October 24, 2024

A Brief Introduction to the Theory of Computation and Complexity

Adrien Pommellet

Contents

An Introduction to the Theory of Computation

The savant Gottfried Leibniz dreamt in the seventeenth century of building a machine that could determine the truth values of mathematical statements. In 1928, the mathematician David Hilbert issued the *entscheidungsproblem*[1] problem: is there an algorithm that can determine the universal truth of a first-order logic statement?

[1] Also known as the *decision problem* in the civilized world.

One may indeed wonder what are the fundamental capabilities and limitations of computers. In this first chapter, we will prove in an informal manner that there exist some properties about programs that cannot be decided by an algorithm.

*The "*hello, world*" Problem*

We wonder if there exists an algorithm that can determine whether the twelve first letters output by a C program are "*hello, world*". This seminal example by Kernigan and Richie obviously complies with this constraint; we merely have to compile and run the program then check its output:

Being mere programmers, our concerns are cruder than Hilbert's.

```c
#include <stdio.h>

main(){
  printf("hello, world\n");
}
```

However, executing the program and reading its output is not a suitable procedure, as some programs may loop infinitely:

```c
#include <stdio.h>

main(){
  for (;;){}
  printf("hello, world\n");
}
```

Setting a time bound on our simulation would lead to some false negatives:

```c
#include <stdio.h>
#include <unistd.h>

main(){
```

```
  sleep(TIME_BOUND + 1);
  printf("hello, world\n");
}
```

Moreover, we can embed very complex problems in this seemingly trivial "*hello, world*" test. As an example, consider the following theorem:

Theorem 1 (Fermat's last theorem)**.** *No three strictly positive integers a, b, and c satisfy the equation $a^n + b^n = c^n$ for any integer value of n greater than 2.*

If we can determine whether the following program prints "*hello, world*" or not:

```
#include <stdio.h>
#include <math.h>

main(){
  int x, y, z;
  int n = 2;
  for (;;){
    for (x = 1; x <= n; x++){
      for (y = 1; y <= n; y++){
        for (z = 1; z <= n; z++){
          if (pow(x, n) + pow(y, n) == pow(z, n))
            printf("hello, world\n");
        }
      }
    }
    n++;
  }
}
```

Then we can confirm or refute Fermat's last theorem.

Keep in mind it took a mere four centuries to prove Fermat's last theorem. This doesn't bode well for our decision algorithm.

Of Undecidable Programs

We assume that programs can take any sequence of bits as input, even the code of another program.

A reasonable assumption, from a certain point of view. That's how code injection attacks work, as an example.

We suppose that there exists a program $\mathcal{T}$ such that, if $\mathcal{T}$ is given a program P and a sequence of bits x as inputs, then $\mathcal{T}$ prints "*yes*" if the twelve first letters output by $P(x)$ are "*hello, world*", and "*no*" otherwise. We can assume that $\mathcal{T}$ does not use the standard output stream `stdout` until it writes "*yes*" or "*no*".

We then write a program $\mathcal{T}_1$ that, given a program P and a sequence of bits x as inputs, outputs "*yes*" when $\mathcal{T}(P, x)$ would output "*yes*", and "*hello, world*" when $\mathcal{T}(P, x)$ would output "*no*" instead.

Eventually, we define $\mathcal{T}_2(x) = \mathcal{T}_1(x, x)$ and consider $\mathcal{T}_2(\mathcal{T}_2)$:

Giving a program its own code as an input is a common computability theory trick that will be used more than once in the chapters to come.

— If $\mathcal{T}_2(\mathcal{T}_2)$ outputs "*yes*", then $\mathcal{T}_1(\mathcal{T}_2, \mathcal{T}_2)$ outputs "*yes*", $\mathcal{T}(\mathcal{T}_2, \mathcal{T}_2)$ outputs "*yes*", and $\mathcal{T}_2(\mathcal{T}_2)$ therefore outputs "*hello, world*".

— However, if $\mathcal{T}_2(\mathcal{T}_2)$ outputs "*hello, world*" instead, then $\mathcal{T}_1(\mathcal{T}_2, \mathcal{T}_2)$ outputs "*hello, world*", $\mathcal{T}(\mathcal{T}_2, \mathcal{T}_2)$ outputs "*no*", and $\mathcal{T}_2(\mathcal{T}_2)$ therefore does not output "*hello, world*".

Both cases are absurd and the program $\mathcal{T}$ therefore doesn't exist. The *answer to Hilbert's decision problem* is therefore *negative*.

On Turing Machines

As a language-agnostic computing framework, we introduce Turing machines[2], a simple mathematical model that predates even the first computers but remains nonetheless expressive enough for a formal study of algorithms.

[2] An interactive Turing machine simulator can be found **here**.

A First Look at Turing Machines

From a certain point of view, a program is nothing but a sequence of instructions that interact with the memory by means of a processor. Thus, we will consider a theoretical computer model known as the *Turing machine* made of:

— An infinite[3] memory *tape* divided into cells. Each cell can contain a symbol from a finite alphabet, or remain blank. The tape is infinitely extendable to its left and to its right.
— A moveable *head* that can read, erase, and write symbols on tape cells. It cannot, however, jump to an arbitrary memory location, but can only read and modify one cell at a time.
— A finite memory (similar to an assembly register) called the *state* of the machine. The state space features both an initial and a final (or accepting) state.
— A finite, deterministic set of instructions called the *transition* function of the machine.

[3] Obviously, there is no such thing as a computer with infinite memory. But the sheer size of a computer's state space makes any finite-state model absurdly inefficient. Infinity remains a mere approximation of the finite but intractable reality, and not the other way round.

The Turing machine starts in its initial state, with a given input written on its tape. At each computation step (or cycle):

— The machine reads the cell of the tape pointed by the head.
— Depending on the content of the cell read by the head and the current state of the machine, the machine applies the transition function.
— The head overwrites the current cell, the machine changes its state, and then the head can move to an adjacent cell to its left or to its right, depending on the transition function's instructions.

The machine then keeps running until it reaches its accepting state or can no longer apply a transition. It can be stuck in an infinite loop.

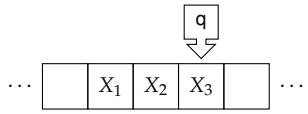

Figure 1: This Turing machine in state q is currently reading the cell X_3.

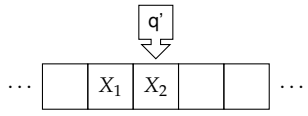

Figure 2: If the machine switches from state q to q', erases X_3, and moves its head to the left, we end in a new configuration.

A Formal Definition

Definition 1. *A Turing machine (TM) is a 7-tuplet* $M = (Q, q_0, q_a, \Sigma, \Gamma, B, \delta)$ *where:*

- Q *is a finite* state alphabet *such that* $Q \cap \Gamma = \emptyset$.
- q_0 *is the* initial state.
- q_a *is the* accepting state.
- Σ *is a finite* input alphabet.
- Γ *is a finite* work alphabet *such that* $\Sigma \subset \Gamma$.
- B *is the* blank symbol *such that* $B \in \Gamma$ *and* $B \notin \Sigma$.
- δ *is a partial* transition function *in* $(Q - \{q_a\}) \times \Gamma \to Q \times \Gamma \times \{L, S, R\}$. *The set* $\{L, S, R\}$ *stands for* $\{Left, Stay, Right\}$.

In order to give an instant description of a TM at a given step of a computation, we introduce configurations:

Definition 2. *A* configuration *c of M is an infinite word in* $Conf_M = B^\omega \Gamma^* Q \Gamma^* B^\omega$.

A configuration represents at the same time the tape content, the head, and the state of the TM: the state is written to the left of the tape cell currently being read by the tape head.

We may omit the infinite blanks to the left and the right of the TM's tape: $c = B^\omega w_1 q w_2 B^\omega$ can be written $c = w_1 q w_2$.

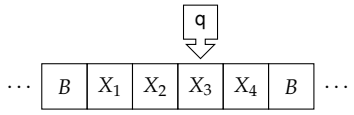

Figure 3: The configuration $X_1 X_2 q X_3 X_4$.

Graphical representation. We can represent a Turing machine as a graph whose nodes are the states in Q and whose edges are labelled in $\Gamma \times \Gamma \times \{L, S, R\}$. If $\delta(q, X) = (q', Z, m)$, then we draw an edge labelled by (X, Z, m) between states q and q'.

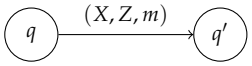

Figure 4: Graphical representation of a TM transition.

Semantics of Turing Machines

We define a formal *transition relation* $\vdash_M$ on configurations of a Turing machine M in the following manner:

Definition 3. *Given a configuration* $w_1 Y q X w_2 \in Conf_M$, *for* $\delta(q, X) = (q', Z, m)$, $m \in \{L, S, R\}$, *we define the following relations:*

Static. *If* $m = S$, $w_1 Y q X w_2 \vdash_M w_1 Y q Z w_2$.
Right move. *If* $m = R$, $w_1 Y q X w_2 \vdash_M w_1 Y Z q w_2$.
Left move. *If* $m = L$, $w_1 Y q X w_2 \vdash_M w_1 q Y Z w_2$.

We consider the *reachability relation* $\vdash_M^*$, which is the transitive closure of $\vdash_M$.

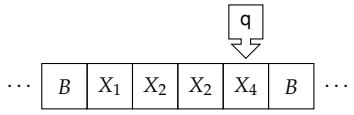

Figure 5: If we apply the transition rule $\delta(q, X_3) = (q', X_2, R)$ to $X_1 X_2 q X_3 X_4$, we end in a new configuration.

Exercise 1. Design a TM M such that, given an integer w written in binary with its least significant bit stored first, on its left (*little endian*), $q_0 w \vdash_M^* q_a w'$ where $w' = 2 \times w$.

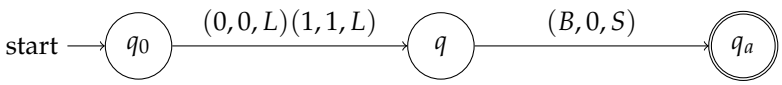

Figure 6: A graphical answer to exercise 1.

Exercise 2. Design a TM M such that, given an integer w written in binary with its most significant bit stored on its left (*big endian*), $q_0 w \vdash_M^* q_a w'$ and $w' = w + 1$.

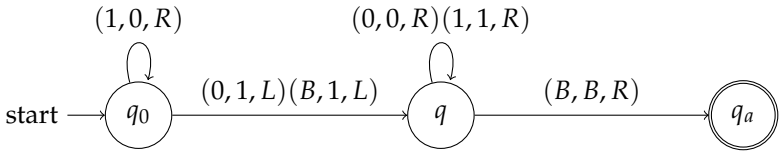

Figure 7: A graphical answer to exercise 2.

Definition 4. *A run of M on an input x is a (possibly infinite) sequence of configurations $(c_i)_{i \geq 0}$ such that $c_0 = q_0 x$ and $\forall i \geq 0$, $c_i \vdash_M c_{i+1}$.*

Definition 5. *We say that a TM M halts on an input w if $q_0 w \vdash_M^* w_1 q X w_2$ and the transition $\delta(q, X)$ is undefined*[4]*. Then M accepts w if $q = q_a$, and rejects w otherwise. If M accepts, then the tape content $w_1 X w_2$ is called the output of M.*

[4] Remember that δ is a partial function.

We can extend this definition to runs as well. Note that a TM will always halt if it reaches the accepting state q_a by design of δ.

Definition 6. *Two TMs M_1 and M_2 are said to be equivalent on their respective inputs x and y if M_1 halts on the input x (resp. rejects, accepts with output z) if and only if M_2 halts on the input y (resp. rejects, accepts with output z). We then write $M(x) \equiv N(y)$.*

If both machines share the same input alphabet Σ and $\forall x \in \Sigma^*$, $M_1(x) \equiv M_2(x)$, then M_1 and M_2 are said to be equivalent[5] *and we write $M \equiv N$.*

[5] Intuitively, if we think of M_1 and M_2 as black boxes producing outputs from inputs, then they are similar.

The Computing Power of Turing Machines

As abstract as it may seem, the simple Turing machine model is actually equivalent to superficially more expressive and realistic computing models.

Binary Turing Machines

Definition 7. *A Turing Machine is said to be* binary *if its input alphabet is* $\{0,1\}$ *and its work alphabet is* $\{0,1,B\}$.

We define the *binary language* $\mathcal{B} = \{0,1\}^*$.

Theorem 2. *Let* M *be a Turing machine and* $\beta : (\Gamma - \{B\}) \to \{0,1\}^k$ *a injective binary translation such that* $k = log_2(|\Gamma - \{B\}|)$. *Then there exists a binary TM* M' *such that* M *halts (resp. rejects, accepts with output* y*) on the input* x *if and only if* M' *halts (resp. rejects, accepts with output* $\beta(y)$*) on the input* $\beta(x)$.

We present an intuitive sketch[6] of the proof of this result:

[6] The full proof is left as an exercise to the reader.

Proof. We can extend β to Γ by assuming that $\beta(B) = B^k$. Our intuition is to simulate one cell of M by k cells of a new TM M'. In order to simulate a transition of M, we do the following operations:

1. Starting from a state q common to M' and M, we read k cells, storing the data read in the control state.
2. Depending on the binary encoding of a symbol in Γ read and the state of M simulated, we will simulate a transition d of δ_M.
3. To do so, we overwrite the k cells from right to left with the encoding of the output symbol of d.
4. We then move the tape head k cells to the left (resp. $2 \cdot k$ cells to the right) if d moved M's head to the left (resp. to the right).

If d ends in the accepting state, then so should the TM M' after simulating d. □

Example 1. Consider a transition rule $d : \delta_M(q, X) = (q', Y, L)$ of M such that $k = 3$, $\beta(X) = 010$ and $\beta(Y) = 101$. We want to design a binary simulation of d.

Starting from configuration $q010$, we must first read[7] 3 contiguous cells in a row:

$$q010 \vdash_N 0q_r^0 10 \vdash_N 01q_r^{01}1 \vdash_N 010q_r^{010}$$

Having read the binary representation of X, we can apply the transition rule d. To do so, we store the word 110 to be written[8] in the control state, then write it from right to left on the tape:

$$\vdash_N 01d_w^{101}0 \vdash_N 0d_w^{10}11 \vdash_N d_w^1001 \vdash_N d_w101$$

We must now simulate the move of M's head to the left. To do so, N moves[9] its head three times to the left before it reaches state q':

$$\vdash_N d_l^3101 \vdash_N d_l^2B101 \vdash_N d_l^1BB101 \vdash_N d_lBBB101 \vdash_N q'BBB101$$

[7] The state q_r^{101} stands for "the TM is in reading mode, simulates state q, and has read the word 010 from left to right".

[8] The state d_w^{101} stands for "the TM is applying the transition d and must write the word 101 from right to left".

[9] The state d_l^3 stands for "the TM is applying the transition d and must move the tape head 3 cells to the left".

Turing Machines with Multiple Tapes

We introduce a new Turing machine model with a finite number n of tapes, each with its own tape head:

Definition 8. *A Turing machine with n tapes is a 7-tuplet $M = (Q, q_0, q_a, \Sigma, \Gamma, B, \delta)$ where:*

- *$Q, q_0, q_a, \Sigma, \Gamma$, and B are similar to the components of a single-tape TM.*
- *δ is a partial transition function in $(Q - \{q_a\}) \times \Gamma^n \to Q \times \{1, \ldots, n\} \times \Gamma \times \{L, S, R\}$.*

Let $\triangleright \notin \Gamma$. A *configuration* of M is an element of $\text{Conf}_M = Q \times (B^\omega \Gamma^* \triangleright \Gamma^* B^\omega)^n$. The symbol $\triangleright$ stands for the position of each head on its respective tape. In a a manner similar to single-tape TM, we may omit writing B^ω when expressing a configuration.

A TM with multiple tapes can simultaneously read the cells pointed by its n heads, but will only (without loss of generality) modify a single tape at a time. We consider the following *semantics*:

Definition 9. *Given a configuration $(q, c_1, \ldots, c_n) \in Conf_M$ such that $\forall j \in \{1, \ldots, n\}, c_j = w_j Y_j \triangleright X_j w'_j$, and given $\delta(q, X_1, \ldots, X_n) = (q', i, Z, m)$, $m \in \{L, S, R\}$, we define a transition relation on configurations $(q, c_1, \ldots, c_n) \vdash_M (q', c'_1, \ldots, c'_n)$ such that $c'_j = c_j$ if $j \neq i$ and:*

Static. If $m = S$, $c_i = w_j Y_j \triangleright Z w'_j$.
Right move. If $m = R$, $c_i = w_j Y_j Z \triangleright w'_j$.
Left move. If $m = L$, $c_i = w_j \triangleright Y_j Z w'_j$.

Other notions such as halting, accepting, or rejecting are defined in a similar manner to TMs with a single tape:

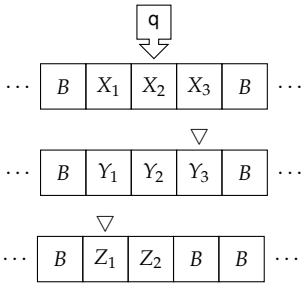

Figure 8: This TM with three tapes and three head is in configuration $(q, X_1 \triangleright X_2X_3, Y_1Y_2 \triangleright Y_3, \triangleright Z_1Z_2)$.

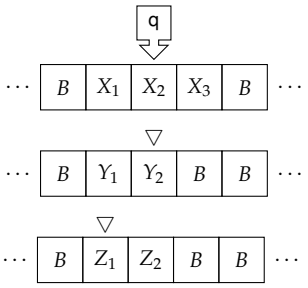

Figure 9: If we apply the transition rule $\delta(q, X_2, Y_3, Z_1) = (q', 2, B, L)$ to $(q, X_1 \triangleright X_2X_3, Y_1Y_2 \triangleright Y_3, \triangleright Z_1Z_2)$, we end in a new configuration. Intuitively, during a transition, a TM with n tapes reads from inputs from its n tape heads, but can only modify one tape.

Definition 10. *We say that a TM M with n tapes* halts *on an input*[10] *w if $(q_0, \triangleright w, \triangleright B, \ldots, \triangleright B) \vdash_M^* (q, w_1 \triangleright X_1 w'_1, \ldots, w_n \triangleright X_n w'_n)$ and the transition $\delta(q, X_1, \ldots, X_n)$ is undefined. Then M* accepts *w if $q = q_a$, and* rejects *w otherwise. If M accepts, then the tape content $w_n X_n w'_n$ is called the* output *of M.*

[10] As a convention, the input is written on the first tape, and the output on the last.

TMs with multiple heads, however, are not more expressive than TMs with a single tape:

Theorem 3. *Let M be a TM with k tapes. Then there exists a TM M' with a single tape such that $M \equiv M'$.*

We present an intuitive sketch[11] of the proof of this result:

[11] The full proof is left as an exercise to the forsaken reader.

Proof. Our intuition is to consider a single-tape TM M' such that each cell is split into $(2 \cdot k)$ sub-cells: k "*tape*" sub-cells to simulate the matching cells of the n tapes of M, and k "*head*" sub-cells than can either be blank or contain a symbol $\triangleright$ if one of the multiple heads of M is reading the current cell.

In order to simulate a transition rule of M, we first scan the tape of M' for $\triangleright$ tape head symbols in the "*head*" sub-cells, recording the content of the matching "*tape*" sub-cells in the control state until all the k heads have been found.

Then, if the i-th tape is meant to be modified, we scan the tape again to find the $\triangleright$ in the i-th "*head*" sub-cell, modify the i-th "*tape*" sub-cell accordingly, and simulate a move of the i-th head of M' by writing $\triangleright$ in the i-th "*head*" sub-cell of one of the adjacent cells of M' as well as erasing it from the current i-th "*head*" sub-cell.

To avoid looping while scanning the tape of M' for the non-blank "*head*" cells, we should keep at all time in the control state of M' the number of simulated heads of M to the left and to the right of the actual tape head.

Finally, if we need to accept with a given output, then right before reaching the accepting state, we replace every non-empty cell of the tape by the content of its n-th component that represents the matching cell on the last tape. □

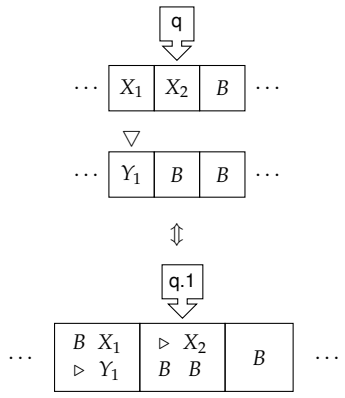

Figure 10: From a TM with two tapes to a TM with one tape. Being in state q.1 means that the TM simulates state q and there is one simulated tape head to its left.

We can therefore use TMs with multiple tapes as a more convenient computation model that often makes writing proofs simpler.

Turing Machines as Programs

At the assembly level, a program is nothing but a sequence of bits interpreted by a computer. In this chapter, we will show how this is true of Turing machines as well.

Encoding Turing Machines

Definition 11. *An* encoding *is an injective function $\mathcal{E}$ from the set of TMs to $\mathcal{B}$.*

An encoding matches a source code to a program. Note that it isn't surjective, as a binary word may not be a source code.

Let us consider a TM $M = (Q, q_0, q_a, \Sigma, \Gamma, B, \delta)$ on the input alphabet $\Sigma = \{0, 1\}$ and the work alphabet $\{0, 1, B\}$. We assume that $Q = \{q_0, \dots, q_n\}$.

We will encode M in the following manner, using the symbol 1 as a separator inserted between each step:

We combine an unary encoding of data with a separator.

1. We write the number of states $0^{|Q|} = 0^{n+1}$.
2. If the accepting state is q_k, we write its index 0^{k+1}.
3. For each transition rule $\delta(q_i, x) = (q_j, y, m)$, we write:
   (a) The input state 0^{i+1}.
   (b) The input symbol: 0 if $x = 0$, 00 if $x = 1$, 000 if $x = B$.
   (c) The output state: 0^{j+1}.
   (d) The output symbol, in a manner similar to the input.
   (e) The head movement: 0 if $m = L$, 00 if $m = S$, 000 if $m = R$.

We can compute this way a binary encoding $\mathcal{E}(M)$. From now on, we will use this standard encoding and write $\mathcal{E}(M) = \langle M \rangle$ as well as $\mathcal{M}_{\langle M \rangle} = M$.

If a binary string x doesn't make sense with regards to the encoding detailed previously, $\mathcal{M}_x$ is interpreted as the machine that always rejects any input immediately.

Application 1. Let M be the TM described by Figure 11. Then:

$$\mathcal{E}(M) = \underbrace{00}_{|Q|}\,1\,\underbrace{00}_{q_a}\,1\,\underbrace{01010101000}_{\delta(q_0,0)=(q_0,0,R)}\,1\,\underbrace{0100100100100}_{\delta(q_0,1)=(q_a,1,S)}$$

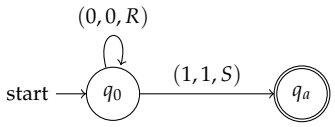

Figure 11: A TM accepting all words containing the symbol 1.

Definition 12. *Let $x \in \Sigma^*$. The Kolmogorov's complexity $K(x)$ of x is the size $|\langle M \rangle|$ of the smallest TM M accepting the empty input with output x.*

Example 2. $(abc)^{1000}$ and a random string of equal length on the alphabet $\{a, b, c\}$ are unlikely to have the same Kolmogorov complexity: the former can be output by a simple TM that simulates a `for` loop.

Simulating Turing Machines

Theorem 4. *There exists a TM $\mathcal{U}$ with three tapes on the input alphabet $\{0, 1, \#\}$ such that for all binary TM M and binary input x, $\mathcal{U}(\langle M \rangle \# x) \equiv M(x)$. $\mathcal{U}$ is called the universal machine.*

$\mathcal{U}$ simulates a run of the TM M on the input x.

We present an intuitive sketch[12] of the proof of this result:

[12] The full proof is left as an exercise to the zealous reader.

Proof. We consider a TM $\mathcal{U}$ with three tapes:

1. The first tape stores the source code $\langle M \rangle$.
2. The second tape stores the simulated control state of M.
3. The third tape simulates the tape of M.

Given an input $\langle M \rangle \# x$ on the first tape, $\mathcal{U}$ copies x on the third tape, erases $\#x$ on the first tape, and writes 0 on the second tape (the simulated initial state of M). $\mathcal{U}$ otherwise rejects any input that isn't of the form $w \# w'$ where $w, w' \in \mathcal{B}$.

Then, $\mathcal{U}$ scans the first tape for a transition rule that matches the simulated state on the second tape and the symbol read by the third tape head, applies said transition rule if it exists.

The TM $\mathcal{U}$ keeps simulating transitions until it no longer finds a rule it can apply, then it checks if the second tape is in an accepting state as described by $\langle M \rangle$ on its first tape: if it is, then $\mathcal{U}$ accepts, and rejects otherwise. □

1	Input code $\langle M \rangle$
2	Simulated state q
3	Simulated tape x

Figure 12: The three tapes of the universal TM $\mathcal{U}$.

The Church-Turing Hypothesis

It should be obvious by now that *a Turing machine can model a modern computer*. We have shown that TMs can read binary programs, interpret them and simulate a computer's memory with multiple infinite tapes.

As a consequence, we can admit the *Church-Turing hypothesis*: ignoring resource limitations, *a function is computable by an algorithm if and only if can be computed by a Turing machine*.

Languages of Turing Machines

In this chapter, we give a formal definition of *algorithmically decidable problems*, using the Turing machine framework introduced in the previous chapters.

Recursively Enumerable Languages

Definition 13. *The* language *of a TM M on the input alphabet Σ is the set $\mathcal{L}(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.*

Obviously, if $M_1 \equiv M_2$, then $\mathcal{L}(M_1) = \mathcal{L}(M_2)$.

Definition 14. *A language L on the input alphabet Σ is said to be* recursively enumerable *(RE) or* semi-decidable *if there exists a TM M on the input alphabet Σ such that $\mathcal{L}(M) = L$.*

We then say that M *accepts* L.

Note that M may or may not halt on words that are not in L.

Exercise 3. Compute a TM accepting the language $\{w \in \mathcal{B} \mid |w|_0 = |w|_1\}$.

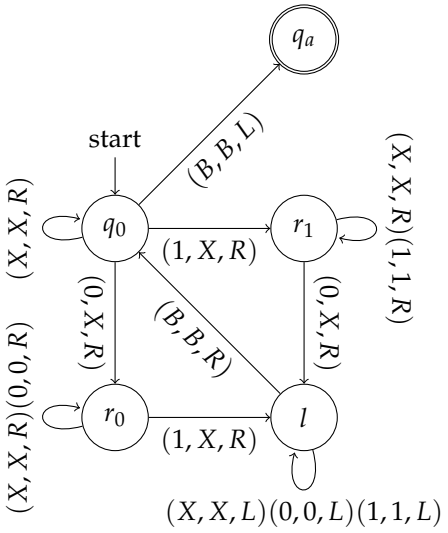

Figure 13: A graphical answer to exercise 3. Our intuition is the following: if we read a 0 (resp. a 1), we replace it by a symbol X, then we scan the input from left to right, looking for a 1 (resp. a 0) that we also replace by a X. We then rewind the head to the left of the input and scan the tape again until every input symbol has been replaced by a X, then we accept.

Proposition 1. *The set of RE languages on Σ is closed by union and intersection.*

Proof. Let L_1, L_2 be two RE languages and M_1, M_2 be two TMs such that $\mathcal{L}(M_1) = L_1$ and $\mathcal{L}(M_2) = L_2$.

Let $M_\cup$ be a TM with two tapes such that, on the input x, $M_\cup$ performs the following operations:

1. Copy x on the second tape.
2. Simulate a step of M_1 on the first tape if possible.
3. Simulate a step of M_2 on the second tape if possible.
4. Accept if either of these simulations accepts.
5. If they don't, go back to step 2.

Then $M_\cup$ accepts $L_1 \cup L_2$. We can design $M_\cap$ in a similar manner by replacing step 4 with "*Accept if both of these simulations accept.*". □

Step 1 of M_1 ⊢ Step 1 of M_2 ⊢ Step 2 of M_1 ⊢ Step 2 of M_2 ⊢ ...

Figure 14: *Interleaving* multiple step-by-step simulations is a common method to avoid being stuck if one of the simulated components loops.

On Recursive Enumeration

Definition 15. *We say that a TM M enumerates a (possibly infinite) sequence $w_0 \# w_1 \# \ldots$ on its tape if there exists a (possibly infinite) run $q_0 B \vdash_M^* w_0 \# q_1 \vdash_M^* w_0 \# w_1 \# q_2 \vdash_M^* \ldots$ and, at any step of this run, M never overwrites any cell located to the left of a # symbol (including the symbol itself).*

If M is a TM with multiple tapes, then the enumeration should be done on its last tape; the use of its other tapes is unconstrained.

Theorem 5. ***(1)*** *A language L on the input alphabet Σ is RE if and only if* ***(2)*** *there exists a TM M such that M enumerates[13] a sequence $w_0 \# w_1 \# \ldots$ where $\forall i \in \mathbb{N}, w_i \in L$ and $\forall l \in L, \exists! j \in \mathbb{N}, l = w_j$.*

[13] M recursively enumerates the words in L, hence, the name of this class of languages.

Proof. **(1) ⇒ (2)**. Let L be a RE language accepted by a TM M. Let M' be a TM with two tapes that, starting on an empty input, performs the following operations:

1. Starting from $i = 1$, simulate the i first steps of M on the i first words of Σ^* (sorted according to the radix order) on the first tape.
2. Write on the second tape any word accepted by one of these simulations if it has not already been written, inserting a # between each new word.
3. Increment i and go back to step 1.

Then M' obviously recursively enumerates the elements of L.

(2) ⇒ (1). Let M be a TM enumerating a sequence of words $w_0 \# w_1 \# \ldots$, and M' a TM with two tapes such that:

1. Its input x is stored at all time on the first tape.

	Word 1	Word 2	Word 3
Step 1 of M	1	2	3
Step 2 of M	2	3	...
Step 3 of M	3	...	...

Figure 15: *Diagonal* simulations are another common method to handle possibly infinite languages.

2. M is simulated on the second tape, enumerating $w_0 \# w_1 \# \ldots$.

Moreover, each time M' writes a # on its second tape, it compares the last word w_i written to the input x on the first tape. If both are equal, M' accepts. If the simulation of M halts, then M' halts as well.

Obviously, the TM M' accepts L. Hence, L is RE. □

We then say that *M enumerates L*.

A Non-recursively Enumerable Language

Theorem 6. *Let $\mathcal{D} = \{\langle M \rangle \mid M \text{ does not accept } \langle M \rangle .\}$ be the* diagonalization language. *Then $\mathcal{D}$ is not RE.*

Proof. We suppose that there exists a TM D accepting $\mathcal{D}$. If $D(\langle D \rangle)$ accepts, then $D(\langle D \rangle)$ does not accept. If $D(\langle D \rangle)$ does not accept, then $D(\langle D \rangle)$ should accept.

Both cases are absurd and D therefore doesn't exist. □

Cantor first designed this method to prove that the set of real numbers is not countable.

Recursive Languages

Definition 16. *A language L on the input alphabet Σ is said to be* recursive *(R) or* decidable *if there exists a TM M on the input alphabet Σ such that M accepts L and always halts*[14].

We say that *M recognizes L*. Obviously:

Proposition 2. *If L is R, then L is RE.*

Moreover:

Proposition 3. *The set of R languages on Σ is closed by complementation, union, and intersection.*

Proof. Let L be a R language accepted by a TM M. We consider the machine M' that simulates M on the input x, then rejects if M accepts and accepts if M rejects. M' obviously recognizes the language ${}^cL$ that is therefore R.

Closure by union and intersection can be proved in a similar manner to RE languages[15]. □

Theorem 7. *If both L and ${}^cL$ are RE, then L is R.*

Proof. Let M and ${}^cM$ be two TM accepting L and ${}^cL$ respectively. Let M' be a TM with two tapes such that, on the input x, M' performs the following operations:

1. Copy x on the second tape.
2. Simulate a step of M on the first tape if possible.

[14] As a direct consequence, M rejects ${}^cL$.

[15] The full proof is left as an exercise to the unlucky reader.

3. Simulate a step of ${}^{c}M$ on the second tape if possible.
4. Accept if the simulation of M accepts. Reject if the simulation of ${}^{c}M$ accepts.
5. If they don't, go back to step 2.

Since both L and ${}^{c}L$ are RE, at least one of the two TMs M and ${}^{c}M$ will halt on any input x. Hence, M' halts and obviously accepts L. □

A Non-recursive Language

Theorem 8. *Let* $\mathcal{H} = \{\langle M \rangle \# x \mid M \text{ halts on the input } x.\}$ *be the* halting problem. *Then* $\mathcal{H}$ *is not R.*

Proof. We suppose that there exists a TM H recognizing $\mathcal{H}$. Let H' be a TM such that, given an input x, if $H(x \# x)$ rejects, then $H'(x)$ accepts, and if $H(x \# x)$ accepts, then $H'(x)$ loops.

If $H'(\langle H' \rangle)$ accepts, then $H(\langle H' \rangle \# \langle H' \rangle)$ accepts. Hence, $H'(\langle H' \rangle)$ should not halt. If $H'(\langle H' \rangle)$ loops, then $H(\langle H' \rangle \# \langle H' \rangle)$ rejects and $H'(\langle H' \rangle)$ should halt.

Both cases are absurd and therefore H doesn't exist. □

Obviously, $\mathcal{H}$ is RE: one merely has to use the universal machine $\mathcal{U}$ on the input $\langle M \rangle \# x$.

Non-deterministic Turing Machines

In this chapter, we will stray from the deterministic computing model and focus on non-deterministic Turing machines, in an automata theoretic fashion.

A Non-deterministic Model

Definition 17. *A non-deterministic Turing machine (NTM) is a 7-tuplet* $N = (Q, q_0, q_a, \Sigma, \Gamma, B, \delta)$ *where:*

— $Q, q_0, q_a, \Sigma, \Gamma$, *and* B *are similar to the components of a deterministic TM.*
— δ *is a transition function in* $(Q - \{q_a\}) \times \to 2^{Q \times \Gamma \times \{L,S,R\}}$.

We can then define the code, the set of configurations, and the runs of a NTM in a manner similar to deterministic TMs. However, the semantics of the *transition* relation $\vdash_N$ of a NTM are defined non-deterministically[16]:

Definition 18. *Given a configuration* $w_1 Y q X w_2 \in Conf_N$, *for all* $(q', Z, m) \in \delta(q, X)$, $m \in \{L, S, R\}$, *we define the following relations in* $\delta(q, X)$.*:*

Static. *If* $m = S$, $w_1 Y q X w_2 \vdash_N w_1 Y q Z w_2$.
Right move. *If* $m = R$, $w_1 Y q X w_2 \vdash_N w_1 Y Z q w_2$.
Left move. *If* $m = L$, $w_1 Y q X w_2 \vdash_N w_1 q Y Z w_2$.

We define halting, accepting, and rejecting runs in a manner similar to deterministic TMs. However, instead of a single, linear run, these semantics also yield an *execution tree* that features all possible runs from a given starting configuration. Hence, NTMs have their own accepting condition on a given input:

Definition 19. *We say that a NTM* N accepts *an input* $w \in \Sigma^*$ *if there exists an accepting run on the input* w.

The *language* $\mathcal{L}(N)$ of a NTM N is the set of all words accepted by this NTM.

Definition 20. *We say that a NTM* N halts *on an input* $w \in \Sigma^*$ *if every run on the input* w *halts.* N rejects *the input* w *if it halts on* w *but does not accept* w.

[16] If a NTM is in state q and reads the work symbol X, then it can *apply one of possibly many rules*.

$$q_0 w \longrightarrow c_1 \longrightarrow c_2$$

Figure 16: A run of a deterministic TM. It is accepting if and only if c_2 is in an accepting state.

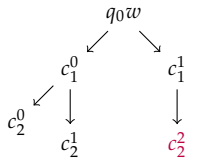

Figure 17: An execution tree of a non-deterministic TM. If, as an example, c_2^2 is in an accepting state, then the NTM accepts w.

Note that unlike TMs, a NTM can accept an input *without halting at the execution tree level*. Therefore, we can't define the output of a NTM on a given input, as different runs may yield different results. As a consequence, we define a new equivalence relation:

Definition 21. *Two machines (be they deterministic or non-deterministic) M_1 and M_2 are said to be* semi-equivalent on their respective inputs *x and y if M_1 halts on the input x (resp. rejects, accepts) if and only if M_2 halts on the input y (resp. rejects, accepts). We then write $M(x) \sim N(y)$.*

If both machines share the same input alphabet Σ and $\forall x \in \Sigma^*$, $M_1(x) \sim M_2(x)$, then M_1 and M_2 are said to be semi-equivalent *and we write $M \sim N$.*

We can prove that any language accepted by a NTM can be accepted by a TM as well:

Theorem 9. *If N is a NTM, then* there exists a TM *M such that $M \sim N$.*

Our intuition is to design a TM M that executes a breadth-first search of the execution tree of N on a given input. Here is a sketch[17] of this proof:

[17] The full proof is left as an exercise to the brave reader.

Proof. We introduce a TM M on the work alphabet $\Gamma \cup Q \cup \{\#, B'\}$ such that, given an input x, M does the following operations:

1. M writes the symbol q_0 to the left of the input x.
2. M stores a queue of configurations[18] of N on its tape, using a # separator symbol that does not belong to N's work alphabet.
3. M halts if this queue is empty.[19]
4. We read the leftmost configuration c of the queue. For each possible transition of N starting from this configuration c, M copies the resulting configuration c' at the end of the queue. Moreover, if c' is in state q_a, then M accepts.
5. M erases c from the queue and loops back to state 2.

[18] Of the form $c_0 \# c_1 \# c_2 \ldots$

[19] In order to check if the queue is empty, we encode configurations so that the queue is a continuous sequence of non-blank symbols. To do so, we use an alternative B' pseudo-blank symbols to encode configurations that is otherwise treated as a B symbol by the simulated rules of the NTM N.

It should be obvious that $M \sim N$. □

In a similar manner to TMs, we can design *NTMs with multiple tapes* that are equivalent to NTMs, hence, to TMs.

Applying Non-determinism

While NTMs accept the same set of RE languages as TMs, they can often do so in a more concise manner, making some proofs easier. Here are two examples of such properties:

Theorem 10. *The set of RE languages is closed by concatenation.*

Proof. Let L_1 and L_2 be two RE languages respectively accepted by the TMs M_1 and M_2. Let N be a NTM with two tapes such that, given an input x on its first tape, N does the following operations:

1. N arbitrarily splits x in two words $x = x_1 \cdot x_2$ by scanning x from left to right and non-deterministically choosing to halt the scan at any given step.
2. N copies x_2 on the second tape and erases it from the first tape.
3. N performs an interleaved simulation of M_1 on the first tape and of M_2 on the second tape. If both simulations accept, the current run of N accepts.

A common use of non-determinism is to *guess* a transformation of the input that can ease the next computation steps of a problem.

N then non-deterministically accepts $L_1 \cdot L_2$: given $l_1 \in L_1$, $l_2 \in L_2$, and $x = l_1 \cdot l_2$, if we consider the split $x = l_1 \cdot l_2$, then N accepts x. □

Theorem 11. *The set of RE languages is closed by Kleene star.*

Proof. Let L be a RE language accepted by the TMs M. Let N be a NTM with two tapes such that, given an input x on its first tape, N does the following operations:

1. N arbitrarily splits x in two words $x = x_1 \cdot x_2$ by scanning x from left to right and non-deterministically choosing to halt the scan at any given step.
2. N copies x_1 on the second tape and erases it from the first tape.
3. N simulates M on x_1. If M rejects x_1, then the current run of N halts.
4. If x_2 is empty, the current run of N accepts. Otherwise, N applies step 1 to x_2.

N then non-deterministically accepts L^*: given $l_1, \ldots, l_n \in L$ and $x = l_1 \cdot \ldots \cdot l_n$, if we consider the split $x = l_1 \cdot \ldots \cdot l_n$, then N accepts x. □

We can prove the following result in a similar manner:

Theorem 12. *The set of R languages is closed by concatenation and Kleene star.*

On Recursive Functions

One of the earliest use of Turing machines was to model computable functions. In this context, the content of the machine's tape as it accepts is just as significant as the fact that it halts in the first place.

Computing Functions

Definition 22. *Let $E \subseteq \Sigma^*$. A function $f : E \to \Sigma'^*$ is said to be* recursive *or* computable *if there exists a TM M on the input alphabet Σ such that $\forall x \in E$, $f(x) = y$ if and only if M accepts x with the output y. We then say that M* computes *f.*

Exercise 4. Consider the unary multiplication function f on 0^*10^* such that $f(0^n10^m) = 0^{nm}$. Prove that f is recursive.

Other common operations such as addition, subtraction, or division are similarly recursive.

Answer. Let M be a Turing Machine with two tapes. From a starting configuration $(q_0, 0^n10^m, B^\omega)$, M does the following operations:

1. Scan the first tape from left to right, seeking for a 1. If it can't find one and the head ends on a blank, M rejects.
2. From there, scan the first tape from left to right, seeking for a 0. If it finds one, then it is overwritten by 1. If it can't find one, then M accepts.
3. From there, scan the first tape from right to left, seeking for a blank, then moving the head to the right of that blank.
4. Scan the first tape from left to right, writing a new 0 on the second tape each time a 0 is read on the first tape, stopping when a 1 is read.
5. Go back to step 2.

Our intuition is to copy 0^n m times on the second tape, replacing a 0 from the input 0^m by a 1 each time a copy is performed until no 0 remains. When this loop ends, M accepts with the output 0^{nm} and has therefore computed the unary multiplication of n and m. □

Properties of Recursive Functions

Theorem 13 (*Kleene's iteration*)**.** *There exists a total, recursive function s on the set*[20] $\mathcal{B}_\# = \{0,1,\#\}^*$ *such that, given any TM M on $\{0,1,\#\}$, $\forall x, y \in \mathcal{B}$, $\mathcal{M}_{s(\langle M\rangle \# x)}(y) \equiv M(x \# y)$.*

[20] The # symbol is used a separator to feed multiple arguments to a TM or a recursive function using a single input string

Proof. Consider the function s that matches to any TM code $\langle M\rangle$ and binary input x the code $\langle M'\rangle$ of the following machine:

1. M' writes $x\#$ before the current input y.
2. M' simulates M on $x \# y$.

Then $\mathcal{M}_{\langle M'\rangle}(y) \equiv M(x \# y)$. □

Thus, the following theorem holds:

Theorem 14 (*Kleene's recursion*)**.** *Let f be a total recursive function. Then there exists a TM M such that $\mathcal{M}_{f(\langle M\rangle)} \equiv M$.*

Intuitively, M is a *fixed point* of f.

Proof. Let M be a TM be a TM with four tapes that, given an input $x \# y$ on its first tape, performs the following operations:

1. M computes $f(s(x \# x))$ on its second tape.
2. M copies y on its fourth tape.
3. M simulates $\mathcal{M}_{f(s(x\#x))}(y)$ with its last three tapes, in a manner similar to the universal machine.

1	Input $x \# y$
2	Simulated code $f(s(x \# x))$
3	Simulated state q
4	Simulated tape y

Figure 18: The four tapes of the TM M.

By definition of M:

$$M(\langle M\rangle \# y) \equiv \mathcal{M}_{f(s(\langle M\rangle \# \langle M\rangle))}(y)$$

And by Kleene's iteration theorem:

$$M(\langle M\rangle, y) \equiv \mathcal{M}_{s(\langle M\rangle \# \langle M\rangle)}(y)$$

Hence:

$$\mathcal{M}_{f(s(\langle M\rangle \# \langle M\rangle))}(y) \equiv \mathcal{M}_{s(\langle M\rangle \# \langle M\rangle)}(y)$$

Therefore $s(\langle M\rangle \# \langle M\rangle)$ is a fixed point of f. □

Application 2. Consider the recursive function f such that, given any TM M, $f(\langle M\rangle)$ is the code of the TM that, on any input, erases it then writes $\langle M\rangle$ on its tape and accepts.

By Kleene's recursion theorem, f has a fixed point x such that $\mathcal{M}_x$ writes its own code x on its tape then accepts. Such a program is called a *quine*.

One can actually compute the source code of a non-trivial[21] quine by following the steps of the proof of Kleene's recursion theorem.

[21] Merely opening and printing the program's own source file is obviously a despicable cheat worthy of a painful and humiliating death.

Reducing Problems

In this chapter, we will see how we can use a relation on languages called *reduction* and previous knowledge on a few canonical problems to determine whether another problem is decidable or not.

Introducing Reductions

Definition 23. *We say that a language A can be* Turing-reduced *to a language B if there exists a total recursive function f such that $x \in A \Leftrightarrow f(x) \in B$.*

We then write $A \leq_T B$. f is called the *reduction* of A to B.

Proposition 4. *If $A \leq_T B$ and B is decidable*[22] *(resp. RE), then A is decidable (resp. RE) as well.*

[22] Think of B being "bounded".

Proof. Since B is decided by a TM M, we can design M' that decides A by computing $f(x)$ on an input x then simulating M. □

As a consequence:

Proposition 5. *If $A \leq_T B$ and A is undecidable*[23] *(resp. not RE), then B is undecidable (resp. not RE) as well.*

[23] Think of A being "infinite".

Proof. If B were decidable (resp. RE), then so would be A. This is obviously not the case. □

Note that A being R or RE does not mean B is.

Undecidability Proofs Featuring Reductions

These canonical problems can all be shown to be undecidable using a proper reduction:

Proposition 6. *The* accepting problem *$\mathcal{A} = \{\langle M \rangle \# x \mid M \text{ accepts } x.\}$ is undecidable.*

Proof. We will prove that $\mathcal{H} \leq_T \mathcal{A}$.

Consider the function f that matches $(\langle M' \rangle \# x)$ to $(\langle M \rangle \# x)$, where M' is a TM that simulates M and accepts whenever M halts.

Obviously M' accepts x if and only if M halts on x. f is therefore a reduction from $\mathcal{H}$ to $\mathcal{A}$, and the latter problem is undecidable. □

Proposition 7. *The* equality problem $\mathcal{E} = \{\langle M_1 \rangle \# \langle M_2 \rangle \mid \mathcal{L}(M_1) = \mathcal{L}(M_2)\}$ *is undecidable.*

Proof. We will prove that $\mathcal{H} \leq_T \mathcal{E}$.

Consider the function f that matches to $(\langle M \rangle \# x)$ the codes $(\langle M_1 \rangle \# \langle M_2 \rangle)$ of two TM M_1 and M_2 such that, on any given input, M_1 erases it[24], writes x instead, simulates M on x, and eventually accepts if M halts, and M_2 always immediately accept regardless of the input.

M_1 either accepts every single word (and $\mathcal{L}(M_1) = \mathcal{L}(M_2)$) or none at all, depending on whether M halts on the input x or not. f is therefore a reduction from $\mathcal{H}$ to $\mathcal{E}$, and the latter problem is undecidable. □

[24] In order to design a reduction, we can define a TM that outright ignores the original input and instead rewrite a new constant input related to another (often simpler) problem that we then try to solve.

Proposition 8. *The* non-emptiness problem $\mathcal{L}_{ne} = \{\langle M \rangle \mid \mathcal{L}(M) \neq \emptyset\}$ *is undecidable.*

Proof. We will prove that $\mathcal{H} \leq_T \mathcal{L}_{ne}$.

Consider the function f that matches to $(\langle M \rangle \# x)$ the code $\langle M' \rangle$ of a TM M' such that, on any given input, M' erases it, writes x instead, simulates M on x, and eventually accepts if M halts.

M' either accepts every single word or none at all, depending on whether M halts on the input x or not. f is therefore a reduction from $\mathcal{H}$ to $\mathcal{L}_{ne}$, and the latter problem is undecidable. □

A diligent reader should have noticed a recurring pattern by now.

Using the same reduction function, we can also prove that:

Corrolary 1. *The* emptiness problem $\mathcal{L}_e = \{\langle M \rangle \mid \mathcal{L}(M) = \emptyset\}$ *is undecidable.*

As a consequence:

Proposition 9. *$\mathcal{L}_{ne}$ is RE and $\mathcal{L}_e$ is not.*

Proof. Consider a TM M' that, on an input $\langle M \rangle$, simulate the i first steps of M on the i first words of Σ^* (sorted according to the radix order), starting from $i = 1$. If any of these simulations accepts, then M' accepts. Obviously, M' accepts $\mathcal{L}_{ne}$.

Since $\mathcal{L}_e = {}^c\mathcal{L}_{ne}$ and $\mathcal{L}_{ne}$ is not decidable, by Theorem 7, $\mathcal{L}_e$ is not RE. □

Rice's Theorem

Let us consider the following class of languages:

Definition 24. *A property is a language $\mathcal{P}$ such that $\forall m, m' \in \Sigma^*$, if $\mathcal{M}_m \equiv \mathcal{M}_{m'}$, then $m \in \mathcal{P} \Leftrightarrow m' \in \mathcal{P}$.*

A property therefore describes the language and the outputs of a Turing machine, but not the way its computations are performed. Properties on TMs are unfortunately undecidable:

"M always halts in less than 10 steps." is not a property. *"M accepts any word of length shorter than 5."* is.

Theorem 15 (Rice's theorem)**.** *Let $\mathcal{P}$ be a non-trivial property (i.e. there exists $m \in \mathcal{P}$ and $m' \notin \mathcal{P}$). Then $\mathcal{P}$ is not decidable.*

Proof. We will prove that $\mathcal{H} \leq_T \mathcal{P}$.

Let M_0 be a TM that always loops on any input. We assume that $\langle M_0 \rangle \notin \mathcal{P}$ and consider $\langle M_1 \rangle \in \mathcal{P}$.

Consider the function f that matches to $(\langle M \rangle \# x)$ the code $\langle M' \rangle$ of a TM M' such that, on any given input y, M' simulates $M(x)$ then $M_1(y)$, accepting if the latter accepts.

If $(\langle M \rangle \# x) \in \mathcal{H}$, then $M \equiv M_1$ and $\langle M \rangle \in \mathcal{P}$. And if $(\langle M \rangle \# x) \notin \mathcal{H}$, then $M \equiv M_0$ and $\langle M \rangle \notin \mathcal{P}$. f is therefore a reduction from $\mathcal{H}$ to $\mathcal{P}$, and the latter problem is undecidable.

If $M_0 \in \mathcal{P}$, we prove that $\mathcal{H} \leq_T {}^c\mathcal{P}$ instead. □

A surprise, to be sure, but a welcome one.

Complexity of Turing Machines

In practice, being able to decide a problem is of very little help if it can't be done within a reasonable amount of time. We will introduce in this chapter the notion of *complexity* of Turing machines.

Time Complexity

Definition 25. *If $r = q_0 w \vdash_M c_1 \vdash_M \ldots \vdash_M c_n$ is an halting (resp. accepting, rejecting) run of a TM, then the integer n is called the* length *or the* number of steps *of r. We then say that M halts (resp. accepts, rejects) on the input w in n steps.*

If a TM does not loop, then we can define its worst case complexity:

If a TM does loop on some inputs, then its worst case complexity is $+\infty$.

Definition 26. *Let M be a TM that always halts. The* time complexity *of a TM M is the function t_M that matches to each integer n the smallest value $t_M(n)$ such that for all input w of length smaller than n, M halts on the input w in less than $t_M(n)$ steps.*

We can then introduce time complexity classes on recursive languages:

Definition 27. *Let $f : \mathbb{N} \to \mathbb{N}$ be an increasing function on integers. If M is a TM with k tapes such that $t_M = \mathcal{O}(f)$, then we write that $\mathcal{L}(M) \in \texttt{DTIME}_k(f)$, meaning that $\mathcal{L}(M)$ belongs to the class of recursive languages asymptotically recognized by a TM with k tapes in $\mathcal{O}(f)$ steps.*

Remember that, given two integer functions f and g, the statement $f = \mathcal{O}(g)$ means that there exists $n_0, K \in \mathbb{N}^*$ such that $\forall n \geq n_0, f(n) \leq K \cdot g(n)$.

We can then define the set $\texttt{DTIME}_*(f) = \bigcup_{k \geq 0} \texttt{DTIME}_k(f)$.

Time-constructible Functions

We consider a class of recursive functions such that their computation time is proportional to the size of the output produced:

Definition 28. *A function $f : \mathbb{N} \to \mathbb{N}$ is said to be* time-constructible *if there exists a TM M with any number of tapes such that, on the input 1^n, M accepts with the* unary *output $1^{f(n)}$ in $\mathcal{O}(f(n))$ steps. We then say that M* computes *f.*

Intuitively, given an input n, we can compute $f(n)$ in roughly $f(n)$ steps.

Note that M could instead take a *binary* input $\beta(n)$ and accept with the binary output $\beta(f(n))$: the unary and binary definitions are equivalent, although they obviously require different implementations[25].

[25] The proof of this statement is left as an exercise to the faithful reader.

Example 3. The function $n \to n^2$ is time-constructible. Consider a TM M with three tapes such that, given an input 1^n, M copies 1^n on its second tape then performs the following operations until the second tape is empty:

1. M copies the content 1^n of the first tape on the third tape.
2. M erases a 1 from the second tape.

As each copy of the input tape takes $\mathcal{O}(n)$ steps and there are n copy operations, M outputs 1^{n^2} in $\mathcal{O}(n^2)$ steps.

Example 4. The function $n \to 2^n$ is time-constructible. Consider a TM M with three tapes such that, given an input 1^n, M writes 1 on its third tape then performs the following operations:

1. M removes the leftmost 1 symbol from the first tape, and halts if it can't.
2. M copies the content of the third tape to the second tape.
3. M adds the content of the second tape to the output tape, erasing it and rewinding the second tape's head to the left as it performs the copy operation.
4. M rewinds the third tape's head to its leftmost position.

At the end of the i-th loop, the content of the third tape is 1^{2^i}. Moreover, step 1 takes $\mathcal{O}(n)$ transitions. Step 2, 3, and 4 are linear in the size of the third tape at the beginning of the loop, i.e. 2^{i-1}. Since $\sum_{i=1}^{n}(2^{i-1}) = 2^n - 1$, the whole computation takes $\mathcal{O}(2^n)$ operations.

Complexity Properties

We have proven that TMs can be simulated by other TMs and that increasing the amount of tapes of a TM does not improve its expressiveness. However, these operations do change the complexity of the computations performed.

Proposition 10. *Given a TM M with k tapes, there exist a single-tape TM M' such that M' can simulate n steps of M in $\mathcal{O}(n^2)$ steps.*

Proof. Let us consider the TM M' introduced in the proof of Theorem 3. In order to simulate the i-th step of M, given an input x, the TM M' must read at most $4 \cdot (|x| + (i-1))$ cells: one two-way pass on its tape (whose size is at most $|x| + (i-1)$) to find the k heads, and another to apply the relevant transition. Hence, in order to simulate the n first steps of M on the input $|x|$, M' takes $\mathcal{O}(\sum_{i=1}^{n}(i-1)) = \mathcal{O}(n^2)$ steps. □

Proposition 11. *There exists an universal TM $\mathcal{U}$ with three tapes such that, given a TM M and an input x, $\mathcal{U}$ can on the input $\langle M \rangle \# x$ simulate n steps of $M(x)$ in $\mathcal{O}(|\langle M \rangle| \cdot n)$ steps.*

Proof. Let us consider the TM $\mathcal{U}$ introduced in the proof of Theorem 4. In order to simulate a single step of M, the TM $\mathcal{U}$ must scan at most $|\langle M \rangle|$ cells of its input tape in order to find the right transition to apply, comparing each starting state of each transition with the simulated TM's current state. This two-way pass takes $\mathcal{O}(|\langle M \rangle|)$ steps. Hence, simulating n steps of $M(x)$ takes $\mathcal{O}(|\langle M \rangle| \cdot n)$ steps. □

Complexity Hierarchy

In this chapter, we will start building a *hierarchy* of languages, i.e. an increasing sequence of complexity classes.

Introducing Complexity Classes

We introduce polynomial and exponential complexity classes:

Definition 29. *We introduce the sets* $\texttt{PTIME} = \bigcup_{k\in\mathbb{N}} \texttt{DTIME}_*(n \to n^k)$ *(or just* P*) and* $\texttt{EXPTIME} = \bigcup_{k\in\mathbb{N}} \texttt{DTIME}_*(n \to 2^{n^k})$ *(or just* EXP*).*

P is the set of problems that can be solved in polynomial time, and EXP, the set of problems that can be solved in exponential time.

Example 5. We say that two integers are *relatively prime* if their greatest common divisor is 1. Then $\texttt{RPRIME} = \{(x, y) \mid x, y \in \mathbb{N}$ are relatively prime.$\}$ is in P. Indeed, the Euclidian algorithm is polynomial.

Example 6. The set $\texttt{CONNECTED} = \{G \mid G$ is a connected graph.$\}$ is in P. Indeed, we can check the connectivity of a given graph using a *depth-first search* algorithm in polynomial time.

Obviously:

Proposition 12. $\texttt{PTIME} \subseteq \texttt{EXPTIME}$.

Moreover, the following closure properties hold:

Proposition 13. *PTIME and EXPTIME are closed by union, intersection, and complementation.*

Proof. Intuitively[26], if we consider the TM designed in the proof of Proposition 3 in order to accept the union of two recursive languages, its time complexity is asymptotically equal to the sum of the complexity of the two TMs it simulates.

Hence, if both languages are in PTIME (resp. EXPTIME), then so is their union. The same holds true for the intersection and the complementation. □

[26] The full proof is left as an exercise to the diligent reader.

Building the Time Hierarchy

We want to prove that PTIME $\neq$ EXPTIME. To this end, we will need the following theorem:

Theorem 16 (Weak time hierarchy)**.** *Let f and g be two time-constructible functions such that $n \to n = o(g)$ and $f = o(n \to \frac{g(n)}{n})$. Then the strict inclusion $\mathtt{DTIME}_1(f) \subset \mathtt{DTIME}_*(g)$ holds.*

Remember that, given two integer functions f and g, the statement $f = o(g)$ means that $\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$.

Proof. Let us consider a TM D on 5 tapes that, given an input w of size n, performs the following operations:

1. D checks that the input is of the form $x \# 0^i$ and rejects otherwise[27].
2. D computes $g(n)$ in unary form on its second tape (this function is indeed time-constructible).
3. D copies n in unary form on its third tape.
4. D simulates $\lceil \frac{g(n)}{n} \rceil$ steps of $\mathcal{M}_x$ on the input w with its fourth and five tapes. In order to do so, it removes n symbols from the second tape each time it simulates a step until the second tape is empty.
5. If the simulation accepts, D rejects. In any other case, D accepts.

[27] We need to pad the argument and inflate it later in order to use the asymptotic complexity bounds defined in the hypothesis.

1	Input code x and padding
2	Counter $g(n)$
4	Value n
4	Simulated state q
5	Simulated tape w

Figure 19: The five tapes of the TM D.

Step 1 and 3 are linear. Step 2 takes $\mathcal{O}(g(n))$ transitions. Step 4 takes $\mathcal{O}(n \cdot \lceil \frac{g(n)}{n} \rceil)$ transitions by Property 11. Hence, D halts in at most $\mathcal{O}(g)$ steps and $\mathcal{L}(D) \in \mathtt{DTIME}_*(g)$.

We suppose that $\mathcal{L}(D) \in \mathtt{DTIME}_1(f)$. Then there exists a single-tape TM M recognizing $\mathcal{L}(D)$ such that $\exists a, b, \forall n \in \mathbb{N}, t_M(n) \leq a \cdot f(n) + b$. Since $f = o(n \to \frac{g(n)}{n})$, there exists n_0 such that $\forall n \geq n_0, a \cdot f(n) + b < \frac{g(n)}{n}$.

Consider $x = \langle M \rangle \# 0^{n_0}$. M halts on the input x in strictly less than $\frac{g(|x|)}{|x|}$ steps. Hence, $D(x)$ can simulate $M(x)$ until it halts. If $x \in \mathcal{L}(D)$, M accepts and D rejects. If $x \notin \mathcal{L}(D)$, M rejects and D accepts. Both cases are absurd[28], hence $\mathcal{L}(D) \notin \mathtt{DTIME}_1(f)$. □

[28] A diagonalization argument. How unexpected.

As a direct consequence of this theorem:

Theorem 17. PTIME $\subset$ EXPTIME.

Proof. Let $f(n) = 2^n$ and $g(n) = 2^{3n}$. We have $\mathtt{PTIME} \subseteq \mathtt{DTIME}_*(f)$ and $\mathtt{DTIME}_*(g) \subseteq \mathtt{EXPTIME}$. Moreover, by Property 10, $\mathtt{DTIME}_*(f) \subseteq \mathtt{DTIME}_1(n \to f(n)^2)$.

However, $(n \to f(n)^2) = o(n \to \frac{g(n)}{n})$. By the weak time hierarchy theorem, $\mathtt{DTIME}_1(n \to f(n)^2) \subset \mathtt{DTIME}_*(g)$. As a consequence, $\mathtt{PTIME} \subset \mathtt{EXPTIME}$. □

We admit[29] the generalized time hierarchy theorem:

[29] A full proof by Markus Krötzsch can be found in his Complexity Theory lecture notes **here**.

Theorem 18 (Time hierarchy)**.** *Let f and g be two time-constructible functions such that $f = o(n \to \frac{g(n)}{log(n)})$. Then $\mathtt{DTIME}_*(f) \subset \mathtt{DTIME}_*(g)$.*

Non-Deterministic Complexity

In this chapter, we extend the idea of *complexity* from runs and deterministic machines to non-deterministic Turing machines and execution trees.

Non-deterministic Time Complexity

Note that Definition 25 and Definition 26 apply to NTMs as well. We can therefore define time complexity classes for NTMs:

Definition 30. *Let $f : \mathbb{N} \to \mathbb{N}$ be an increasing function on integers. If M is a NTM with k tapes such that $t_M = \mathcal{O}(f)$, then we write that $\mathcal{L}(M) \in \texttt{NTIME}_k(f)$, meaning that $\mathcal{L}(M)$ belongs to the class of languages asymptotically recognized by a NTM with k tapes in $\mathcal{O}(f)$ steps.*

$t_M(n)$ is the lowest integer bound on the depth of the execution tree on any input of size n or smaller.

We also define the set $\mathit{NTIME}_*(f) = \bigcup_{k \geq 0} \mathit{NTIME}_k(f)$. Obviously:

Proposition 14. $\texttt{DTIME}_k(f) \subseteq \texttt{NTIME}_k(f)$.

Non-deterministic Complexity Properties

Using Theorem 9, we can show that the following relationship between TMs and NTMs hold:

Theorem 19. *If $N \in \texttt{NTIME}(f)$ is a NSTM, then there exists a STM $M \in \texttt{DTIME}(2^{\mathcal{O}(f)})$ such that $M \sim N$.*

The notation $g = 2^{\mathcal{O}(f)}$ means that there exists a function $h = \mathcal{O}(f)$ such that $g = \mathcal{O}(n \to 2^{h(n)})$.

Proof. Let k be an integer such that, from any given configuration of N, one can apply at most k transitions. Since $N \in \texttt{NTIME}(f)$, there exists a constant K such that $t_N(n) \leq K \cdot f(n)$. Given an input of size n, the TM M introduced in the proof of Theorem 9 performs a breadth-first search on a tree of arity[30] k at most and of depth $K \cdot f(n)$ at most. Hence, assuming it takes at most $k^{(K \cdot f(n))}$ transitions to simulate a single step of N (i.e. reading the entire set of possible configurations), the whole computation takes $\mathcal{O}((k^{(K \cdot f(n))})^2) = 2^{\mathcal{O}(f(n))}$ steps. □

[30] The *arity* of a tree is the maximum number of children any given node can have.

As a direct consequence:

Theorem 20. $\texttt{NTIME}(f) \subseteq \texttt{DTIME}(2^{\mathcal{O}(f)})$.

By Theorem 4, there exists an universal TM that can simulate single-tape TMs from their source code. We show that an universal machine exists for NTMs as well:

Theorem 21. *There exists a NTM $\mathcal{U}'$ with three tapes on the input alphabet $\{0,1,\#\}$ such that for all binary NTM N and binary input x, $\mathcal{U}'(\langle N\rangle \# x) \sim N(x)$. $\mathcal{U}'$ is called the* non-deterministic universal machine.

Moreover, it takes $\mathcal{O}(|\langle N\rangle| \cdot n)$ steps to simulate n steps of $N(x)$.

$\mathcal{U}'$ simulates a run of the NTM N on the input x.

Proof. In a similar manner to the proof of Theorem 4, we consider a NTM $\mathcal{U}'$ with three tapes:

1. The first tape stores the source code $\langle N\rangle$.
2. The second tape stores the simulated control state of N.
3. The third tape simulates the work tape of N.

1	Input code $\langle N\rangle$
2	Simulated state q
3	Simulated tape x

Figure 20: The three tapes of the universal NTM $\mathcal{U}'$.

Given an input $\langle N\rangle \# x$ on its the first tape, $\mathcal{U}$ copies x on the third tape, erases $\#x$ from the first tape, and writes 0 on the second tape (the simulated initial state of N).

Then, $\mathcal{U}'$ keeps scanning the first tape for a transition rule that matches the simulated state on the second tape and the symbol read by the third tape head; if it finds such a transition rule, it can non-deterministically choose either to apply it or to skip it and keep scanning the tape for another transition.

However, to ensure $\mathcal{U}'$ does eventually apply a transition if one exists, it must deterministically apply the first matching transition it finds after its first non-deterministic two-way pass of $\langle N\rangle$ on the first tape. A simulated transition therefore takes at most $\mathcal{O}(|\langle N\rangle|)$ steps.

If $\mathcal{U}'$ can't find such a transition rule, then it checks if the second tape is in an accepting state as described by $\langle N\rangle$ on the first tape: if it is, then $\mathcal{U}'$ accepts, and halts otherwise. □

Non-Deterministic Complexity Hierarchy

In this chapter, we extend the time hierarchy with non-deterministic complexity classes and introduce what is perhaps the most famous unsolved problem in computer science.

Non-deterministic Complexity Classes

In a manner similar to deterministic TMs, we introduce polynomial and exponential complexity classes:

Definition 31. *We introduce the sets* NPTIME $= \bigcup_{k\in\mathbb{N}} \mathit{NTIME}_*(n \to n^k)$ *(or just* NP*) and* NEXPTIME $= \bigcup_{k\in\mathbb{N}} \mathit{DTIME}_*(n \to 2^{n^k})$ *(or just* NEXP*).*

NP is the set of problems that can be non-deterministically solved in polynomial time, and EXP, the set of problems that can be non-deterministically solved in exponential time.

Example 7. The set $\mathit{SAT} = \{\varphi \mid \varphi \text{ is a satisfiable Boolean formula.}\}$ is in NP. If we guess[31] the right valuation, checking that a Boolean formula holds true for this valuation only takes a polynomial amount of time.

[31] By guessing, we mean that a non-deterministic program can write every possible value in the valuation space, in particular any one that satisfies the Boolean formula.

Example 8. The travelling salesman problem $\mathit{TSP} = \{(G, W) \mid G \text{ has a Hamiltonian circuit of weight at most } W.\}$ is in NP. We can guess a path in the graph G, then check that it is indeed a Hamiltonian[32] circuit of weight lesser than W in polynomial time.

[32] A *Hamiltonian* circuit is a path in a graph that loops and visits each vertex exactly once.

Obviously:

Proposition 15. $\mathit{NPTIME} \subseteq \mathit{NEXPTIME}$.

Moreover, as a consequence of Theorem 19:

Theorem 22. NPTIME $\subseteq$ EXPTIME.

The following closure properties also hold[33]:

[33] The full proof is left as an exercise to the unfortunate reader.

Proposition 16. *NPTIME and NEXPTIME are closed by union and intersection.*

Note that we do not know if NPTIME and NEXPTIME are closed by complementation. For a given complexity class X, we therefore introduce the class coX such that $L \in \text{coX} \Leftrightarrow {}^cL \in \text{X}$.

Example 9. The set $\mathit{ANTILOGY} = \{\varphi \mid \varphi \text{ is a Boolean formula that never holds true.}\}$ is in coNP. Indeed, its complement SAT is in NP.

Proposition 17. *$P \subseteq NP \cap coNP$.*

Proof. Obviously, $\mathtt{P} \subseteq \mathtt{NP}$ and $\mathtt{coP} \subseteq \mathtt{coNP}$. However, $\mathtt{P} = \mathtt{coP}$ by Proposition 13. Thus, $\mathtt{P} \subseteq \mathtt{NP} \cap \mathtt{coNP}$. □

Building the Non-deterministic Hierarchy

In a similar manner to the deterministic case, a time hierarchy theorem that we will admit[34] holds for non-deterministic complexity classes:

[34] A full proof can be found in Stephen A. Cook's paper *A hierarchy for non-deterministic time complexity*.

Theorem 23 (Non-deterministic time hierarchy)**.** *Let f and g be two time-constructible functions such that $(n \to f(n+1)) = o(g)$. Then the strict inclusion $\mathtt{NTIME}_*(f) \subset \mathtt{NTIME}_*(g)$ holds.*

As a direct consequence:

Theorem 24. $\mathtt{NPTIME} \subset \mathtt{NEXPTIME}$.

Proof. Similar to that of Theorem 17. □

Note that the following relation between P and NP holds:

Theorem 25 (Certification)**.** ***(1)*** *$L \in NP$ if and only if* ***(2)*** *there exists $L' \in P$ on an input alphabet Σ' and a polynomial π such that:*

$$x \in L \Leftrightarrow \exists y \in (\Sigma')^{\leq \pi(|x|)} \text{ such that } x \# y \in L'$$

The word y is then called the certificate *or* witness *of x.*

Intuitively, we can compute a solution x to a given problem non-deterministically in polynomial time if and only if we can provide a certificate y such that, given x and y, we can using y deterministically check that x is indeed a correct solution in polynomial time with regards to x and y.

Proof. **(1)** $\Rightarrow$ **(2)**. If $L \in \mathtt{NP}$, then there exists a NTM N on the input alphabet Σ recognizing L and a time-computable polynomial π such that $t_N \leq \pi$. Let Δ be the set of transitions of N. Note that to each run r of N on a input x of size n, we can match the sequence of transitions $d_r \in \Delta^{\leq \pi(n)}$ used by r.

Let M be a TM with 3 tapes on the input alphabet $\Sigma' = \Sigma \cup \Delta \cup \{\#\}$ that performs the following operations on an input w of size n:

1. M checks that the input w is of the form $x \# y$, $y \in \Delta^*$, and rejects otherwise.
2. M computes $\pi(|x|)$ in unary form on its second tape.
3. M simulates on its third tape the sequence of transitions y of the NTM N on the input x for $\pi(|x|)$ steps.
4. If the simulation accepts after applying the whole sequence y, then M accepts. M rejects in any other case.

1	Input x and transitions y
2	Counter $\pi(\|x\|)$
3	Simulated tape of N

Figure 21: The three tapes of the TM M.

M accepts $x \# y$ if and only if there exists a run r of N accepting the input x such that $d_r = y$. Moreover, step 1 is linear. Step 2 takes $\mathcal{O}(\pi(n))$ operations. Simulating a transition in the sequence y takes a constant amount of time and M simulates $\pi(|x|) \leq \pi(n)$ transitions, so

step 3 takes $\mathcal{O}(\pi(n))$ operations as well. Hence, M runs in polynomial time and $L' = \mathcal{L}(M) \in \mathtt{P}$.

(2) ⇒ (1). Let M be a single-tape TM accepting L' in polynomial time, and N be a NTM with two tapes that performs the following operations on an input x:

1. N computes $1^{\pi(|x|)}$ on its second tape.
2. N non-deterministically guesses $y \in \Delta^{\leq \pi(|x|}$ and writes $\#y$ after x on its first tape.
3. N simulates M on the input $x \# y$.

1	Input x # guess y
2	Counter $\pi(\|x\|)$

Figure 22: The two tapes of the NTM N.

$M(x)$ accepts if and only if $\exists y \in (\Sigma')^{\leq \pi(|x|)}$ such that $x \# y \in L'$. Hence, $L = \mathcal{L}(M)$. Moreover, step 1 of M is polynomial. Step 2 is linear. Step 3 is polynomial since $L' \in \mathtt{P}$. As a consequence, $L = \mathcal{L}(M) \in \mathtt{NP}$. □

Example 10. A certificate x of an instance φ of SAT would be any valuation such that $\varphi(x)$ holds true.

Example 11. A certificate h of an instance (G, W) of TSP would be any path h of G such that h is a circuit of weight lesser than W visiting each vertex exactly once.

Collapsing the Time Hierarchy

We know that $\mathtt{P} \subseteq \mathtt{NP} \subseteq \mathtt{EXP} \subseteq \mathtt{NEXP}$, $\mathtt{P} \neq \mathtt{EXP}$, and $\mathtt{NP} \neq \mathtt{NEXP}$. We don't know, however, if $P = NP$. This is the most well-known unsolved problem in computational complexity theory.

Thanks to the certification theorem, $\mathtt{P} = \mathtt{NP}$ would mean that if the solution to a problem is easy (polynomial) to check for correctness, then so is the original problem as well.

Solving it would bear major consequence on the time hierarchy. Indeed:

Theorem 26. *If $P = NP$, then $\mathtt{EXP} = \mathtt{NEXP}$.*

Proof. Let $L \in \mathtt{NEXP}$ and let N be a NTM accepting L. Then $\exists k_1, k_2 \in \mathbb{N}$ such that $\forall n \in \mathbb{N}$, $t_N(n) \leq (k_1 \cdot 2^{n^{k_2}})$. Consider $L' = \{x \# 0^{k_1 \cdot 2^{|x|^{k_2}}} \mid x \in L\}$. By the padding theorem[35], $L' \in \mathtt{NP}$.

By hypothesis, $L' \in \mathtt{P}$, hence, by the padding theorem, $L \in \mathtt{EXP}$. Therefore, $\mathtt{NEXP} \subseteq \mathtt{EXP}$. However, $\mathtt{EXP} \subseteq \mathtt{NEXP}$ by Proposition 14. As a consequence, $\mathtt{EXP} = \mathtt{NEXP}$. □

[35] Check exercise 2 of sheet 4. A similar theorem holds for non-deterministic complexity classes.

Theorem 27. *If $P = NP$, then $\mathtt{NP} = \mathtt{coNP}$.*

Proof. Obviously, $\mathtt{P} = \mathtt{coP}$ by Proposition 13. Moreover, if $\mathtt{P} = \mathtt{NP}$, then $\mathtt{coP} = \mathtt{coNP}$. Thus, $\mathtt{NP} = \mathtt{P} = \mathtt{coP} = \mathtt{coNP}$. □

Polynomial Reductions

We have proven that recursive functions called *reductions* can preserve decidability. We will show this is true of complexity classes as well if the reductions can be computed in polynomial time.

Introducing Polynomial Reductions

Definition 32. *A polynomial-time reduction of a language A to a language B is a reduction f of A to B such that f can be computed in polynomial-time, i.e. there exists a TM M computing f and a polynomial π such that $t_M \leq \pi$.*

We then write $A \leq_T^p B$.

Example 12. Consider the problems $\texttt{CLIQUE} = \{(G,k) \mid G \text{ has a clique of size } l \leq k.\}$ and $\texttt{IS} = \{(G,k) \mid G \text{ has an independent set of size } l \leq k.\}$. Obviously, there exists an l-clique in a graph G if and only if there exists an independent set of size l in ${}^cG$. The function $f : (G,k) \to ({}^cG,k)$ is a polynomial reduction of `CLIQUE` to `IS` and of `IS` to `CLIQUE`.

A *clique* is a set of vertices in a graph such that every two distinct vertices in the set are adjacent. An *independent set* is a set of vertices, no two of which are adjacent.

Proposition 18 (Transitivity)**.** *If $A \leq_T^p B$ and $B \leq_T^p C$, then $A \leq_T^p C$.*

Proof. Let f and g be polynomial-time reductions of A to B and of B to C, respectively. Then $(g \circ f)$ is a polynomial-time reduction of A to C. □

Proposition 19. *If $A \leq_T^p B$ and $B \in \texttt{P}$ (resp. $\texttt{NP}$), then $A \in \texttt{P}$ (resp. $\texttt{NP}$).*

Proof. Let M_B be a TM that recognizes B in polynomial time and f be a polynomial-time reduction of A to B. Consider the TM M_A that, on the input x, computes $f(x)$ then simulates M_B on $f(x)$. M_A recognizes A in polynomial time, hence $A \in \texttt{P}$.

The proof of the `NP` case is similar. □

Completeness and Hardness

Definition 33. *Let $\texttt{X}$ be a complexity class. A language L is said to be* X-hard *if, $\forall A \in \texttt{X}$, $A \leq_T^p L$. Moreover, if $L \in \texttt{X}$, then L is said to be* X-complete.

Obviously:

Proposition 20. *If A is X-hard and $A \leq_T^p B$, then B is X-hard.*

We can prove that P-completeness is trivial:

Proposition 21. ***(1)*** *$L \in \mathsf{P}$ and $L \neq \emptyset, \Sigma^*$ if and only if* ***(2)*** *L is P-complete.*

Proof. **(2) ⇒ (1)**. $\emptyset$ can't be P-complete: if $A \leq_T^p \emptyset$, then there exists a polynomial reduction such that $\forall x \in A$, $x \in A \Leftrightarrow f(x) \in \emptyset$, hence, $A = \emptyset$. In a similar manner, Σ^* can't be P-complete.

(1) ⇒ (2). Let $L, A \in \mathsf{P}$ such that $\exists x_1 \in L$ and $\exists x_2 \notin L$. Consider f such that $f(x) = x_1$ if $x \in A$ and $f(x) = x_2$ if $x \notin A$. Membership of A can be decided in polynomial time, hence f is a polynomial-time reduction of A to L and $A \leq_T^p L$. Therefore, L is P-complete. □

NP-complete languages are known to be the best candidates for being in NP but not in P. Otherwise:

Proposition 22. *Let L be a NP-complete language. If $L \in \mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$.*

Proof. Any language $A \in \mathsf{NP}$ can be reduced in polynomial-time to L. By Proposition 19, if $L \in \mathsf{P}$, then $A \in \mathsf{P}$ as well and $\mathsf{NP} \subseteq \mathsf{P}$. Since $\mathsf{P} \subseteq \mathsf{NP}$, we would have $\mathsf{P} = \mathsf{NP}$. □

NP-complete and NP-hard Problems

Proposition 23. *The* non-deterministic halting problem *$\mathcal{H}' = \{\langle N \rangle \# x \mid$ N is a NTM halting on the input x.$\}$ is NP-hard but not NP-complete.*

Proof. The deterministic halting problem $\mathcal{H}$ is undecidable, hence, $\mathcal{H}'$ is undecidable as well and $\mathcal{H}' \notin \mathsf{NP}$.

Let $L \in \mathsf{NP}$ and N be a NTM accepting L in polynomial time. We consider a NTM N' similar to N, but that only ever halts if N accepts and loops otherwise. The function $f : x \to \langle N' \rangle \# x$ is obviously computable in polynomial time. Moreover, $x \in L \Leftrightarrow \langle N' \rangle \# x \in \mathcal{H}'$. Hence, $L \leq_T^p \mathcal{H}'$. □

Proposition 24. *The* bounded accepting problem *$\mathcal{A}_t = \{\langle N \rangle \# x \# 0^t \mid$ N is a NTM accepting x in less than t steps.$\}$ is NP-complete.*

Proof. We first prove that $\mathcal{A}_t \in \mathsf{NP}$. We can recognize $\mathcal{A}_t$ in polynomial time by using an NTM that simulates the t first steps of $M(x)$ in a manner similar to the universal NTM of Theorem 21.

Let $L \in \mathsf{NP}$, N be a NTM accepting L, and π a polynomial such that $t_N \leq \pi$. Consider the function $f : x \to \langle N \rangle \# x \# 0^{\pi(x)}$. It can be computed in polynomial time, and $x \in B \Leftrightarrow N(x)$ accepts x in less than $\pi(x)$ steps $\Leftrightarrow f(x) = \langle N \rangle \# x \# 0^{\pi(x)} \in \mathcal{A}_t$. □

One of the most useful NP-complete problems is the following:

Theorem 28 (Cook's). SAT *is NP-complete.*

Proof. We have already proved that SAT is NP. Let us now prove that it is NP-hard. Consider $L \in \mathtt{NP}$; let N be a NTM accepting L and π a polynomial such that $t_N \leq \pi$.

A SAT formula can be encoded in the alphabet $\{0, 1, \vee, \neg, \wedge\}$ by using an unary representation 0^i of each variable x_i, writing the unary number 0^n of variables before the formula itself, and using the prefix notation on operators with the symbol 1 as a separator between arguments.

As an example, an encoding of the SAT formula $\varphi = \neg x_1 \vee x_2$ is $\langle\varphi\rangle = 001 \vee \neg 0100$.

Note that if the NTM N runs on an input x of size n, then its head can move at most $\pi(n)$ cells to the left or to the right by the end of the computation. Hence, if we index each cell of the NTM's tape with an integer in $\mathbb{Z}$, 0 being the index of the starting head position, at any step of a run on an input of size n, the entire TM's tape configuration lies between the cells indexed by $-\pi(n)$ and $+\pi(n)$.

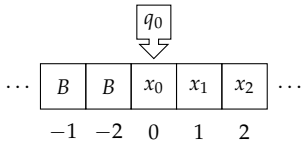

Figure 23: Indexing the cells of the NTM N.

For all $0 \leq j \leq \pi(n)$, for all $-\pi(n) \leq i \leq \pi(n)$, $\forall a \in \Gamma$, $\forall q \in Q$, and $\forall d \in \delta$, we define four Boolean variables a_i^j, q^j, h_i^j, and d^j whose meaning is the following:

— $a_i^j = 1$ if and only if the i-th cell at step j contains a.
— $q^j = 1$ if and only if N is in state q at step j.
— $h_i^j = 1$ if and only if the head points towards cell i of N's tape at step j.
— $d^j = 1$ if and only if rule d is applied at the end of step j.

Our intuition now is to find a formula $\psi(x)$ that constrains these variables in such a manner that there exists a valuation v for which $\psi(x)(v)$ holds true if and only if there exists an accepting run of N on x. Given an input x of size n, we can[36] design in polynomial time the following formulae of polynomial size:

[36] The full proof is left as an exercise to the bravest readers.

— For all $0 \leq j \leq \pi(n)$, a mutual exclusion formula $\varphi^j_{\text{run}}(x) = \bigoplus_{d \in \delta} d^j$ that ensures exactly one rule is applied at step j. Similar exclusion formulae have to be designed to ensure that the content of each cell, the position of the head, and the current state of N are unambiguous at any step j of the computation.
— For all $0 \leq j < \pi(n)$, for all $-\pi(n) < i \leq \pi(n)$, and $\forall d \in \delta$, assuming $d = (p, a) \to (q, b, L)$, a formula $\varphi^j_{d,i}(x) = (d_j \wedge h_{i,j}) \Rightarrow (p^j \wedge a_i^j \wedge q^{j+1} \wedge b_i^{j+1} \wedge h_{i-1}^{j+1})$ stating that in order to apply rule d at step j to cell i, the incoming and resulting configurations have to be compatible with rule d. Similar formulae are to be designed to handle rules that move the head to the right or do not move it at all.
— $\varphi_{\text{start}}(x)$ stating that N starts in state q_0 and that the initial content of the i-th cell is x_i for $-\pi(n) \leq i \leq \pi(n)$.

— $\varphi_{\text{end}}(x) = q_a^{\pi(n)}$ stating that the NTM is in state q_a at the end of step $\pi(n)$.

To ensure that an accepting run always lasts exactly $\pi(n)$ steps, we allow[37] the NTM to loop once it reaches state q_a. The formula $\psi(x)$ is then defined as the conjunction of the formulae introduced previously. By design, the NTM N accepts x if and only if there exists a valuation v for which $\psi(x)(v)$ holds true.

Moreover, $\psi(x)$ uses a polynomial number of variables and features a polynomial number of clauses of polynomial size that can be computed in polynomial time. Finally, $f : x \to \langle \psi(x) \rangle$ is a polynomial reduction of L to SAT. SAT is therefore NP-hard, thus, NP-complete. □

[37] We contradict the formal definition of the transition function as the accepting state is meant to be a sink state. The language accepted by the NTM stays the same, though.

We admit[38] the following consequences of Cook's theorem:

Corrolary 2. CSAT $= \{\varphi \mid \varphi$ *is a satisfiable Boolean formula in conjunctive normal form (CNF).*$\}$ *is* NP*-complete.*

Corrolary 3. 3-SAT $= \{\varphi \mid \varphi$ *is a satisfiable Boolean formula in 3-CNF.*$\}$ *is* NP*-complete.*

[38] A full proof can be found in Sections 10.3 of Hopcroft, Motwani, and Ullman's *Introduction to Automata Theory, Languages, and Computation* (second edition).

In order to prove that a language L is NP-hard, it is common to use a reduction of one of these SAT problems to L. As an example:

Proposition 25. IS *is* NP*-complete.*

Proof. IS is obviously in NP, an independent set being a suitable certificate. We will prove that 3-SAT $\leq_T^p$ IS.

Let $\varphi = \bigwedge_{i=1}^{m} C_i$ be a Boolean formula in 3-CNF on a set of variables $\{x_1, \ldots, x_n\}$, where each C_i is a clause of the form $(y_1^i \vee y_2^i \vee y_3^i)$ such that $\forall j \in \{1,2,3\}$, $\exists k \in \{1, \ldots, n\}$, $(y_j^i = x_k)$ or $(y_j^i = \neg x_k)$.

Consider the set of vertices $V = \{x_k^i \mid i \in \{1, \ldots, m\} \wedge j \in \{1,2,3\} \wedge (y_j^i = x_k)\} \cup \{\neg x_k^i \mid i \in \{1, \ldots, m\} \wedge j \in \{1,2,3\} \wedge (y_j^i = \neg x_k)\}$ representing the various literals of the clauses. Obviously, $|V| = (3 \cdot m)$. Let E be the set of edges such that:

1. Given a clause C_i, the set of vertices matched to the literals of C_i is a 3-clique (i.e. a triangle).
2. If $(y_j^i = x_k)$ and $(y_{j'}^{i'} = \neg x_k)$, then $(x_k^i, \neg x_k^{i'}) \in E$.

We can prove that there is an independent set of size m in the undirected graph $G_\varphi = (V, E)$ if and only if φ is satisfiable.

Intuitively[39], if G has an independent set of size m, then each vertex in the set must belong to a different triangle representing a clause of the whole formula by rule (1). These vertices can then be used to design a valuation of the variables $x_1, \ldots, x_n$ such that the formula φ holds. Rule (2) ensures that we can't have literals matched to x_k and $\neg x_k$ in the same independent set.

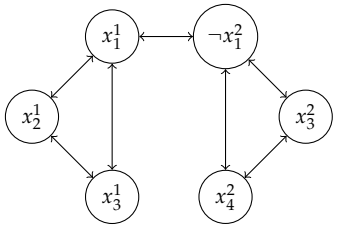

Figure 24: A graph G_φ representing the 3-CNF formula $\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee x_4)$.

[39] The full proof is left as an exercise to the remaining readers.

On the other hand, if φ is satisfiable, then at least one of the literals of each clause C_i must be true. We can therefore pick a vertex v_i in each triangle representing a clause C_i such that $\{v_1, \ldots, v_m\}$ is a independent set of size m. G_φ has $(3 \cdot m)$ vertices and at most $(3 \cdot m)^2$ edges, hence, $\langle G_\varphi \rangle$ can be built from $\langle \varphi \rangle$ in polynomial time.

As a consequence, $f : \langle \varphi \rangle \to \langle G_\varphi \rangle$ is a polynomial-time reduction of `3-SAT` to `IS`. By Proposition 20, `IS` is `NP`-complete. □

Following the previous proposition:

Corrolary 4. `CLIQUE` *is* `NP`*-complete.*

Proof. `CLIQUE` $\in$ `NP` and `IS` can be reduced to `CLIQUE` by complementation. By Proposition 20, `CLIQUE` is `NP`-complete. □

Corrolary 5. `VERTEXCOVER` $= \{(G, k) \mid G$ *has a vertex cover of size* $l \leq k.\}$ *is* `NP`*-complete.*

A *vertex cover* of a graph is a subset of vertices such that each edge is adjacent to at least one of these vertices.

Proof. Note that a graph G with n vertices admits a vertex cover of size l if and only if it admits an independent set of size $n - l$. From there, one can reduce the `NP`-complete problem `IS` to `VERTEXCOVER`.

Moreover, `VERTEXCOVER` is in `NP`, as the vertex cover itself is a certificate that can be checked in polynomial time. □

Going Further

This course was a mere introduction to the theory of computation and complexity. The most eager readers should be looking for material on the following topics:

The Post correspondance problem Also known as the PCP, this is one of the earliest provably undecidable problems ever introduced. Its definition is the following: given two sequences $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$, is there a sequence $i_1, \ldots, i_k$ of integers such that $u_{i_1} \cdot \ldots \cdot u_{i_k} = v_{i_1} \cdot \ldots \cdot v_{i_k}$?

Space complexity The *space complexity* $s_M(n)$ of a Turing machine M is the maximum amount of space it will use on a input of size n. In a similar manner to time complexity, we can define complexity classes such as L (logarithmic space), NL, PSPACE and NPSPACE. The hierarchy $L \subset NL \subset P \subset NP \subset PSPACE = NPSPACE \subset EXP \subset NEXP$ holds.

Probabilistic Turing machines There exists a class of non-deterministic Turing machines such that, at each computation step, the next transition is chosen according to some *probability distribution*. This class of machines features various acceptance conditions and its own complexity hierarchy.

The following reading material is recommended:

Introduction to Automata Theory, Languages, and Computation by John E. Hopcroft, Rajeev Motwani, and Jeffrey Ullman. Chapter 10 of the second edition focuses on NP-completude. Chapter 11 introduces space complexity classes and probabilistic Turing machines.

Introduction to the Theory of Computation by Michael Sipser. Chapter 8 of the second edition focuses on space complexity. Chapter 9 features proofs of various hierarchy theorems.