Chapter 1

Turing machines

1.1 Definition

A *Turing machine*[1] (TM) is a tuple (Q, Σ, δ, q_0) where Q is a finite set of states (always including the two special states q_{accept} and q_{reject}), Σ is a finite alphabet (and unless it is otherwise specified it is $\{0, 1\}$), $\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, R\}$ is the *transition function* which in state q and the head reading a symbol σ changes to a new state q', overwrites σ with a new symbol σ', and then the head moves left (L) or right (R). Note that neither q' nor σ' have to be different from q and σ, respectively. The definition of a TM can be adapted to suit any occasion: there can be more than one tape, and the tape(s) can be infinite in one or two directions, etc. From a computational point of view, all these models are equivalent (all Turing-complete).

Given a string w as input, we place this string on the initial squares of the tape, and start the TM. The computation ends when either the state q_{accept} is entered (in which case it is understood that the TM accepts w), or the state q_{reject} is entered (in which case it is understood that the TM rejects w). It is very possible for the TM to never enter q_{accept} or q_{reject}, in which case the computation does not halt.

A *configuration* is a tuple (q, w, u) where $q \in Q$ is a state, and $w, u \in \Sigma^*$, and the cursor is on the last symbol of w. A configuration (q, w, u) *yields* (q', w', u') in one step, $(q, w, u) \overset{M}{\to} (q', w', u')$ if one step of M on

[1] Turing in 1936 ([Tur36]) was the first to use imaginary computers (which came to be known as *Turing machines*) for characterizing the class of algorithmic functions.

(q, w, u) results in (q', w', u'). Analogously, we define $\overset{M^k}{\to}$ (yields in k steps) and $\overset{M^*}{\to}$ (yields in any number of steps, including zero steps). The initial configuration, C_{init}, is $(q_0, \triangleright, x)$ where q_0 is the initial state, x is the input, and $\triangleright$ is the left-most tape symbol, which is always there to indicate the left-end of the tape[2]. For a TM with k tapes, a configuration is given by $(q, w_1, u_1, \ldots, w_k, u_k)$.

There are many definitions of computation other than the one given by Turing. Another model of computation is the so called *Unlimited Register Ideal Machine* (URIM)[3]. A URIM program P is a sequence of commands $\langle c_1, c_2, \ldots, c_n \rangle$, operating on a finite (but arbitrarily large) set of registers $R_1, R_2, \ldots, R_m$, and each command is one of the following types:

$$\begin{aligned} &(1) \quad R_i \leftarrow 0 \\ &(2) \quad R_i \leftarrow R_i + 1 \\ &(3) \quad \texttt{goto } i \texttt{ if } R_j = R_k \end{aligned}$$

If $R_j \neq R_k$, then simply the next command on the list is run. If we run out of commands (or are sent to an $i > n$ by a `goto`, then the program terminates. We can always assume that the input is given in R_1 at the beginning, and the output is given in R_m at the end (with 0 being "no" and 1 being "yes"). Note that the subscripts i, j, k are part of the instruction, i.e., they are not variable, but rather "hard-coded" into the program.

Finally, yet another model of computation is given by *recursive functions*[4]. A function f ($f(\vec{x})$, where $\vec{x} = x_1, x_2, \ldots, x_n$) is defined by *composition* from $g, h_1, \ldots, h_m$ if $f(\vec{x}) = g(h_1(\vec{x}), \ldots, h_m(\vec{x}))$, and f is defined by *primitive recursion* from g, h if $f(\vec{x}, 0) = g(\vec{x})$, and $f(\vec{x}, y+1) = h(\vec{x}, y, f(\vec{x}, y))$. There are three *initial functions* $Z, S, I_{n,i}$, where Z is the constant function equal to 0, $S(x) = x + 1$ (the successor function), and

[2]Note that $\triangleright$ is not part of the alphabet Σ, but is part of the *tape alphabet* Γ which always contains Σ. The symbol $\triangleright$ is convenient for denoting the left-end of the tape—and it can be moved to the right, when it is useful to ignore some initial segment of the tape; we follow [Pap94, Definition 2.1], where, besides the left-end of tape symbol $\triangleright$, the blank square $\sqcup$ is also introduced as part of Γ and not part of Σ. The point is the we can define Turing machines in a convenient way, with small alterations at will.

[3]Following the historical remarks in [BM77, §17]: Imaginary computers like URIMs where invented by several people independently (Shepherdson and Sturgis, Lambek, Minsky) in the late 1950s.

[4]The first definition of the class of recursive functions was given by Gödel in 1934 following a suggestion by Herbrand. This definition was proved by Kleene in 1936 to be equivalent to the definition which we are presenting here. (Once again, this historical remark comes from [BM77, §17].)

$I_{n,i}(\vec{x}) = x_i$ (the projection function). We say that a function is *primitive recursive* if it can be obtained from the initial functions by finitely many applications of primitive recursion and composition. Finally, we say that a function f is defined by *minimization* from g if $f(\vec{x}) = \mu y[g(\vec{x}, y) = 0]$, where this means that $f(\vec{x})$ is the smallest b such that $g(\vec{x}, b) = 0$, and a function is *recursive* if it can be obtained from the initial functions by finitely many applications of primitive recursion, composition, and minimization. Note that primitive recursive functions are always total, but recursive functions are not necessarily so.

Exercise 1.1.1 *Show that Turing machines, URIMS, and recursive functions, are all equivalent models of computation.*

Given a TM M we define $L(M)$ to be the set of strings accepted by M, i.e., $L(M) = \{w|\ M \text{ accepts } w\ \}$, or, put another way, $L(M)$ is the set of strings such that $(q_0, \triangleright, w)$ yields an accepting configuration.

1.2 Basic properties

We define $\mathsf{TIME}(f(n))$ to be the class of languages decidable by TMs running in time $O(f(n))$ (i.e., TMs that take at most $O(f(n))$ many steps before making a decision). We define $\mathsf{SPACE}(f(n))$ a little bit more carefully, because we want to make sense out of sub-linear space (i.e., less working space than the actual input size; for example, logarithmic space). In the context of space, we assume that we have a read-only input tape on which the input string is presented, and a work tape on which we bound how much tape we are allowed to use. Thus, we say that $\mathsf{SPACE}(f(n))$ is the class of languages decidable by TMs that use at most $O(f(n))$ squares of the work tape.

Theorem 1.2.1 *Given a k-tape TM M operating within time $f(n)$, we can construct a single-tape TM M' operating within time $O(f(n)^2)$, such that $L(M) = L(M')$.*

Exercise 1.2.2 *Prove theorem 1.2.1.*

Theorem 1.2.3 (Speed-Up) *Suppose that a TM decides a language L in time bounded by $f(n)$. For any $\varepsilon > 0$, there exists a TM that decides L in time $f'(n) = \varepsilon \cdot f(n) + n + 2$.*

Exercise 1.2.4 *Prove theorem 1.2.3.*

Exercise 1.2.5 *What does theorem 1.2.3 say about languages decidable in time $O(n)$? In general, what does this theorem say about the "big-Oh" notation?*

Exercise 1.2.6 *Suppose that we insist that the tape alphabet be $\{\triangleright, \sqcup, 0, 1\}$. Can the speed-up theorem be still applied?*

In a nondeterministic TM the transition function δ becomes a transition relation Δ, so "yields" is now a relation as well. A nondeterministic computation can be viewed as a tree, where we accept iff at least one branch ends in an accepting configuration. We define NTIME and NSPACE just as TIME and SPACE, except we allow nondeterministic TMs.

Theorem 1.2.7 $\mathsf{NTIME}(f(n)) \subseteq \bigcup_{c>1} \mathsf{TIME}(c^{f(n)})$.

PROOF: Let d be the *degree of nondeterminism*, meaning that d bounds the number of choices of Δ (this number is a constant, independent of the input). Thus, the computation tree has size at most $d^{f(n)}$, and we can traverse this tree (in a breadth-first manner) by recording where we are on the tree with a number in base d. □

1.3 Crossing sequences

1.3.1 A lower bound for palindromes

If $x = x_1 x_2 \dots x_n \in \Sigma^*$ is a string, then the *reverse* of x, denoted x^R, is just $x_n \dots x_2 x_1$. We say that a string is a *palindrome*[5] if $x = x^R$, and we define the language of palindromes as $L_{\text{pal}} = \{x \in \{0,1\}^* | x = x^R\}$.

[5]The invention of palindromes is generally attributed to Sotades the Obscence of Maronea, who lived in the third century BC in Greek-dominated Egypt. Surprisingly, palindromes appear not just in witty word games (such as `madamimadam` in James Joyce's *Ulysses*, or the title of the famous NOVA program, *A Man, a Plan, a Canal, Panama*), but also in the structure of the male defininig chromosome. Other human chromosome pairs fight damaging mutations by swapping genes, but because the Y chromosome lacks a partner, genome biologists have previously estimated that its genetic cargo was about to dwindle away in perhaps as listtle as five million years. However, researchers on the sequencing team discovered that the chromosome fights withering with palindromes. About six million of its fifty million DNA letters form palindromic sequences—sequences that read the same forward as backward on the two strands of the double helix. These copies provide backups in case of a bad mutation. (These observations come from [Liv05].)

Theorem 1.3.1 *Suppose that M is a one-tape TM that decides the language L_{pal}. Then, M requires $\Omega(n^2)$ many steps.*

PROOF: We first define the *i-th crossing sequence of M on x* to be

$$\{(q_1, \sigma_1), (q_2, \sigma_2), \ldots, (q_m, \sigma_m)\},$$

and it means that the first time M crosses from square i to square $i+1$, M leaves σ_1 on the i-th square, and finds itself in state q_1, and then the first time it crosses from square $i+1$ to square i, it arrives at square i in state q_2 and encounters σ_2 on the i-th square, etc.

Note that the odd pairs denote crossings left-to-right, and even pairs denote crossings right-to-left, and that $\sigma_{2i} = \sigma_{2i-1}$.

Now consider inputs of the form $x0^n x^R$, where $|x| = n$. Let $T_M(x)$ be the number of steps that M takes to decide (and accept) $x0^n x^R$.

There must be some i, $n < i \leq 2n$, i.e., some square in 0^n, for which the i-th crossing sequence has length $m \leq \frac{T_M(x)}{n}$. The reason is that the sum of the length of the crossing sequences corresponding to each square in 0^n has to be bounded above by $T_M(x)$, and therefore not all squares can have crossing sequences that are "long" ($> \frac{T_M(x)}{n}$). Let S be this "short" ($\leq \frac{T_M(x)}{n}$) crossing sequence. (Note that we assumed in this analysis that the Turing machine never stays put; it always either moves right or left. But this is not a crucial restriction as we can always "speed-up" a Turing machine, combining several moves into one, so that it never stays put.)

Claim 1.3.2 *M, n, i, S describe x uniquely.*

PROOF: To see this, we show how to "extract" x from M, n, i, S. For each $x \in \{0,1\}^n$, we simulate M on $x0^{i-n}$. The first time M wants to cross from square i to square $i+1$, we check that it would have done so with the pair (q_1, σ_1), and we immediately reset the state to q_2 and put the head back on square i and continue. The second time M crosses from square i to square $i+1$ we again check that it does so with (q_3, σ_3), and again we immediately reset the state to q_4 and put the head back on square i and continue. If all the odd-numbered crossing pairs are correct (i.e., correspond to those in S), we know that we have found x. Otherwise, we move on to examine the next x.

Our procedure identifies x correctly: suppose that some $y \neq x$ passed all the tests. Then M would have accepted $y0^n x^R$ which is *not* a palindrome.

This would contradict the correctness of M. (A minor technical point is that M must return to $\triangleright$ before accepting—this adds only a linear number of steps to the computation.) □

So M, n, i, S describe x uniquely, and they can be encoded with

$$c_M + \log(n) + \log(n) + c \cdot m \leq c_M + 2\log(n) + c \cdot \frac{T_M(x)}{n}$$

many bits. However, for every n, no matter how we encode strings, there must be a string x_0 whose encoding requires at least n many bits (otherwise, we would have a bijection from the set of strings of length n to the set of strings of length $n-1$, which is not possible).

Therefore, we have that

$$n \leq c_m + 2\log(n) + c \cdot \frac{T_M(x_0)}{n}.$$

which gives us the result. □

Let LT be the class of languages decidable in linear time (i.e., in time $O(n)$). (We are going to encounter this class again in section 4.5.) From theorem 1.3.1 we know that LT is *not robust* [6] (because we can decide L_{pal} in time $O(n)$ on a two-tape TM very easily, while on a single-tape TM we require $\Omega(n^2)$ time to decide L_{pal}). However, NLT (nondeterministic linear time) is quite robust as the next lemma shows.

Lemma 1.3.3 *If M_k^{nlt} is a nondeterministic linear time bounded TM with k tapes, then M_k^{nlt} can be simulated by M_2^{nlt}.*

PROOF: Let us assume that the input to M_2^{nlt} is written on tape 1. M_2^{nlt} starts by guessing the entire computation of M_k^{nlt}, and writing the guess on tape 2. This computation does not contain the tape configurations of M_k^{nlt}, but rather for each step of M_k^{nlt} it writes down the (supposed) state and symbols scanned by each of the k heads. Note that this information (if correct) is enough to determine the next move of M_k^{nlt}.

Now M_2^{nlt} checks that the computation it has written is correct, as follows. First it checks that the state sequence is consistent, by making one

[6] We say that a model of computation is *robust* if it is insensitive to "small" changes in the definition. This is a vague notion, but there are good examples of robust models: TMs with one or several tapes, infinite in one or two directions all decide the same set of recursive languages; if we restrict these TMs to be polytime, they are still insensitive to these modifications, so polytime TMs are a robust model of computation.

pass on tape 2, and checking that at each step the next state is what it should be, given the information from the previous step. Next, for each tape i of M_k^{nlt}, it makes a pass on tape 2 and checks that the scanned symbol information it has written for tape i is consistent. It does this by using tape 1 to simulate tape i, and using the information it has written on tape 2 concerning what the other $k-1$ tapes are scanning.

If all these consistency checks are passed, then the information on tape 2 is correct.

Exercise 1.3.4 *Carry out this induction: show that the t-th step is correct by induction on t.*

Hence M_2^{nlt} can figure out what M_k^{nlt} did. □

Exercise 1.3.5 *Show that L_{pal} is in $\mathsf{TIME}(n) \cap \mathsf{SPACE}(\log(n))$.*

1.3.2 Little space is no space at all

In this section we show that if a language can be decided with $o(\log\log n)$ space[7], then the language is in fact regular[8].

Theorem 1.3.6 *If a language L can be decided by a TM M such that M has a read-only input tape, and a work-tape where space is bounded by a function in $o(\log\log n)$, then L is regular.*

As the next exercise shows, $\log\log n$ is an exact threshold.

Exercise 1.3.7 *Consider the language (over $\{0,1,\#\}$)*

$$L = \{\#b_k(0)\#b_k(1)\#b_k(2)\#\ldots\#b_k(2^k-1)\#|k \geq 0\}$$

where $b_k(i)$ is the k-bit binary representation of $i \leq 2^k-1$. Show that this language is (i) not regular, and (ii) decidable in space $O(\log\log n)$.

[7] Recall that $o(f(n))$ is "little-oh" of $f(n)$, and $g(n) \in o(f(n))$ if $\lim_{n\to\infty} g(n)/f(n) = 0$.

[8] A language is *regular* if it can be decided by a Deterministic Finite Automaton (DFA). A DFA is just a TM which uses no space besides the input, and it reads the input from left-to-right, and it enters an accepting or rejecting state after it scans the last symbol of the input. See [Sip06b] for the background on DFAs.

We show first that if a language is decidable by a TM with work-tape space bounded by a constant, then the language is regular. Next, we show that if a language is not regular, then the work-tape requires $\Omega_{\text{weak}}(\log\log n)$ space. Note that, as the notation indicates, Ω_{weak} is the "weak" version of Ω, which means that for some constant c, $c\log\log n$ is a lower bound for *infinitely many* n's. On the other hand, the standard version of Ω would assure a lower bound for *all* n, rather than infinitely many.

We assume that our TMs have a read-only input tape, and a work-tape that is read & write, and we bound the space on the work-tape.

Claim 1.3.8 *If L can be decided in bounded space (i.e., there is a constant k bounding the space used on the work-tape), then L is in fact regular.*

PROOF: Note that we can get rid of constant space by keeping a record of all the possible $k \cdot |\Sigma|^k$ configurations (constantly many) as states. (Here there are k positions for the work-tape head, and the size of the alphabet is $|\Sigma|$, so $|\Sigma|^k$ are the possible contents of the work-tape.) But this is not enough, because the input-tape head can be moving back-and-forth, so we cannot say directly at this point that the machine can be simulated by a finite automaton.

We need to show that the machine can be transformed so that the head on the input-tape moves only from left-to-right. This can be accomplished as follows: for any string $s \in \Sigma^*$, define two functions $f_s, g_s : Q \longrightarrow Q$, where $f_s(q_1) = q_2$ if when the machine is started on s in state q_1, with the head on the first symbol of s, then the first time the machine leaves s (by moving from $s_{|s|}$ to the empty square $\sqcup$) or halts, it does so in state q_2. Let g_s be defined similarly, but instead of the head starting on the first symbol of s, it starts on the last symbol of s. Note that there are $|Q|^{|Q|}$ many functions from Q to Q, so they can all be hard-wired into the machine. We can now make the machine move only from left-to-right, and accept or reject after reading the last symbol of s, by doing the following: when we leave the i-th symbol of s, we know the values of $f_{s_1 s_2 \ldots s_i}$ and $g_{s_1 s_2 \ldots s_i}$. We want to compute $f_{s_1 s_2 \ldots s_{i+1}}$ and $g_{s_1 s_2 \ldots s_{i+1}}$. This is easy using the transition function of the original machine[9]. □

A crucial observation in the above reasoning is that the two new functions $f_{s_1 s_2 \ldots s_{i+1}}$ and $g_{s_1 s_2 \ldots s_{i+1}}$ depend only on the following: (i) the TM M, (ii) the two functions $f_{s_1 s_2 \ldots s_i}$ and $g_{s_1 s_2 \ldots s_i}$, and (iii) the single bit s_{i+1}. In particular,

[9]The original proof showing that a "one-way-automaton" can simulate a "two-way-automaton" can be found in [She59].

the two new functions *do not depend* on remembering the string $s_1 s_2 \dots s_i$, and this is very fortunate as arbitrarily long strings cannot be stored by a finite automaton.

Claim 1.3.9 *If a language L is not regular, then it requires $\Omega_{\text{weak}}(\log\log n)$ space. In other words, for any TM deciding L, there exists a constant c so that for* infinitely *many n's, there exists an input x of length n on which the machine will take at least $c \cdot \log\log n$ many steps.*

PROOF: Suppose that L is not regular. Then, by claim 1.3.8, we know that it cannot be decided in constant space. Let M be any machine deciding L.

We want to show that there exists an infinite sequence $\{n_i\}$ (and a constant c which depends only on M) such that $\forall n_i$, $\exists x_i$ such that $|x_i| = n_i$, and M requires at least $c \cdot \log\log n_i$ space to decide x_i.

To accomplish this, we use the fact that for any k we choose, we can always find an input x, such that M requires more than k squares of space to decide x. Pick a k_1 ($k_1 = 100$—it does not matter what it is) and find an x_1 of minimal length n_1, such that M requires $s_1 \geq k_1$ space to decide x_1. We show now that $c \cdot \log\log n_1 \leq s_1$.

For each $j = 1, 2, \dots, (n_1 - 1)$, let S_j be the j-th "crossing sequence", meaning the sequence consisting of the snapshots of M each time the head on the input tape crosses from square j to square $(j+1)$ or vice-versa. (Note that S_j contains more information than the crossing sequence S defined in theorem 1.3.1, as S_j contains the state, and the contents of the entire work-tape, and the position of the head on the work-tape.)

There are at most $N = s_1 \cdot |\Sigma|^{s_1}$ snapshots, and there are

$$N^1 + N^2 + \cdots + N^m = (N^{m+1} - 1)/(N - 1)$$

many possible crossing sequences of length at most m.

Since x_1 was chosen to be of minimal length, no two crossing sequences on x_1 are equal (for otherwise, the portion of the input between them could be eliminated, obtaining a new shorter input that still requires s_1 squares of space), so we know that $(n_1 - 1) \leq (N^{m+1} - 1)/(N - 1)$, and so $n_1 \leq N^{m+1}$. On the other hand, $m \leq N$ because otherwise we would have a loop, and M is a decider. Thus, $n_1 \leq N^{N+1}$, i.e.,

$$n_1 \leq (s_1 \cdot |\Sigma|^{s_1})^{(s_1 \cdot |\Sigma|^{s_1} + 1)}.$$

By taking log of both sides twice we get what we want (and note that the constant arises from $|\Sigma|$, the size of the alphabet of M).

We now pick k_2 sufficiently large so that there exists an x_2 of minimal length $n_2 > n_1$ such that M requires $s_2 \geq k_2$ space to decide x_2. Since L is not regular, we know that no matter how large k_2 is, we can always find an x_2 that requires at least k_2 space. The only problem is how to ensure that $n_2 > n_1$? As we are always picking an x_2 of minimal length (this is necessary for the argument to work) we might end up with $n_2 = n_1$. But there are finitely many x_2's of length n_1 (i.e., 2^{n_1}), so to make sure that this does not happen, we take a k_2 larger than the required space of any input of lenght n_1.

This procedure can be repeated *ad infinitum* to obtain $\{n_i\}$. □

If a language L can be decided in $o(\log\log n)$ space, then it is *not* the case that it requires $\Omega_{\text{weak}}(\log\log n)$ space, and so by the contrapositive of claim 1.3.9 it follows that it must be regular. This proves theorem 1.3.6. Note that this proof is not constructive, in the sense that we do not obtain a finite automaton from the $o(\log\log n)$-space bounded TM.

1.4 Answers to selected exercises

Exercise 1.2.2. Concatenate all the k tapes into one tape, and simulate one move of M with two passes of the entire tape of M' (the first pass of M' to determine what is underneath the heads on the tapes of M, and the second pass to make the necessary changes). Note that M uses at most $f(n)$ squares of each of its tapes[10].

Exercise 1.2.4. The idea is to "increase the word size." Introduce new symbols which encode several symbols of M.

Exercise 1.2.5. It only says that if a language can be decided in time $O(n)$, then it can be decided in time $(1+\varepsilon)n + 2$, for any $\varepsilon > 0$, and hence in a time that is almost strictly linear. That is, the constant can be made arbitrarily close to 1. As far as "big-Oh" notation, it is a way of justifying it, as the theorem says that constants are not important.

Exercise 1.2.6. At least not with the proof given, since it depends crucially on increasing the word size.

[10]This is an observation that we make all the time: in t many steps, a TM can write on at most the first t squares.

Exercise 1.3.5. L_{pal} is in linear-time because we can copy the string to a second tape, move the second head to the end, and then move the two heads simultaneously towards each other, comparing symbols. It is in logspace because we can work in $|w|$ many stages, in stage i we compare w_i to $w_{|w|+1-i}$. We keep track of the stages with one counter ($O(\log(|w|))$ many bits, and in each stage compute the positions i and $|w|+1-i$ with a second counter of the same size.

1.5 Notes

Exercise 1.1.1 is based on the presentation of URIM machines in [Coo06]. For a detailed proof of theorem 1.2.1 see [Pap94, Theorem 2.1], for theorem 1.2.3 see [Pap94, Theorem 2.2], and for theorem 1.2.7 see [Pap94, Theorem 2.6]. Section 1.3.1 is based on [Pap94, exercise 2.8.5], and it is also presented in [Koz06].