Computability Theory

Based on lectures by Gadi Aleksandrowicz

Notes taken by yehelip

Winter 2025

These notes are not endorsed by the lecturers. I have revised them outside lectures to incorporate supplementary explanations, clarifications, and material for fun. While I have strived for accuracy, any errors or misinterpretations are most likely mine.

Contents

1 Turing Machines

The undecidability problem: Does there exist a "definite method" that, when given any possible statement in mathematics, can decide whether that statement is true or false?

Alan Turing set out with the goal of showing that the answer to this problem is no. First, in order to prove that there is not algorithm capable of determining whether a mathematical statement is false, we need to formally define what an algorithm is, and once we have a sensible definition, it could be shown that no algorithm under this definition satisfies the undecidablity problem.

In order to mimic the ability to follow algorithms, we find useful considering the way we solve problems ourselves. We can notice, that when we are faced with a problem, in order to solve it we may need to write stuff down, we may need to read stuff, and sometimes we may need to perform certain operations with the data we gathered. These different operations can be interpreted as states of thinking, and the model we are about to define models exactly what we described so far.

Definition 1.1 (Turing machine)**.** A Turing machine is a 7-tuple $M = (Q, \Gamma, b, \Sigma, \delta, q_0, F)$ such that:

- Q is a finite, nonempty set. The elements of Q are called states;
- $q_0 \in Q$ is called the initial state;
- $F \subseteq Q$ is a set of final states, or accepting states;
- Γ is a finite, nonempty set such that $Q \cap \Gamma = \emptyset$. The elements of Γ are called the tape alphabet symbols;
- $\Sigma \subsetneq \Gamma$ is called the set of input symbols;
- $b \in \Gamma \setminus \Sigma$ is called the blank symbol;
- $\delta \colon (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$ is called the transition function.

In order to define a computation on a Turing machine we first have to define a configuration (a momentary state) of the machine.

Definition 1.2 (Configuration)**.** A configuration (a momentary state) of a Turing machine M is a triple $C = (\alpha, q, i)$ such that:

- $\alpha \in \Gamma^*$ is a finite string of letters from Γ that is called the tape content;
- $q \in Q$ is a state of M called the current state of the computation;
- $i \in \mathbb{N}$ is a natural number that is called the head's position.

Additionally, the initial configuration of M on input x is the configuration $(x, q_0, 0)$.

Definition 1.3 (Finite configuration)**.** A finite configuration is any configuration (α, q, i) such that $q \in F$.

Definition 1.4 (Computational step)**.** A computational step of a Turing machine M is a change from configuration (α, q, i) to configuration (β, p, j). A computational step is said to be legal if $\delta(q, \alpha[i]) = (p, \gamma, X)$ such that

- The head moved according to X:

$$j = \begin{cases} i+1, & X = R \\ \max\{0, i-1\} & X = L \text{ (the head can't move left at the beginning of the tape)} \\ i, & X = S \end{cases}.$$

- We inserted γ correctly as such:

$$\beta = \begin{cases} \alpha_0 \cdots \alpha_{i-1}\gamma\alpha_{i+1}\cdots\alpha_n, & i < n \quad \text{or} \quad i = n \text{ and } X \neq R \\ \alpha_0 \cdots \alpha_{n-1}\gamma b, & i = n \text{ and } X = R \end{cases}.$$

Remark 1.1. Let C, C' are two configurations. We denote $C \vdash C'$ if a computational step from C takes us to C'. In this case we say that C' is the succeeding computational step of C.

Definition 1.5 (Computation)**.** A computation (or a run) of a Turing machine M on input x is a sequence of configurations $C_0, C_1, C_2, \dots$ such that

- C_0 is the initial configuration of M on x.
- For all i, if there exists a configuration C such that $C_i \vdash C$, then $C_{i+1} = C$.

If in a computation there exists n such that C_n is a final configuration we say that the computation end and the machine halts.

Definition 1.6 (Output)**.** Let M be a machine that halts on a configuration $C = (\alpha, q, i)$. The output of the computation is $\alpha_0\alpha_1 \cdots \alpha_{i-1}$, that is, all of the characters to the left of the head. If there are no such characters we say that the output is ε.

Remark 1.2. The output of a computation is ε if and only if the head's position is 0 at the end of the computation.

Remark 1.3. If the computation of M on x is infinite, we say that M does not halt on x, and the output of M on x is undefined.

This makes a Turing machine some kind of function, that accepts tapes as input, and outputs finite strings.

Definition 1.7 (Functions computed by Turing machine)**.** Let M be a Turing machine. The function that M computes is the function $f_M \colon \Sigma^* \to \Gamma^*$ defined as such:

- If the output of M on x is y then $f_M(x) = y$.
- If the output of M on x is undefinedm then $f_M(x)$ is undefined (this is sometimes denoted as $f_M(x) = \perp$).

Definition 1.8 (Total function)**.** A function f_M computed by a Turing machine M is said to be total if $f_M \colon \Sigma^* \to \Gamma^*$ is defined on all of its domain. That is, M halts for any input x.

Example 1.1. We want to construct the Turing machine that computes the function $f(x) = 0x$. In other words, the machine returns our input preceded by the letter 0. We will define the machine $M = (Q, q_0, F, \Gamma, \Sigma, b, \delta)$ as such:

$$Q = \{q_0, q_1, q_f\} \quad \text{and} \quad F = \{q_f\} \quad \text{and} \quad \Gamma = \{0, 1, b\} \quad \text{and} \quad \Sigma = \{0, 1\}$$

and define the transition function δ as such:

	0	1	b
q_0	$(q_0, 0, R)$	$(q_1, 0, R)$	$(q_f, 0, R)$
q_1	$(q_0, 1, R)$	$(q_1, 1, R)$	$(q_f, 1, R)$

Structural induction can be used to prove correctness, but it will be ommited.

1.1 Equivalence of models

Now that we have defined computations on a Turing machine, we have the ability to prove statements about the computability of all algorithms. In order to describe algorithms, depeding on the algorithm we may find it easier to implement it with a different computation model. For these cases we consider different models equivalent to the Turing machine.

Example 1.2. We define M to be a fast Turing machine just like a normal Turing machine, but with the ability to move two steps if wanted.

In order to prove the model of the fast Turing machine is equivalent to the Turing machine, we need to show that

(1) for any Turing machine M there exists a fast Turing machine such that $f_M = f_{M'}$.

(2) for any fast Turing machine M there exists a Turing machine such that $f_M = f_{M'}$.

Proving (1) is rather easy because we can ignore the additional functionality of the Fast turing machine when creating it.

In order to prove (2) there's more work to do. Let $M = (Q, q_0, F, \Gamma, \Sigma, b, \delta)$ be a fast Turing machine. We define a new Turing machine $M = (Q', q_0, F, \Gamma, \Sigma, b, \delta')$ such that for

$$Q_R = \left\{q^R\right\} q \in Q$$
$$Q_L = \left\{q^L\right\} q \in Q$$

we have $Q' = Q \cup Q_L \cup Q_R$. We interpret the states in Q_L and Q_R as "stay in the current state without changing the content of the tape and move one step to the left or right accordingly". We define δ' as follows:

- For $p = q^R$ we define $\delta'(p, \sigma) = (q, \sigma, R)$.
- For $p = q^L$ we define $\delta'(p, \sigma) = (q, \sigma, L)$.
- For $p \in Q$ we denote $\delta(p, \sigma) = (r, \tau, X)$ and define

$$\delta'(p, \sigma) = \begin{cases} (r, \tau, X), & X \in \{L, R, S\} \\ (r^L, \tau, X), & X = RR \\ (r^L, \tau, X), & X = LL \end{cases}.$$

It should be clear by this definition that $f_M = f_{M'}$. However, it is also easy to see why M is called a "fast" Turing machine as although M' reaches the same finite state as M in all computations, M can do reach it with less computation steps.

Example 1.3 (k-tape Turing machine)**.** The explanation of the equivalence of a k-tape Turing machine and a stadard Turing machine will be rather intuitive, as formalizing everything is quite standard.

Let k be a natural number greater than 0. A k-tape Turing machine is defined much like the standard Turing machine, but since we want to think of this machine as having multiple tapes, the transition function is defined as

$$\delta\colon (Q \setminus F) \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k.$$

It is clear that any Turing machine can be described using a k-tape Turing machine, and thus we will only explain why the other side of the quivalence is also true.

The main three challenges are proving we can describe k-tapes on a single tape, know the positions of the k heads at each step of the computation, and executing the computation itself.

- (describing k tapes) A simple way to describe k tapes is to add a special symbol \$ that is not part of the original tape alphabet symbols, and then write the contents of each tape in a row, using the symbol \$ to seperate the contents of the different tapes.
- (Identifying positions of k-heads) In order to identify the positions of the k heads each we add a new symbol c' for each symbol $c \in \Gamma$. The mark $'$ will signify the machine, this is the current place the head in at the current tape.
- (Executing the computation) When executing the computation we want the head to first remember the values of each head (those marked by $'$) and after reaching the kth strip, it will start exectuing on the kth strip according to the transition function of the original k-tape machine. After that it will execute on the $k-1$th strip and so on until the computation is finished.

Example 1.4.