CS 3719 (Theory of Computation and Algorithms) – Definitions and Turing machines Lectures 2-5

Antonina Kolokolova [*]

January 7, 2015

1 Definitions: what is a computational problem?

Now that we looked at examples of computational problems, let us formalize what we mean by a such a problem. Think about a problem as a set of inputs to an algorithm, for which it produces some output. For example, given an integer, find its prime factorization; or, given a graph G and two vertices s and t, find a shortest path from s to t in G. First assumption that we will make is that inputs to all problems will be represented by finite strings.

Definition 1 *Let Σ (pronounced "sigma", written the same as the summation sign) be a finite alphabet of symbols, By Σ^* (pronounced "sigma star") we mean the set of all finite strings consisting of elements of Σ, including the empty string ϵ (epsilon). By a* language *over Σ we mean a set of strings $L \subseteq \Sigma^*$. For a string $x \in \Sigma^*$, we denote the length of x by $|x|$.*

For example, $\{0,1\}^*$ is a set (also a language) of all binary strings:

$$\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}.$$

We can use some of these strings to denote numbers, by saying that string "0" encodes number 0, and for any other number, there is a string starting with 1 corresponding to its binary representation (i.e., one is encoded as 1, two is encodes as 10, three is encoded as 11, and so on). So the language

$$NUMBERS = \{0, 1, 10, 11, 100, 101, \dots\}.$$

[*]The material in this set of notes came from many sources, in particular "Introduction to Theory of Computation" by Sipser and course notes of U. of Toronto CS 364. Special thank you to Richard Bajona for sharing his scribed lecture notes.

And NUMBERS $\subset \{0,1\}^*$. Note that computationally it is a very computationally easy language. It takes only two time steps to determine whether a given string codes a number. If a string starts with 1, or is a single symbol "0", then it is a number, otherwise not.

For much of the course we will assume that our inputs are binary strings. But in most problems we consider the inputs are more complicated – graphs, pairs of numbers and so on. In this case, we can still encode these objects as binary strings without losing much space. For example, a pair of numbers (x, y) could be encoded very easily by making a binary string in which odd digits are the digits of the numbers, and even digits are, say, 1 except the two digits 00 used as a delimiter between the numbers (so $101, 011$ can be encoded as 11011100011111). As another example, graphs can be encoded as binary strings representing the adjacency matrix of a graph; usually for convenience when encoding graphs the matrix is preceded by the number of vertices in unary, then 0, then the matrix. So an undirected graph on 4 vertices with edges $(1,3),(2,4),(3,4)$ becomes 111100010000110010110. One more note about the encoding: the same problem can be encoded in multiple different ways (for example, we could have used a different order of vertices in the graph, or a more compact pairing function such as Cantor's pairing function for numbers). We usually imply that our algorithm knows precisely the kind of encoding of a problem it is getting as an input. One more note: we could have encoded the input in unary (that is, a number, say, 5 would be represented as 11111, rather than 101); but it would make our encoding exponentially longer to write down. You can check for yourself that going from base 2 to any other constant basis does not change the length of the encoding that much.

A *decision* (or *membership*) problem is a function from strings to Boolean values $f : \Sigma^* \to \{yes, no\}$.

For example, "given integer n, is n prime?"

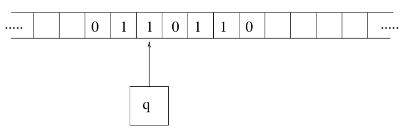
A *language* associated with a given decision problem f is the set $\{x \in \Sigma^* \mid f(x) = yes\}$. For example, `PRIME`$=\{x \in \mathbb{N} \mid x \text{ is a prime number }\}$.

A *function* problem is a function from strings to strings $f : \Sigma^* \to \Sigma^*$.

In this course, we will mainly study models of computation that compute decision problems (in most cases the complexity of solving decision and function problems is very similar).

2 Turing machines

In 1936, Alan Turing gave the first definition of an algorithm, in the form of (what we call today) a Turing machine. He wanted a definition that was simple, clear, and sufficiently general. Although Turing was only interested in defining the notion of "algorithm", his model is also good for defining the notion of "efficient" (polynomial-time) algorithm.



A Turing machine consists of an infinite tape and a finite state control. The tape is divided up into squares, each of which holds a symbol from a finite tape alphabet that includes the blank symbol $\not b$. Some definitions give two-way infinite tape; Sipser's book uses tape infinite to the right (so it has a start cell, but no end cell).

The machine has a read/write head that is connected to the control, and that that scans squares on the tape. Depending on the state of the control and symbol scanned, it makes a move, consisting of

- printing a symbol
- moving the head left or right one square
- assuming a new state

The tape will initially be completely blank, except for an input string over the finite input alphabet Σ; the head will initially be pointing to the leftmost symbol of the input.

A Turing machine M is specified by giving the following:

- Σ (a finite input alphabet).
- Γ (a finite tape alphabet). $\Sigma \subseteq \Gamma$. $\not b \in \Gamma - \Sigma$.
- Q (a finite set of states). There are 3 special states:
  - q_0 (the initial state)
  - q_{accept} (the state in which M halts and accepts)
  - q_{reject} (the state in which M halts and rejects)
- $\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ (the transition function)
  (If $\delta(q, s) = (q', s', h)$, this means that if q is the current state and s is the symbol being scanned, then q' is the new state, s' is the symbol printed, and h is either L or R, corresponding to a left or right move by one square.

The Turing machine M works as follows on input string $x \in \Sigma^*$.
Initially x appears on the tape starting from its left end, with infinitely many blanks to the right, and with the head pointing to the leftmost symbol of x. (If x is the empty string, then the tape is completely blank and the head is pointing to the leftmost square.) The control is initially in state q_0.
M moves according to the transition function δ.

M may run forever, but if it halts, then it halts either in state q_{accept} or q_{reject}. (We will mainly be interested in whether M accepts or rejects; there are definitions of what it means for a Turing machine to output a string: e.g., the output is what is left on the tape when the machine halts)

Definition 2 M accepts *a string $x \in \Sigma^*$ if M with input x eventually halts in state q_{accept}. We write $\mathcal{L}(M) = \{x \in \Sigma^* | M$ accepts $x\}$, and we refer to $\mathcal{L}(M)$ as the language accepted by M.*

(Notice that we are abusing notation, since we refer both to a Turing machine accepting a string, and to a Turing machine accepting a language, namely the set of strings it accepts.)

Example 1 The language NUMBERS described above can be recognized by a Turing machine in just two steps. From initial state q_0, if the symbol is 1 then accept (first symbol is 1 means it is a valid number). From q_0, if the symbol is blank, then reject (empty string is not a number). And if the symbol is 0, need to check if the symbol after is a blank. For that, as we are now looking for a different condition (blank is good, others are bad), let's use another state q_1. So $(q_0, 0) \to (q_1, 0, L)$. Here, it does not really matter what the machine writes on the tape, but say it wants to preserve what is there. Now, from q_1 on blank go to q_{accept}, and from q_1 on 0 or 1 go to q_{reject}.

Here, $\Sigma = \{0,1\}$, $\Gamma = \{0, 1, \not b\}$, $Q = \{q_0, q_1, q_{accept}, q_{reject}\}$ and δ is defined by the following table. Here, it does not really matter where to move and what to write when entering q_{accept} and q_{reject} states.

State q	*Symbol s*	*Action $\delta(q,s)$*
q_0	0	$(q_1, 0, R)$
q_0	1	$(q_{accept}, 1, R)$
q_0	$\not b$	$(q_{reject}, \not b, R)$
q_1	0	$(q_{reject}, 0, R)$
q_1	1	$(q_{reject}, 1, R)$
q_1	$\not b$	$(q_{accept}, \not b, R)$

Example 2 PARITY= the set of all binary strings that have an even number of 1s = $\{x | x \in \{0,1\}^*, x$ has even number of occurrences of symbol $1\}$.

A very simple Turing machine recognizes the set of such strings in time linear in the number of symbols in the strings. For that, it moves once through the string from left to right, at each point remembering whether the number of 1s it has seen so far is even or odd. If by the time it reaches a blank symbol it has seen an even number of 1s, the Turing machine accepts, and if it has seen an odd number of 1s, it rejects.

State q	*Symbol s*	*Action $\delta(q,s)$*
q_0	0	$(q_0, \not{b}, R)$
q_0	1	$(q_1, \not{b}, R)$
q_0	$\not{b}$	$(q_{accept}, \not{b}, L)$
q_1	0	$(q_1, \not{b}, R)$
q_1	1	$(q_0, \not{b}, R)$
q_1	$\not{b}$	$(q_{reject}, \not{b}, L)$

A more complicated example is a Turing machine recognizing the set of palindromes.

Example 3 PAL = the set of even length palindromes = $\{yy^r | y \in \{0,1\}^*$, where y^r means y spelled backwards.

We will design a Turing machine M that accepts the language PAL$\subseteq \{0,1\}^*$. M will have input alphabet $\Sigma = \{0,1\}$, and tape alphabet $\Gamma = \{0,1,\not{b}\}$. (Usually it is convenient to let Γ have a number of extra symbols in it, but we don't need to for this simple example.) We will have state set $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_{accept}, q_{reject}\}$.

If, in state q_0, M reads 0, then that symbol is replaced by a blank, and the machine enters state q_1; the role of q_1 is to go to the right until the first blank, remembering that the symbol 0 has most recently been erased; upon finding that blank, M enters state q_2 and goes left one square looking for symbol 0; if 0 is not there then we *reject*, but if 0 is there, then we erase it and enter state q_3 and go left until the first blank; we then goes right one square, enter state q_0, and continue as before.

If we read 1 in state q_0, then we operate in a manner similar to that above, using states q_4 and q_5 instead of q_1 and q_2.

If we read $\not{b}$ in state q_0, this means that all the characters of the input have been checked, and so we accept.

2.1 Multi-tape Turing machines

Definition 3 *A k-tape Turing machine has k tapes and k heads (although still only one control unit). Thus a transition function is $\delta: (Q - \{q_{accept}, a_{reject}) x\Gamma^k \to \{Q, \Gamma^k, \{L,R\}^k\}$.*

Palindrome can be solved faster on a two-tape Turing machine by the following algorithm. Here, a Turing machine has two tapes (and two heads), and it starts with its input written on the first tape, and both heads at the leftmost position. When we write the transition function, list the state, then the symbols being read at the first and second tape, respectively, then the direction of movement for the first and second tape.

1) Copy the input to the second tape: $\{q_0, 0, \not b\} \rightarrow \{q_0, 0, 0, R, R\}$ and the same for 1s.

2) Move the first head to the beginning of the tape; leave the second head at the end of the input (to imitate a head staying at one place, use two movements, one left and one right).

3) Now start moving first head right and the second left, comparing the symbols they are looking at. If at any moment they look at different symbols, reject, otherwise accept when the first head sees the blank.

We now formally define the notion of worst case time complexity of Turing machines.

Let M be a Turing machine over input alphabet Σ. For each $x \in \Sigma^*$, let $t_M(x)$ be the number of steps required by M to *halt* (i.e., terminate in one of the two final states) on input x. (Each step is an execution of one instruction of the machine, and we define $t_M(x) = \infty$ if M never halts on input x.)

Definition 4 (Worst case time complexity of M) *The worst case time complexity of M is the function $T_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ defined by*

$$T_M(n) = \max\{t_M(x) \mid x \in \Sigma^*, |x| = n\}.$$

If T_M is polynomial in $|x|$, we say that the Turing machine runs in polynomial time; if it is linear in $|x|$, then in linear time.

Church-Turing thesis

Church-Turing thesis: A Turing machine captures exactly the intuitive notion of an algorithm. That is, anything that can be algorithmically solved (in the intuitive sense of "algorithmically") can also be solved by a Turing machine.

An extension of the Church-Turing thesis is as follows:

Extended Church–Turing Thesis *Everything* efficiently *computable on a physical computer (in the intuitive sense) is* efficiently *computable by a Turing machine (in the formal sense).*

Quantum computers pose a potential challenge to this thesis, but they haven't been built yet.