represent the empty tuple (with $n = 0$ elements). Therefore, the set of strings of length n can be defined by induction:

$$\Sigma^n = \begin{cases} \{\varepsilon\} & \text{if } n = 0 \\ \Sigma \times \Sigma^{n-1} & \text{if } n > 0. \end{cases}$$

Finally, the Kleene closure of this sequence is the set of all *finite* strings on the alphabet Σ:

$$\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n.$$

It is worthwile to note that Σ^*, while being infinite in itself, only contains *finite* sequences of symbols. Moreover, for every $n \in \mathbb{N}$, Σ^n is finite ($|\Sigma^n| = |\Sigma|^n$, where $| \cdot \|$ represents the cardinality of a set).

Our main focus will be on functions that map input strings to output strings on a given alphabet,

$$f : \Sigma^* \to \Sigma^*,$$

or in functions that map strings onto a "yes"/"no" decision set,

$$f : \Sigma^* \to \{0, 1\};$$

in such case, we talk about a *decision problem*.

Examples:

- Given a natural number n, is n prime?

- Given a graph, what is the maximum degree of its nodes?

- From a customer database, select the customers that are more than fifty years old.

- Given a set of pieces of furniture and a set of trucks, can we accommodate all the furniture in the trucks?

As long as the function's domain and codomain are finite, they can be represented as sequences of symbols, hence of bits, therefore as strings (although some representations make more sense than others); observe that some problems among those listed are decision problems, others aren't.

Decision functions and sets

There is a one-to-one correspondence between decision functions on Σ^* and subsets of Σ^*. Given $f : \Sigma^* \to \{0, 1\}$, its obvious set counterpart is the subset of strings for which the function answers 1:

$$S_f = \{s \in \Sigma^* : f(s) = 1\}.$$

Conversely, given a string subset $S \subseteq \Sigma^*$, we can always define the function that decides over elements of the set:

$$f_S(s) = \begin{cases} 1 & \text{if } s \in S \\ 0 & \text{if } s \notin S. \end{cases}$$

Given a function, or equivalently a set, we say that it is **computable**[1] (or **decidable**, or **recursive**) if and only if a procedure can be described to compute the function's outcome in a finite number of steps. Observe that, in order for this definition to make sense, we need to define what an acceptable "procedure" is; for the time being, let us intuitively consider any computer algorithm.

Examples of computable functions and sets are the following:

- the set of even numbers;
- a function that decides whether a number is prime or not;
- any finite or cofinite[2] set, and any function that decides on them;
- any function studied in a basic Algorithms course (sorting, hashing, spanning trees on graphs. . .).

1.1.1 A possibly non-recursive set

Collatz numbers

Given $n \in \mathbb{N} \setminus \{0\}$, let the *Collatz sequence* starting from n be defined as follows:

$$\begin{aligned} a_1 &= n \\ a_{i+1} &= \begin{cases} a_i/2 & \text{if } a_i \text{ is even} \\ 3a_i + 1 & \text{if } a_i \text{ is odd,} \end{cases} \qquad i = 1, 2, \dots \end{aligned}$$

In other words, starting from n, we repeatedly halve it while it is even, and multiply it by 3 and add 1 if it is odd.

The *Collatz conjecture*[3] states that every Collatz sequence eventually reaches the value 1. While most mathematicians believe it to be true, nobody has been able to prove it.

Suppose that we are asked the following question:

> "Given $n \in \mathbb{N} \setminus \{0\}$, does the Collatz sequence starting from n reach 1?"

[1] `https://en.wikipedia.org/wiki/Recursive_set`

[2] A set is *cofinite* when its complement is finite.

[3] `https://en.wikipedia.org/wiki/Collatz_conjecture`

```
function collatz ($n \in \mathbb{N} \setminus \{0\}$) $\in \{0,1\}$
    repeat
        if $n = 1$ then return 1
        if $n$ is even
            then $n \leftarrow n/2$
            else $n \leftarrow 3n+1$
    return 0
```

```
function collatz ($n \in \mathbb{N} \setminus \{0\}$) $\in \{0,1\}$
    return 1
```

Figure 1.1: Left: the only way I know to decide whether n is a Collatz number isn't guaranteed to work. Right: a much better way, but it is correct if and only if the conjecture is true.

If the answer is "yes," let us call n a *Collatz number*. Let $f : \mathbb{N} \setminus \{0\} \to \{0,1\}$ be the corresponding decision function:

$$f(n) = \begin{cases} 1 & \text{if } n \text{ is a Collatz number} \\ 0 & \text{if } n \text{ is not a Collatz number,} \end{cases} \qquad n = 1, 2, \dots$$

Then the Collatz conjecture simply states that all positive integers are Collatz numbers or, equivalently, that $f(n) = 1$ on its whole domain.

Decidability of the Collatz property

Let us consider writing a function, in any programming language, to answer the above question, i.e., a function that returns 1 if and only if its argument is a Collatz number. Figure1.1 details two possible ways to do it, and both have problems: the rightmost one requires us to have faith in an unproven mathematical conjecture; the left one only halts when the answer is 1 (the final **return** is never reached).

In more formal terms, we are admitting that we are **not** able to prove that the Collatz property is *decidable* (i.e., that there is a computer program that always terminates with the correct answer[4]). However, we have provided a procedure that terminates with the correct answer when the answer is "yes" (the function is not *total*, in the sense that it doesn't always provide an answer). We call such set **recursively enumerable**[5] (or RE, in short).

Having a procedure that only terminates when the answer is "yes" maight not seem much, but it actually allows us to enumerate all numbers having the property. The function in Fig. 1.2 shows the basic trick to enumerate a potentially non-recursive set, applied to the Collatz sequence: the **diagonal method**[6]. Rather than performing the whole decision function on a number at a time (which would expose us to the risk of an endless loop), we start by executing the first step of the decision function for the first input ($n = 1$), then we perform the second step for $n = 1$ and the first step of $n = 2$; at the i-th iteration, we perform the i-th step of the first input, the $(i-1)$-th for the second, down to the first step for the i-th input. This way, every Collatz number will eventually hit 1 and be printed out.

The naïf approach of following the table rows is not guaranteed to work, since it would loop indefinitely, should a non-Collatz number ever exist.

Observe that the procedure does not print out the numbers in increasing order.

1.2 A computational model: the Turing machine

Among the many formal definition of computation proposed since the 1930s, the Turing Machine (TM for short) is by far the most similar to our intuitive notion. A Turing Machine[7] is defined by:

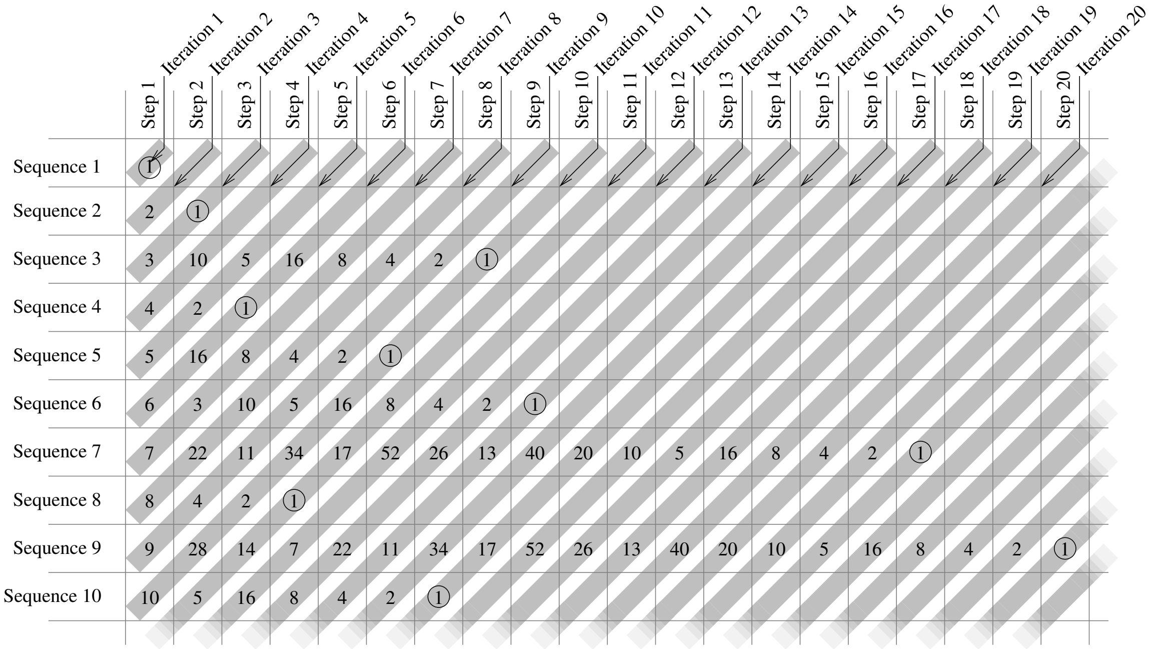
[4] To the best of my knowledge, which isn't much.

[5] `https://en.wikipedia.org/wiki/Recursively_enumerable_set`

[6] See `https://comp3.eu/collatz.py` for a Python version.

[7] `https://en.wikipedia.org/wiki/Turing_machine`

```
procedure enumerate_collatz
  queue ← []
  for $n$ ← 1 ... ∞                                   Repeat for all numbers
    queue$_n$ ← $n$                      Add n to queue with itself as starting value
    for $i$ ← 1 ... $n$:                             Iterate on all numbers up to n
      if queue$_i$ = 1                             i is Collatz, print and forget it
        print $i$
        delete queue$_i$                    deleted means "Already taken care of"
      else if queue$_i$ is not deleted   if current number wasn't printed and forgotten yet
        if queue$_i$ is even                Advance i-th sequence in the queue by one step
          then queue$_i$ ← queue$_i$ / 2
          else queue$_i$ ← 3 · queue$_i$ + 1
```



	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9	Step 10	Step 11	Step 12	Step 13	Step 14	Step 15	Step 16	Step 17	Step 18	Step 19	Step 20
Sequence 1	1																			
Sequence 2	2	1																		
Sequence 3	3	10	5	16	8	4	2	1												
Sequence 4	4	2	1																	
Sequence 5	5	16	8	4	2	1														
Sequence 6	6	3	10	5	16	8	4	2	1											
Sequence 7	7	22	11	34	17	52	26	13	40	20	10	5	16	8	4	2	1			
Sequence 8	8	4	2	1																
Sequence 9	9	28	14	7	22	11	34	17	52	26	13	40	20	10	5	16	8	4	2	1
Sequence 10	10	5	16	8	4	2	1													

Figure 1.2: Enumerating all Collatz numbers: top: the algorithm; bottom: a working schematic

- a finite alphabet Σ, with a distinguished "default" symbol (e.g., "␣" or "0") whose symbols are to be read and written on an infinitely extended tape divided into cells;
- a finite set of states Q, with a distinguished initial state and one or more distinguished halting states;
- a set of rules R, described by a (possibly partial) function that associates to a pair of symbol and state a new pair of symbol and state plus a direction:

$$R : Q \times \Sigma \to \Sigma \times Q \times \{L, R\}.$$

Initially, all cells contain the default symbol, with the exception of a finite number; the non-blank portion of the tape represent the *input* of the TM. The machine also maintains a *current position* on the tape. The machine has an initial state $q_0 \in Q$. At every step, if the machine is in state $q \in Q$, and the symbol $\sigma \in \Sigma$ appears in the current position of the tape, the machine applies the rule set R to (q, σ):

$$(\sigma', q', d) = R(q, \sigma).$$

The machine writes the symbol σ' on the current tape cell, enters state q', and moves the current position by one cell in direction d. If the machine enters one of the distinguished halting states, then the computation ends. At this point, the content of the (non-blank portion of) the tape represents the computation's *output*.

Observe that the input size for a TM is unambiguously defined: the size of the portion of tape that contains non-default symbols. Also the "execution time" is well understood: it is the number of steps before halting. Therefore, when we say that the computational complexity of a TM for inputs of size n is $T(n)$ then we mean that $T(n)$ is the worst-case number of steps that a TM performs before halting when the input has size n.

1.2.1 Examples

In order to experiment with Turing machines, many web-based simulators are available. The two top search results for "turing machine demo" are

- `http://morphett.info/turing/turing.html`
- `https://turingmachinesimulator.com/`.

Students are invited to read the simplest examples and to try implementing a TM for some simple problem (e.g., some arithmetic or logical operation on binary or unary numbers). Also, see the examples provided in the course web page.

1.2.2 Computational power of the Turing Machine

With reference to more standard computational models, such as the Von Neumann architecture of all modern computers, the TM seems very limited; for instance, it lacks any random-access capability. The next part of this course is precisely meant to convince ourselves that a TM is exactly as powerful as any other (theoretical) computational device. To this aim, let us discuss some possible generalizations.

Multiple-tape Turing machines

A k-tape Turing machine is a straightforward generalization of the basic model, with the following variations:

- the machine has k unlimited tapes, each with an independent current position;

- the rule set of the machine takes into account k symbols (one for each tape, from the current position) both in reading and in writing, and k movement directions (each current position is independent), with the additional provision of a "stay" direction S in which the position does not move:
$$R : Q \times \Sigma^k \to \Sigma^k \times Q \times \{L, R, S\}^k.$$

Multiple-tape TMs are obviously more practical for many problems. For example, try following the execution of the binary addition algorithms below:

- 1-tape addition from `http://morphett.info/turing/turing.html`: select "Load an example program/Binary addition";

- 3-tape addition from `https://turingmachinesimulator.com/`: select "Examples/3 tapes/Binary addition".

However, it turns out that any k-tape Turing machine can be "simulated" by a 1-tape TM, in the sense that it is possible to represent a k-tape TM on one tape, and to create a set of 1-tape rules that simulates the evolution of the k-tape TM. Of course, the 1-tape machine is much slower, as it needs to repeatedly scan its tape back and forth just to simulate a single step of the k-tape one.

Theorem 1 (k-tape Turing machine emulation)**.** *If a k-tape Turing machine $\mathcal{M}$ takes time $T(n)$ on inputs of time n, then it is possible to program a 1-tape Turing machine $\mathcal{M}'$ that simulates it (i.e., essentially performs the same computation) in time $O(T(n)^2)$.*

Proof. See Arora-Barak, Claim 1.9 in the public draft.

Basically, the k tapes of $\mathcal{M}$ are encoded on the single tape of $\mathcal{M}'$ by alternating the cell contents of each tape; in order to remember the "current position" on each tape, every symbol is complemented by a different version (e.g., a "hatted" symbol) to be used to mark the current position. To emulate a step of $\mathcal{M}$, the whole tape of $\mathcal{M}'$ is first scanned in order to find the k symbols in the current positions; then, a second scan is used to replace each symbol in the current position with the new symbol; then a third scan performs an update of the current positions.

Since $\mathcal{M}$ halts in $T(n)$ steps, no more that $T(n)$ cells of the tapes will ever be visited; therefore, every scan performed by $\mathcal{M}'$ will take at most $kT(n)$ steps. Given some more details, cleanup tasks and so on, the simulation of a single step of $\mathcal{M}$ will take at most $5kT(n)$ steps by $\mathcal{M}'$, therefore the whole simulation takes $5kT(n)^2$ steps. Since $5k$ is constant wrt the input size n, the result follows. □

Size of the alphabet

The number of symbols that can be written on a tape (the size of the alphabet Σ) can make some tasks easier; for instance, in order to deal with binary numbers a three-symbol alphabet ("0", "1", and the blank as a separator) is convenient, while working on words is easier if the whole alphabet is available.

While a 1-sized alphabet $\Sigma = \{\textvisiblespace\}$ is clearly unfit for a TM (no way to store information on the tape), a 2-symbol alphabel is enough to simulate any TM:

Theorem 2 (Emulation by a two-symbol Turing Machine)**.** *If a Turing machine $\mathcal{M}$ with a k-symbol alphabet Σ takes time $T(n)$ on an input of size n, then it can be simulated by a Turing machine $\mathcal{M}'$ with a 2-symbol alphabet $\Sigma' = \{0, 1\}$ in time $O(T(n))$ (i.e., with a linear slowdown).*

Proof. See Arora-Barak, claim 1.8 in the public draft, where for convenience machine $\mathcal{M}'$ is assumed to have 4 symbols and the tape(s) extend only in one direction.

Every symbol from alphabet Σ can be encoded by $\lceil \log_2 k \rceil$ binary digits from Σ'. Every step of machine $\mathcal{M}$ will be simulated by $\mathcal{M}'$ by reading $\lceil \log_2 k \rceil$ cells in order to reconstruct the current symbol in $\mathcal{M}$; the symbol being reconstructed bit by bit is stored in the machine state (therefore, $\mathcal{M}'$ requires many more states that $\mathcal{M}$). This scan is followed by a new scan to replace the encoding with the new symbol (again, all information needed by $\mathcal{M}'$ will be "stored" in its state), and a third (possibly longer) scan to place the current position to the left or right encoding. Therefore, a step of $\mathcal{M}$ will require less than $4\lceil\log_2 k\rceil$ steps of $\mathcal{M}'$, and the total simulation time will be

$$T'(n) \leq 4\lceil\log_2 k\rceil T(n).$$

□

Simulating other computational devices

Although they are very simple devices, we can convince ourselves quite easily that Turing machines can emulate a simple CPU/RAM architecture: just replace random access memory with sequential search on a tape (tremendous slowdown, but we are not concerned by it now), the CPU's internal registers can be stored in separate tapes, and every opcode of the CPU corresponds to a separate set of states of the machine. Operations such as "load memory to a register," "perform an arithmetic or logical operation between registers," "conditionally junp to memory" and so on can be emulated.

1.2.3 Universal Turing machines

The main drawback of TMs, as described up to now, with respect to our modern understanding of computational systems, is that each serves one specific purpose, encoded in its rule set: a machine to add numbers, one to multiply, and so on.

However, it is easy to see that a TM can be represented by a finite string in a finite alphabet: each transition rule can be seen as a quintuplet, each from a finite set, and the set of rules is finite. Therefore, it is possible to envision a TM $\mathcal{U}$ that takes another TM $\mathcal{M}$ as input on its tape, properly encoded, together with an input string s for $\mathcal{M}$, and simulates $\mathcal{M}$ step by step on input s. Such machine is called a Universal Turing machine (UTM).

One such machine, using a 16 symbol encoding and a single tape, is described in

`https://www.dropbox.com/sh/u7jsxm232giwown/AADTRNqjKBIe_QZGyicoZWjYa/utm.pdf`

and can be seen in action at the aforementioned link `http://morphett.info/turing/turing.html`, clicking "Load an example program / Universal Turing machine."

1.2.4 The Church-Turing thesis

We should be convinced, by now, that TMs are powerful enough to be a fair computational model, at least on par with any other reasonable definition. We formalize this idea into a sort of "postulate," i.e., an assertion that we will assume to be true for the rest of this course.

Postulate 1 (Church-Turing thesis). *Turing machines are at least as powerful as every physically realizable model of computation.*

This thesis allows us to extend every the validity negative result about TMs to every physical computational device.

1.3 Uncomputable functions

It is easy to understand that, even if we restrict our interest to decision functions, almost all functions are not computable by a TM. In fact, as the following Lemmata 1 and 2 show, there are simply too many functions to be able to define a TM for each of them.

Lemma 1. *The set of decision functions $f : \mathbb{N} \to \{0,1\}$ (or, equivalently, $f : \Sigma^* \to \{0,1\}$), is uncountable.*

Proof. By contradiction, suppose that a complete mapping exists from the naturals to the set of decision functions; i.e., there is a mapping $n \mapsto f_n$ that enumerates all functions. Define function $\hat{f}(n) = 1 - f_n(n)$. By definition, function $\hat{f}$ differs from f_n on the value it is assigned for n (if $f_n(n) = 0$ then $\hat{f}(n) = 1 - f_n(n) = 1 - 0 = 0$, and vice versa). Therefore, contrary to the assumption, the enumeation is not complete because it excluded function $\hat{f}$. □

Lemma 1 is an example of *diagonal argument*, introduced by Cantor in order to prove the uncountability of real numbers: focus on the "diagonal" values (in our case $f_n(n)$, by using the same number as function index and as argument), and make a new object that systematically differs from all that are listed.

Lemma 2. *Given a finite alphabet Σ, the number of TMs on that alphabet is countable.*

Proof. As shown in the Universal TM discussion, every TM can be encoded in some appropriate alphabet. As shown by Theorem 2, every alphabet with at least two symbols can emulate and be emulated by every other alphabet. Therefore, it is possible to define a representation of any TM in any alphabet.

We know that strings can be enumerated: first we count the only string in Σ^0, then the strings in Σ^1, then those in Σ^2 (e.g., in lexicographic order), and so on. Since every string $s \in \Sigma^*$ is finite ($s \in \Sigma^{|s|}$), sooner or later it will be enumerated. Therefore there is a mapping $\mathbb{N} \to \Sigma^*$, i.e., Σ^* is countable.

Since TMs can be mapped on a subset of Σ^* (those strings that define TMs according to the chosen encoding), and are still infinite, it follows that TMs are countable. □

Therefore, whatever way we choose to enumerate TMs and to associate them with decision functions, we will inevitably leave out some functions. Hence, given that TMs are our definition of computing,

Corollary 1. *There are uncomputable decision functions.*

1.3.1 Finding an uncomputable function

Let us introduce a little more notation. As already defined, the alphabet Σ contains a distinguished, "default" symbol, which we assume to be "␣". Before the computation starts, only a finite number of cell tapes have non-blank symbols. Let us define as "input" the smallest, contiguous set of tape cells that contains all non-blank symbols.

A Turing machine transforms an input string into an output string (the smallest contiguous set of tape cells that contain all non-blank symbols at the *end* of the computation), but it might never terminate. In other words, if we see a TM machine as a function from Σ^* to Σ^* it might not be a *total* function.

As an alternative, we may introduce a new value, ∞, as the "value" of a non-terminating computation; given a Turing machine $\mathcal{M}$, if its compuattion on input s does not terminate we will write $\mathcal{M}(s) = \infty$.

While TM encodings have a precise syntax, so that not all strings in Σ^* are syntactically valid encodings of some TM, we can just accept the convention that any such invalid string encodes the TM that immediately halts (think of s as a program, executed by a UTM that immediately stops if there is a syntax error). This way, all strings can be seen to encode a TM, and most string just encode the "identity function" (a machine that halts immediately leaves its input string unchanged). Let us therefore call $\mathcal{M}_s$ the TM whose encoding is string s, or the machine that immediately terminates if s is not a valid encoding.

With this convention in mind, we can design a function whose outcome differs from that of any TM. We employ a diagonal technique akin to the proof of Lemma 1: for any string $\alpha \in \Sigma*$, we define our function to differ from the output of the TM encoded by α on input α itself.

Theorem 3. *Given an alphabet Σ and a encoding $\alpha \mapsto \mathcal{M}_\alpha$ of TMs in that alphabet, the function*

$$UC(\alpha) = \begin{cases} 0 & \text{if } \mathcal{M}_\alpha(\alpha) = 1 \\ 1 & \text{otherwise} \end{cases} \qquad \forall \alpha \in \Sigma^*$$

is uncomputable.

Proof. Let $\mathcal{M}$ be any TM, and let $m \in \Sigma^*$ be its encoding (i.e., $\mathcal{M} = \mathcal{M}_m$). By definition, $UC(m)$ differs from $\mathcal{M}(m)$: the former outputs one if and only if the latter outputs anything else (or does not terminate).
See also Arora-Barak, theorem 1.16 in the public draft. □

What is the problem that prevents us from computing UC? While the definition is quite straightforward, being able to emulate the machine $\mathcal{M}_\alpha$ on input α is not enough to always decide the value of $UC(\alpha)$. We need to take into account also the fact that the emulation might never terminate. This allows us to prove, as a corollary of the preceding theorem, that there is no procedure that always determines whether a machine will terminate on a given input.

Theorem 4 (Halting problem)**.** *Given an alphabet Σ and a encoding $\alpha \mapsto \mathcal{M}_\alpha$ of TMs in that alphabet, the function*

$$HALT(s,t) = \begin{cases} 0 & \text{if } \mathcal{M}_s(t) = \infty \\ 1 & \text{otherwise} \end{cases} \forall (s,t) \in \Sigma^* \times \Sigma^*$$

(i.e., which returns 1 if and only if machine $\mathcal{M}_s$ halts on input t) is uncomputable.

Proof. Let's proceed by contradiction. Suppose that we have a machine $\mathcal{H}$ which computes $HALT(s,t)$ (i.e., when run on a tape containing a string s encoding a TM and a string t, always halts returning 1 if machine $\mathcal{M}_s$ would halt when run on input t, and returning 0 otherwise). Then we could use $\mathcal{H}$ to compute function UC.
For convenience, let us compute UC using a machine with two tapes. The first tape is read-only and contains the input string $\alpha \in \Sigma^*$, while the second will be used as a work (and output) tape. To compute UC, the machine will perform the following steps:

- Create two copies of the input string α onto the work tape, separated by a blank (we know we can do this because we can actually write the machine);
- Execute the machine $\mathcal{H}$ (which exists by hypothesis) on the work tape, therefore calculating whether the computation $\mathcal{M}_\alpha(\alpha)$ would terminate or not. Two outcomes are possible:
  - If the output of $\mathcal{H}$ is zero, then we know that the computation of $\mathcal{M}_\alpha(\alpha)$ wouldn't terminate, therefore, by definition of function UC, we can output 1 and terminate.
  - If, on the other hand, the output of $\mathcal{H}$ is one, then we know for sure that the computation $\mathcal{M}_\alpha(\alpha)$ would terminate, and we can emulate it with a UTM $\mathcal{U}$ (which we know to exist) and then "inverting" the result à la UC, by executing the following steps:
    - As in the first step, create two copies of the input string α onto the work tape, separated by a blank;
    - Execute the UTM $\mathcal{U}$ on the work tape, thereby emulating the computation $\mathcal{M}_\alpha(\alpha)$;
    - At the end, if the output of the emulation was 1, then replace it by a 0; if it was anything other than 1, replace it with 1.

This machine would be able to compute UC by simply applying its definition, but we know that UC is not computable by a TM; all steps, apart from $\mathcal{H}$, are already known and computable. We must conclude that $\mathcal{H}$ cannot exist.
See also Arora-Barak, theorem 1.17 in the public draft. □

This proof employs a very common technique of CS, called *reduction*: in order to prove the impossibility of $HALT$, we "reduce" the computation of UC to that of $HALT$; since we know that the former is impossible, we must conclude that the latter is too.

The Haliting Problem for machines without an input

Consider the special case of machines that do not work on an input string; i.e., the class of TMs that are executed on a completely blank tape. Asking whether a computation without input will eventually halt might seem a simpler question, because we somehow restrict the number of machines that we are considering.

Let us define the following specialized halting function:

$$HALT_\varepsilon(s) = HALT(s,\varepsilon) = \begin{cases} 0 & \text{if } \mathcal{M}_s(\varepsilon) = \infty \\ 1 & \text{otherwise} \end{cases} \forall s \in \Sigma^*.$$

It turns out that if we were able to compute $HALT_\varepsilon$ then we could also compute $HALT$:

Theorem 5. *$HALT_\varepsilon$ is not computable.*

Proof. By contradiction, suppose that there is a machine $\mathcal{H}'$ that computes $HALT_\varepsilon$. Such machine would be executed on a string s on the tape, and would return 1 if the machine encoded by s would halt when run on an empty tape, 0 otherwise.
Now, suppose that we are asked to compute $HALT(s,t)$ for a non-empty input string t. We can transform the computation $\mathcal{M}_s(t)$ on a computation $\mathcal{M}_{s'}(\varepsilon)$ on an empty tape where s' contains the whole encoding s, but prepended with a number of states that write the string t on the tape. In other words, we transform a computation on a generic input into a computation on an empty tape that writes the desired input before proceeding.
After modifying the string s into s' on tape, we can run $\mathcal{H}'$ on it. The answer of $\mathcal{H}'$ is precisely $HALT(s,t)$, which would therefore be computable. □

Again, the result was obtained by reducing a known impossible problem, $HALT$ to the newly introduced one, $HALT_\varepsilon$.

1.3.2 Recursive enumerability of halting computations

Although $HALT$ is not computable, it is clearly recursively enumerable. In fact, we can just take a UTM and modify it to erase the tape and write "1" whenever the emulated machine ends, and we would have a partial function that always accepts (i.e., returns 1) on terminating computations.

It is also possible to output all $(s,t) \in \Sigma^* \times \Sigma^*$ pairs for which $\mathcal{M}_s(t)$ halts by employing a diagonal method similar to the one used in Fig. 1.2[8].

Function $HALT$ is our first example of R.E. function that is provably not recursive.

Observe that, unlike recursivity, R.E. does *not* treat the "yes" and "no" answer in a symmetric way. We can give the following:

Definition 1. *A decision function $f : \Sigma^* \to \{0,1\}$ is* co-R.E. *if it admits a TM $\mathcal{M}$ such that $\mathcal{M}(s)$ halts with output 0 if and only if $f(s) = 0$.*

In other words, co-R.E. functions are those for which it is possible to compute a "no" answer, while the computation might not terminate if the answer is "yes". Clearly, if f is R.E., then $1 - f$ is co-R.E.

Theorem 6. *A decision function $f : \Sigma^* \to \{0,1\}$ is recursive if and only if it is both R.E. and co-R.E.*

[8] See the figure at `https://en.wikipedia.org/wiki/Recursively_enumerable_set#Examples`

Proof. Let us prove the "only if" part first. If f is recursive, then there is a TM $\mathcal{M}_f$ that computes it. But $\mathcal{M}_f$ clearly satisfies both the R.E. definition ($\mathcal{M}_f(s)$ halts with output 1 if and only if $f(s) = 1$) and the co-R.E. definition ($\mathcal{M}_f(s)$ halts with output 0 if and only if $f(s) = 0$).
About the "if" part, if f is R.E., then there is a TM M_1 such that $M_1(s)$ halts with output 1 iff $f(s) = 1$; since f is also co-R.E., then there is also a TM $\mathcal{M}_0$ such that $M_1(s)$ halts with output 0 iff $f(s) = 0$. Therefore, a machine that alternates one step of the execution of $\mathcal{M}_1$ with one step of $\mathcal{M}_0$, halting when one of the two machines halts and returning its output, will eventually terminate (because, whatever the value of f, at least one of the two machines is going to eventually halt) and precisely decides f. □

Observe that, as already pointed out, any definition given on decision functions with domain Σ^* also works on domain $\mathbb{N}$ (and on any other discrete domain), and can be naturally extended on subsets of strings or natural numbers. We can therefore define a set as recursive, recursively enumerable, or co-recursively enumerable.

Decision and acceptance

In the following, we will use the following terms when speaking of languages.

Definition 2.

- *If language S is recursively enumerable, i.e. there is a TM $\mathcal{M}$ such that $\mathcal{M}(s) = 1 \Leftrightarrow s \in S$, then we say that $\mathcal{M}$* accepts *S (or that it "recognizes" it).*
- *Given a TM $\mathcal{M}$, the language recognized by it (i.e., the set of all inputs that are accepted by the machine) is represented by $L(\mathcal{M})$.*
- *If language S is recursive, i.e. there is a TM $\mathcal{M}$ that accepts it and always halts, then we say that $\mathcal{M}$* decides *S.*

In the case of functions transforming strings, we will use the following terms.

Definition 3. *If a function $f : \Sigma^* \to \Sigma^*$ is computable, i.e. there is a TM $\mathcal{M}$ that always halts and such that $\mathcal{M}(s) = f(s)$, then we say that $\mathcal{M}$* computes *f.*

We generalize the notion to functions outside the realm of strings by considering suitable representations. E.g., a machine $\mathcal{M}$ *computes* an integer function $f : \mathbb{N} \to \mathbb{N}$ if it transforms a representation of $n \in \mathbb{N}$ (e.g., its decimal, binary or unary notation) into the corresponding representation of $f(n)$. Since all representations of integer numbers can be converted to each other by a TM, the choice of a specific one is arbitrary and does not impact on the definition. Therefore, we can resort to unary notation and say that

Theorem 7. *A function $f : \mathbb{N} \to \mathbb{N}$ is computable if and only if there is a TM $\mathcal{M}$ on alphabet $\Sigma = \{1, \textvisiblespace\}$ such that*

$$\forall n \in \mathbb{N} \quad \mathcal{M}(1^n) = 1^{f(n)}.$$

I.e., the TM $\mathcal{M}$ maps a string of n ones into a string of $f(n)$ ones.

1.3.3 Another uncomputable function: the Busy Beaver game

Since we might be unable to tell at all whether a specific TM will halt, the question arises of how complex can machine's output be for a given number of states.

Definition 4 (The Busy Beaver game)**.** *Among all TMs on alphabet $\{0, 1\}$ and with $n = |Q|$ states (not counting the halting one) that halt when run on an empty (i.e., all-zero) tape:*

- *let $\Sigma(n)$ be the largest number of (not necesssarily consecutive) ones left by any machine upon halting;*

- *let $S(n)$ be the largest number of steps performed by any such machine before halting.*

Function $\Sigma(n)$ is known as the *busy beaver* function for n states, and the machine that achieves it is called the Busy Beaver for n states.

Both functions grow very rapidly with n, and their values are only known for $n \leq 4$. The current Busy Beaver candidate with $n = 5$ states writes more than 4K ones before halting after more than 47M steps.

Theorem 8. *The function $S(n)$ is not computable.*

Proof. Suppose that $S(n)$ is computable. Then, we could create a TM to compute $HALT_\varepsilon$ (the variant with empty input) on a machine encoded in string s as follows:

```
on input $s$
    count the number $n$ of states of $\mathcal{M}_s$
    compute $\ell \leftarrow S(n)$
    emulate $\mathcal{M}_s$ for at most $\ell$ steps
    if the emulation halts before $\ell$ steps
        then $\mathcal{M}_s$ clearly halts: accept and halt
        else $\mathcal{M}_s$ takes longer than the BB: reject and halt
```

□

Observe that, by construction, $\Sigma(n) \leq S(n)$ (a TM cannot write more than a symbol per step). The next result is even stronger. Given two functions $f, g : \mathbb{N} \to \mathbb{N}$, we say that f "eventually outgrows" g, written $f >_E g$, if $f(n) \geq g(n)$ for a sufficiently large value of n:

$$f >_E g \Leftrightarrow \exists N : \forall n > N f(n) \geq g(n).$$

Theorem 9. *The function $\Sigma(n)$ eventually outgrows any computable function.*

Proof. Let $f : \mathbb{N} \to \mathbb{N}$ be computable. Let us define the following function:

$$F(n) = \sum_{i=0}^{n} \bigl[f(i) + i^2\bigr].$$

By definition, F clearly has the following properties:

$$F(n) \geq f(n) \quad \forall n \in \mathbb{N}, \tag{1.1}$$

$$F(n) \geq n^2 \quad \forall n \in \mathbb{N}, \tag{1.2}$$

$$F(n+1) > F(n) \quad \forall n \in \mathbb{N} \tag{1.3}$$

the latter because $F(n + 1)$ is equal to $F(n)$ plus a strictly positive term. Moreover, since f is computable, F is computable too. Suppose that M_F is a TM on alphabet $\{0, 1\}$ that, when positioned on the rightmost symbol of an input string of x ones and executed, outputs a string of $F(x)$ ones (i.e., computes the function $x \mapsto F(x)$ in unary representation) and halts below the rightmost one. Let C be the number of states of M_F.

Given an arbitrary integer $x \in \mathbb{N}$, we can define the following machine $\mathcal{M}$ running on an initially empty tape (i.e., a tape filled with zeroes):

- Write x ones on the tape and stop at the rightmost one (i.e., the unary representation of x: it can be done with x states, see Exercise 2 at page 22);

- Execute M_F on the tape (therefore computing $F(x)$ with C states);

- Execute M_F again on the tape (therefore computing $F(F(x))$ with C more states).

The machine $\mathcal{M}$ works on alphabet $\{0,1\}$, starts with an empty tape, ends with $F(F(x))$ ones written on it and has $x+2C$ states; therefore it is a busy beaver candidate, and the $(x+2C)$-state busy beaver must perform at least as well:

$$\Sigma(x+2C) \geq F(F(x)). \tag{1.4}$$

Now,

$$F(x) \geq x^2 >_E x+2C;$$

the first inequality comes from (1.2), while the second stems from the fact that x^2 eventually dominates any linear function of x. By applying F to both the left- and right-hand sides, which preserves the inequality sign because of (1.3), we get

$$F(F(x)) >_E F(x+2C). \tag{1.5}$$

By concatenating (1.4), (1.5) and (1.1), we get

$$\Sigma(x+2C) \geq F(F(x)) >_E F(x+2C) \geq f(x+2C).$$

Finally, by replaxing $n = x+2C$, we obtain

$$\Sigma(n) >_E f(n).$$

□

This proof is based on the original one by Rado (1962)[9].

1.3.4 Reductions

Note that a few results in the past sections (Theorems 4, 5 and 8) made use of similar arguments: "If A were computable, then we could use it to solve B; however, we know that B is uncomputable, therefore A is too." Now we want to formalize such reasoning scheme.

Definition 5. *Let $L_1 \subset \Sigma_1^*$ and $L_2 \subset \Sigma_2^*$ be two languages (on possibly different alphabets). A function*

$$f : \Sigma_1^* \to \Sigma_2^*$$

is said to be a reduction *from L_1 to L_2 if*

$$\forall s \in \Sigma_1^* \quad s \in L_1 \Leftrightarrow f(s) \in L_2.$$

Basically, we can use a reduction to transform the question "Does s belong to L_1?" into the equivalent question "Does $f(s)$ belong to L_2?"

Clearly, to be useful in computability results, f has to be computable (meaning, as usual, that there is a TM $\mathcal{M}_f$ that computes f).

Definition 6. *We say that $f : \Sigma_1^* \to \Sigma_2^*$ is a* Turing reduction *from $L_1 \subset \Sigma_1^*$ to $L_2 \subset \Sigma_2^*$ if it is a reduction from L_1 to L_2 and it is computable.*

If f is a reduction from L_1 to L_2 we write $L_1 <_f L_2$. In general, if there is a Turing reduction from L_1 to L_2 we say that L_1 is Turing reducible to L_2 and write $L_1 <_T L_2$.

Note that we do *not* require f to have any specific property such as being injective or surjective: just that it "does its work" by transforming any element of L_1 into an element of L_2 and every string that is not in L_1 into a string that is not in L_2.

All computability proofs by reduction follow one of the schemes listed in the following theorem:

[9] See for instance:
`http://computation4cognitivescientists.weebly.com/uploads/6/2/8/3/6283774/rado-on_non-computable_functions.pdf`

Theorem 10. *Let languages L_1 and L_2 and function f be such that $L_1 <_f L_2$; then*

1. *if L_2 is decidable and f is computable, then L_1 is decidable too;*
2. *if L_1 is undecidable and f is computable, then L_2 is undecidable too;*
3. *lf L_1 is undecidable and L_2 is decidable, then f is uncomputable.*

Proof. The first point is proven by showing that, if we have a machine for f and a machine for L_2 we can build a machine for L_1. Let $\mathcal{M}_{L_2}$ be a TM that decides L_2, and let $\mathcal{M}_f$ be a TM that computes f. Then the machine $\mathcal{M}$ that concatenates an execution of $\mathcal{M}_f$ and an execution of $\mathcal{M}_{L_2}$, i.e. computes $\mathcal{M}(s) = \mathcal{M}_{L_2}\big(\mathcal{M}_f(s)\big)$, decides L_1 by definition of f.

The other two points follow by contradiction.

□

In other words, by writing $L_1 <_T L_2$ we mean that L_1 is "less uncomputable" than L_2.

Observe that the proofs of Theorems 4 and 5 follow the second scheme of Theorem 10, while the proof of Theorem 8 follows the third scheme, where the function $S(n)$ is part of the reduction.

Part II

Questions and exercises

Appendix A

Self-assessment questions

This chapter collects a few questions that students can try answering to assess their level of preparation.

A.1 Computability

A.1.1 Recursive and recursively enumerable sets

1. Why is every finite set recursive?
   (Hint: we need to check whether s is in a finite list)

2. Try to prove that if a set is recursive, then its complement is recursive too.
   (Hint: invert 0 and 1 in the decision function's answer)

3. Let S be a recursively enumerable set, and let algorithm $\mathcal{A}$ enumerate all elements in S. Prove that, if $\mathcal{A}$ lists the elements of S in increasing order, then S is recursive.
   (Hint: what if $n \notin S$? Is there a moment when we are sure that n will never be listed by $\mathcal{A}$?)

A.1.2 Turing machines

1. Why do we require a TM's alphabet Σ and state set Q to be finite, while we accept the tape to be infinite?

2. What is the minimum size of the alphabet to have a useful TM? What about the state set?

3. Try writing machines that perform simple computations or accept simply defined strings.

shapes.misc,shadows,positioning,automata

Appendix B

Exercises

Preliminary observations

Since the size of the alphabet, the number of tapes or the fact that they are infinite in one or both directions have no impact on the capabilities of the machine and can emulate each other, unless the exercise specifies some of these details, students are free to make their choices.

As for accepting or deciding a language, many conventions are possible. The machine may:

- erase the content of the tape and write a single “1” or “0”;
- write “1” or “0” and then stop, without bothering to clear the tape, with the convention that acceptance is encoded in the last written symbol;
- have two halting states, `halt-yes` and `halt-no`;
- any other unambiguous convention;

with the only provision that the student writes it down in the exercise solution.

Exercise 1

For each of the following classes of Turing machines, decide whether the halting problem is computable or not. If it is, outline a procedure to compute it; if not, prove it (usually with with a reduction from the general halting problem). Unless otherwise stated, always assume that the non-blank portion of the tape is bounded, so that the input can always be finitely encoded if needed.
1.1) TMs with 2 symbols and at most 2 states (plus the halting state), starting from an empty (all-blank) tape.
1.2) TMs with at most 100 symbols and 1000000 states.
1.3) TMs that only move right;
1.4) TMs with a circular, 1000-cell tape.
1.5) TMs whose only tape is read-only (i.e., they always overwrite a symbol with the same one);

Hint — *Actually, only one of these cases is uncomputable. . .*

Solution 1

The following are minimal answers that would guarantee a good evaluation on the test.
1.1) The definition of the machine meet the requirements for the Busy Beaver game; Since we know the BB for up to 4 states, it means that every 2-state, 2-symbol machine has been analyzed on an empty tape, and its behavior is known. Therefore the HP is computable for this class of machines.
1.2) As we have seen in the lectures, 100 symbols and 1,000,000 states are much more than those needed to build a universal Turing machine $\mathcal{U}$. If this problem were decidable by a machine, say $\mathcal{H}_{1,000,000}$, then we could solve the general halting problem "does $\mathcal{M}$ halt on input s" by asking $\mathcal{H}_{1,000,000}$ whether $\mathcal{U}$ would halt on input (M, s) or not. In other words, we could reduce the general halting problem to it, therefore it is undecidable.
1.3) If the machine cannot visit the same cell twice, the symbol it writes won't have any effect on its future behavior. Let us simulate the machine; if it halts, then we output 1. Otherwise, sooner or later the machine will leave on its left all non-blank cells of the tape: from now on, it will only see blanks, therefore its behavior will only be determined by its state. Take into account all states entered after this moment; as soon as a state is entered for the second time, we are sure that the machine will run forever, because it is bound to repeat the same sequence of states over and over, and we can interrupt the simulation and output 0; if, on the other hand, the machine halts before repeating any state, we output 1.
1.4) As it has a finite alphabet and set of states (as we know from definition), the set of possible configurations of a TM with just 1000 cells is fully identified by (i) the current state, (ii) the current position, and (iii) the symbols on the tape, for a total of $|Q| \times 1000 \times |\Sigma|^1000$ configurations. While this is an enormous number, a machine running indefinitely will eventually revisit the same configuration twice. So we just need to simulate a run of the machine: as soon as a configuration is revisited, we can stop simulating the machine and return 0. If, on the other hand, the simulation reaches the halt state, we can return 1.
1.5) Let $n = |Q|$ be the number of states of the machine. Let us number the cells with consecutive integer numbers, and consider the cells a and b that delimit the non-null portion of the tape. Let us simulate the machine. If the machine reaches cell $a-(n+1)$ or $b+n+1$, we will know that the machine must have entered some state twice while in the blank portion, therefore it will go on forever: we can stop the simulation and return 0. If, on the other hand, the machine always remains between cell $a-n$ and $b+n$, then it will either halt (then we return 1) or revisit some already visited configuration in terms of current cell and state; in such case we know that the machine won't stop because it will deterministically repeat the same steps over and over: we can then stop the simulation and return 0.

Exercise 2

2.1) Complete the proof of Theorem 9 by writing down, given a positive integer n, an n-state Turing machine on alphabet $\{0,1\}$ that starts on an empty (i.e., all-zero) tape, writes down n consecutive ones and halts below the rightmost one.
2.2) Test it for n=3.

Solution 2

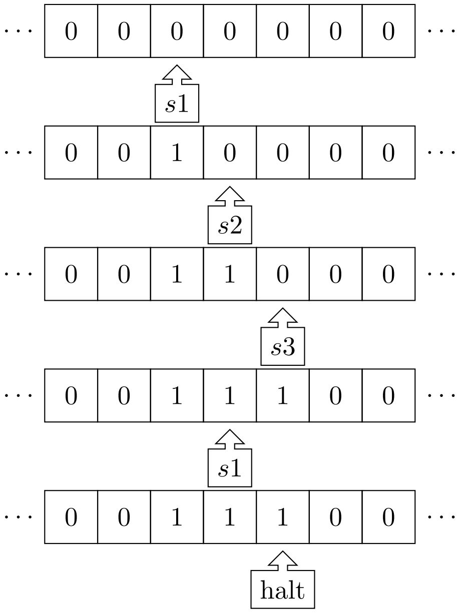
2.1) Here is a possible solution:

	0	1
s_1	1, right, s_2	1, right, halt
s_2	1, right, s_3	—
	$\vdots$	
s_i	1, right, s_{i+1}	—
	$\vdots$	
s_{n-1}	1, right, s_n	—
s_n	1, left, s_1	—

Entries marked by "—" are irrelevant, since they are never used. Any state can be used for the final move.
2.2) For $n = 3$, the machine is

	0	1
s_1	1, right, s_2	1, right, halt
s_2	1, right, s_3	—
s_3	1, left, s_1	—

Here is a simulation of the machine, starting on a blank (all-zero) tape:

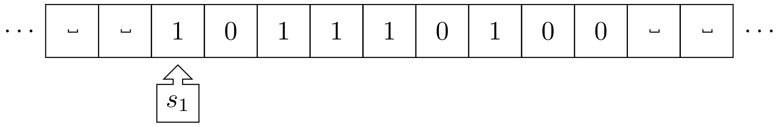


Exercise 3

3.1) Write a Turing machine according to the following specifications:

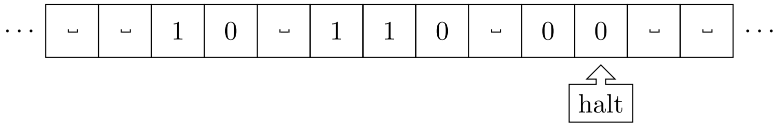
- the alphabet is $\Sigma = \{␣, 0, 1\}$, where '␣' is the default symbol;
- it has a single, bidirectional and unbounded tape;
- the input string is a finite sequence of symbols in $\{0, 1\}$, surrounded by endless '␣' symbols on both sides;
- the initial position of the machine is on the leftmost symbol of the input string;
- every '1' that immediately follows '0' must be replaced with '␣' (i.e., every sequence '01' must become '0␣').
- the final position of the machine is at the righmost symbol of the output sequence.

For instance, in the following input case



··· ␣ ␣ 1 0 1 1 1 0 1 0 0 ␣ ␣ ···

the final configuration should be



··· ␣ ␣ 1 0 ␣ 1 1 0 ␣ 0 0 ␣ ␣ ···

You can assume that there is at least one non-'␣' symbol on the tape, but considering the more general case in which the input might be the empty string is a bonus.

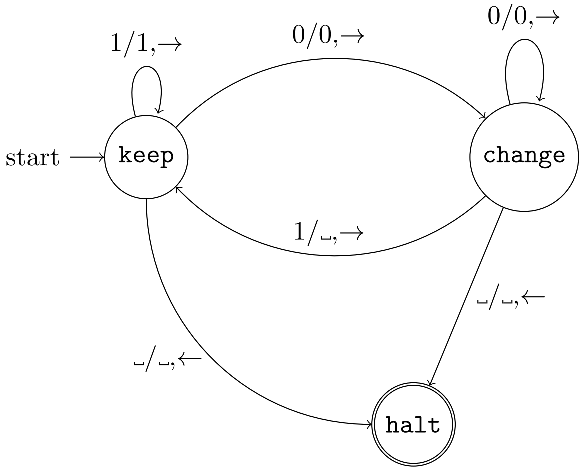
3.2) Show the sequence of steps that your machine performs on the input

"010011000111"

Solution 3

Two possible representations of the Turing machine are shown below; many other representations and transition rule sets are possible.

	␣	0	1
`keep`	␣/←/`halt`	0/→/`change`	1/→/`keep`
`change`	␣/←/`halt`	0/→/`change`	␣/→/`keep`


1/1,→
0/0,→
0/0,→
start
keep
change
1/␣,→
␣/␣,←
␣/␣,←
halt

Exercise 4

Let $\mathcal{M}$ represent a Turing Machine, let there be an encoding $s \to \mathcal{M}_s$ mapping string $s \in \Sigma^*$ to the TM $\mathcal{M}_s$ encoded by it. Finally, remember that in our notation $\mathcal{M}(x) = \infty$ means "$\mathcal{M}$ does not halt when executed on input x". Consider the following languages:

$$\begin{aligned}
L_1 &= \{s \in \Sigma^* \,|\, \exists x \mathcal{M}_s(x) \neq \infty\} = \{s \in \Sigma^* \,|\, \mathcal{M}_s \text{ halts on some inputs}\} \\
L_2 &= \{s \in \Sigma^* \,|\, \forall x \mathcal{M}_s(x) \neq \infty\} = \{s \in \Sigma^* \,|\, \mathcal{M}_s \text{ halts on all inputs}\} \\
L_3 &= \{s \in \Sigma^* \,|\, \exists x \mathcal{M}_s(x) = \infty\} = \{s \in \Sigma^* \,|\, \mathcal{M}_s \text{ doesn't halt on some inputs}\} \\
L_4 &= \{s \in \Sigma^* \,|\, \forall x \mathcal{M}_s(x) = \infty\} = \{s \in \Sigma^* \,|\, \mathcal{M}_s \text{ doesn't halt on any input}\}
\end{aligned}$$

4.1) Provide examples of TMs $\mathcal{M}_1, \dots, \mathcal{M}_4$ such that $\mathcal{M}_1 \in L_1, \dots, \mathcal{M}_4 \in L_4$.
4.2) Describe the set relationships between the four languages (i.e., which languages are subsets of others, which are disjoint, which have a non-empty intersection).

Solution 4

Observe that this exercise has very little to do with computability; however, being able to understand and answer it is a necessary prerequisite to the course. **4.1)** The machine that immediately halts ($s_0 =$ `HALT`) is an example for L_1 and L_2. The machine that never halts (e.g., always moving right and staying in state s_0) is an example for L_3 and L_4.
4.2) If a machine always halts, it clearly halts on some inputs; therefore, $L_2 \subset L_1$ (equality is ruled out by the fact that there are machines that halt on some inputs and don't on others: $L_1 \cap L_3 \neq \emptyset$). With similar considerations, we can say that $L_4 \subset L_3$.
L_2 is disjoint from both L_3.
Also, observe that $L_2 = L_1 \setminus L_3$ and $L_4 = L_3 \setminus L_1$.
The relationship among the sets can be shown in the following diagram: