Handout
Computability, Complexity, Models of Computation

Benjamin Hellouin de Menibus, IUT d'Orsay

Version of January 23, 2023

Foreword

This document is a handout for the class "Computability, Complexity, Models of Computation" taught at the MPRI – university track, Université Paris-Saclay, 2022–2023. Thanks to Emma Caizergues, Atte Torri, Léo Kulinski, Matthieu Robeyns, Mohamed Bassiouni, Pablo Arnault and Paul Patault for various suggestions, improvements, and course notes.

Contents

A few notations

For a finite set of symbols A, called an **alphabet**, A^* denotes the set of **finite words** or **strings** made of symbols from A. For example, an integer written in binary is a word on the alphabet $A = \{0, 1\}$. The empty string is denoted ε.

A **partial function** $f : A \to B$ is a function that, on some inputs, does not have an output; in this case, we write $f(a) = \perp$. A partial function that always has an output is called **total**. In contrast with the usual convention in mathematics, functions that are not always total unless specified.

1 Models of computation and intuitive computability

1.1 Models of computation, Church-Turing thesis

A **model of computation** is a set of formal rules to describe programs. Every program, when given an **input**, computes an **output**, both being finite words on an alphabet A.

A function $A^* \to A^*$ that can be computed by such a program is called **computable** (in this model).

Note. *The choice of A is a part of the model (and often $A = \{0, 1\}$); some models naturally work on $\mathbb{N}$, which amounts to the same. We will see later that this distinction is not very important, through the use of encodings.*

You might know some theoretical models of computation such as Turing machines, λ-calculus, Boolean circuits, recursive functions.... Any programming language, in its way, defines a model of computation[1].

In the computability section of this course, we will use various models depending on the circumstances, and the reason we can do that is because they compute the same functions:

Church-Turing thesis A function that is computable by a reasonable model of computation is computable by a Turing machine. A model that computes exactly the same function as Turing machines is called **Turing-complete**; there are many such models.

This is a general (one might say philosophical) principle; you can understand it as the basic expectation when introducing a new model, which does not mean we should not prove it! There is a lot of leeway around the word "reasonable", but we will use various Turing-complete models (without proving this fact) and non-Turing complete models will be explicitely mentioned as such.

Definition 1. *A function is **computable** if it is computable by any Turing-complete model, such as Turing machines.*

Since most models are equivalent, why should we use different models? They can differ in terms of:

- **expressivity**: programs are easy to write.
- **minimalism**: there are few allowed operations, so proofs are easy to write.
- **practicality**: they are easy to translate to actual hardware (we won't care about that).

Programming languages are robustly Turing-complete, so this point of view lets you write proofs of computability using your favorite language or even pseudocode in a slightly sloppy manner. For example, the following Python program is a reasonable proof that the function $n \mapsto 2^n$ (on positive integers) is computable:

```
def power2 (int n){
        int result = 1
        for i from 1 to n
                result = 2*result
        output result
}
```

[1]This requires us to disregard some practical limits — size of the stack, integers of limited size... — that are present in the actual compiler or interpreter. Rather, imagine the expected behaviour of these programs on an ideal computer with no such software or hardware limitation.

1.2 Turing machines

This section is written in a compact way and is only for fixing conventions and notation; you are expected to know what a Turing machine is.

A **Turing machine** is given by a tuple (n, m, δ), where:

- n is the number of states, m is the number of tape symbols;
- $\delta : \{0, \dots, n-1\} \times \{\#, 0, \dots, m-1\} \to \{0, \dots, n-1\} \times \{\#, 0, \dots, m-1\} \times \{-1, 0, +1\}$ is a **transition function**.

A Turing machine is a program that works as follows. The machine has:

- an infinite **tape** whose every cell contains a symbol in $\{\#, 0, \dots, m-1\}$, where $\#$ is a special "empty" symbol;
- a **head** whose position is given by an integer $p \in \mathbb{N}$;
- a **state** in $\{0, \dots, n-1\}$.

A **configuration** is a tuple (x, p, e) with x a finite word, p a head position, and e a state. It means that the tape contains x followed by empty symbols.

- The machine receives as **input** some $x \in \{1, \dots, m\}^*$. The initial configuration (at time 0) is $M^0(x) = (x, 1, 0)$.
- Assume that we are at time t in configuration $M^t(x) = (x, p, e)$. Let $i, d, e' = \delta(x_p, e)$, where x_p is the tape symbol at position p. The configuration at time $t+1$ is $M^{t+1}(x) = (x', p', e')$, where:
  - x' is equal to x except that $x'_p = i$;
  - $p' = p + d$ (the head moves).
- Whenever $M^t(x) = (x, p, n-1)$ ($n-1$ is the last state), the machine **halts**: the computation is complete and the **output** is x, the nonempty part of the tape. After this point, the configuration no longer changes.

Note. *You have probably encountered alternative definitions for Turing machines: different symbols, more tapes... This does not change the Turing-completeness of the model, so this is a matter of taste and the computed functions are the same (again, the Church-Turing thesis).*

1.3 Computable functions

For a Turing machine M on input x, we denote:

- $M(x)\downarrow$ if the machine eventually halts. In this case, we write $M(x)\downarrow = y$ (and say "on input x, M halts and outputs y") where $y \in A^*$ is the word written on the tape in the halting configuration.
- $M(x)\uparrow$ if the machine never halts.

Turing machines compute, in general, partial functions: there is no output if the machine doesn't halt.

Definition 2. *A Turing machine M* ***computes*** *a function $f : \{0, \dots, m-1\}^* \to \{0, \dots, m-1\}^*$ if for any input x, we have:*

$$f(x) = \bot \Rightarrow M(x)\uparrow$$
$$f(x) \neq \bot \Rightarrow f(x) = M(x)\downarrow .$$

In other words,

- *if $f(x)$ is defined, the machine must halt on input x and output the expected output;*
- *if $f(x)$ is not defined, the machine never halts (or it "loops") on input x.*

Now is the proper time to make a clear distinction in your head between functions and programs (= Turing machines). A program always computes a single function, but a function may be computed by no program, or by many of them. A function doesn't halt or loop and it always has a value (even if it is $\bot$).

1.4 A few operations on computable functions

This intuitive point of view on computability might be slighlty jarring if you are used to more solid mathematical frameworks. I choose to do it this way because writing programs in Turing machines or other minimalistic models is tedious and does not help to build the simple intuition that a computable function is just a function for which you can write a program.

It should be intuitive that computability is preserved by the usual operations on booleans, strings or integers. such as addition, concatenation, composition, logic operators...

Let us do a proof for, say, composition. Assume f and g are computable functions, and write the program:

```
program f\circ g(int n){
        int m = P_g(n)
        output P_f(m)
}
```

where `P_f` and `P_g` are the programs that compute f and g. This program clearly computes $f \circ g$. Notice that if either $g(n) = \bot$ or $f(g(n)) = \bot$, the whole program loops: this seems like a reasonable result when you consider the definition of composing two partial functions.

If we wanted to write a formal proof, we would use a simple well-defined model such as Turing machines.

Exercice 1. *Assume that f and g are total computable functions $\mathbb{N} \to \mathbb{N}$. Which of the following functions are computable? For which can't you tell in general?*

1. *$n \mapsto 1$ if $f(n) = g(n)$, 0 otherwise.*
2. *$n \mapsto$ the smallest $m > n$ such that $f(m) = 0$.*
3. *$n \mapsto f^{-1}(n)$ (if you know that f is a bijection).*

2 Encodings, cardinality issues and diagonalisation

2.1 Encodings and countability

For an object to be handled by a computer program, it must be represented in memory — **encoded** — by a finite number of symbols. This is the case for inputs and outputs of Turing machines[2], and in fact, for every other model that we will use. Sets whose objects can be represented by a finite amount of bits are said to be **countable**.

Definition 3. *A set X is **countable** if, for some alphabet A, one of the two following equivalent conditions holds:*

- *there is an **encoding** of X in A^*, that is, an injection $\varphi_X : X \to A^*$;*
- *there is a **decoding** from A^* to X, that is, a surjection $\varphi_X^{-1} : A^* \to X$.*

As the notation suggests, encoding then decoding should output the initial value. The injectivity / surjectivity condition means that every object from X can be represented by one word in A^*, or possibly several. It is fine if a word in A^* encodes nothing, but two different objects in X cannot have the same encoding.

Any finite set X is of course countable; if X is infinite, we can even find a bijection. $X = \mathbb{N}$ can be encoded on $\{0, 1\}$ through the binary encoding, or on $\{0, \dots, 9\}$ through the decimal encoding, or many other ways.

An important equivalent definition is the following:

Definition 4. *A set X is **countable** if there is an **enumeration** of X, i.e. a surjection $\mathbb{N} \to X$, or (equivalently) an encoding of X in $\mathbb{N}$, i.e. an injection $X \to \mathbb{N}$.*

If X is infinite, the enumeration gives us a list $x_1, x_2, x_3 \dots$ that lists all objects is X.

Countability as "the ability to represent objects using a finite number of symbols" should make it an intuitive notion. To train this intuition, let us do a formal proof.

Theorem 1. *Let D and D' be two countable sets. Then $D \times D'$ is countable.*

Proof. Intuitively, we can encode an element $(d, d') \in D \times D'$ by concatenating an encoding of d to an encoding of d'. To make sure we can decode unambiguously, we add a fresh comma symbol **,** to mark the limit between the encodings.

Formally, let $\varphi_D : D \to S^*$ be the encoding for D and $\varphi_{D'} : D' \to S'^*$ for D'. We define $\varphi_{D \times D'}$ as the encoding that, on input (d, d'), outputs the string $\varphi_D(d), \varphi_{D'}(d')$. Let us check that $\varphi_{D \times D'}$ is an injection: take two pairs $(a, a') \neq (b, b')$. If $a \neq b$, then the prefixes of $\varphi_{D \times D'}(a, a')$ and $\varphi_{D \times D'}(b, b')$ that come before the comma are different, so $\varphi_{D \times D'}(a, a') \neq \varphi_{D \times D'}(b, b')$. Similarly if $a' \neq b'$. □

[2] You can consider the behaviour of Turing machines on infinite inputs, but it does not make sense in the standard notion of computability where you halt after a finite number of steps.

2.2 Encodings and computability

We have defined computable functions on words (strings). We can now extend the definition to functions $D \to D'$, where D and D' are countable sets, by encoding members of D and D' as words.

Definition 5. *Let φ_D and φ'_D be encodings of D and D', respectively. A function $f : D \to D'$ is computable if the function $\varphi_D(x) \mapsto \varphi'_D(f(x))$ is computable (the latter function works on words, so we can use the previous definition).*

This makes intuitive sense: if you want to compute a function $\mathbb{N} \to \mathbb{N}$, you first encode the input in binary, execute the program, and decode the output to get the result.

This means that, outside of strings, the definition of computability may depend on the choice of encoding. In practice, this is not a problem and the definition is invariant for any reasonable encoding[3]. For example, computable functions are the same when written in binary and decimal.

Note. *In the definition, to compute the function $\varphi_D(x) \mapsto \varphi'_D(f(x))$, notice that not all words in A^* are acceptable inputs (they may not encode an object from D — for example, in the binary encoding, we usually forbid leading zeroes). In a program computing this function, the behaviour for these inputs is undefined and it may do whatever it wants.*

Exercice 2. *Among the following functions, for which does it make sense to ask whether they are computable?*

$\mathbb{Z} \to \mathbb{Z}$ *:* $n \mapsto -n$*;*

$\mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ *:* $(a, b) \mapsto (a + b, a - b)$*;*

$\mathbb{N} \to \mathbb{Q}$ *:* $n \mapsto 1/n$*;*

$\mathbb{N} \to \mathbb{R}$ *:* $x \to \sin(x)$.

2.3 Uncomputable functions: the diagonalisation technique

Here we take a small mathematical detour to see how the countability hypothesis on inputs and outputs restricts the universe of computable functions.

Theorem 2. *The set of computable functions is countable.*

Proof. For any computable function, there exists a program that computes it. This program is a finite string. Therefore we can encode any computable function by the shortest program that computes it. Since each program computes a single function, this is an injection. □

This proof works for any model. By the alternative definition, we can enumerate programs $P_0, P_1 \ldots$.

Exercice 3. *Find an encoding of Turing machines.*

To prove the existence of noncomputable functions, we show that there are more functions than programs. We do the following result for $\mathbb{N} \to \mathbb{N}$ but it would work for any countable infinite sets.

[3] To get a formal proof, we would need to prove that the function that switches encodings is computable.

Theorem 3. *The set of total functions $\mathbb{N} \to \mathbb{N}$ is uncountable. It follows that the set of partial functions is also uncountable (there are even more of them).*

The following proof uses a very important technique called **diagonalisation**.

Proof. By contradiction, suppose that the set of total functions $\mathbb{N} \to \mathbb{N}$ is countable. Let us enumerate them as $f_0, f_1 \dots$ (this is a surjection, so all total functions should appear in the list). Let us define:

$$F : \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & f_n(n) + 1 \end{array}$$

F is total, and I claim that F does not appear in the list. For any k, we have by definition $F(k) \neq f_k(k)$, and therefore $F \neq f_k$. This is a contradiction. □

Exercice 4. *Using the same proof method, show that the set of real numbers is uncountable.*

Again, the following theorem holds for any infinite countable sets.

Theorem 4. *There is an uncomputable function $\{0,1\}^* \to \{0,1\}^*$.*

Proof. The set of functions $\{0,1\}^* \to \{0,1\}^*$ is uncountable, and the set of computable functions $\{0,1\}^* \to \{0,1\}^*$ is countable. Therefore there must be a noncomputable function; in fact, uncountably many of them. □

The noncomputable function built by the diagonalisation method is not very explicit. We will see more concrete examples later.

This result offers a new perspective on classical mathematical objects. All integers can be described using finite information (a finite word), and the same is true for computable functions (a program), but there is no language that allows us to describe all functions $\mathbb{N} \to \mathbb{N}$ using finite words.

However, you have probably already encountered many functions that had a finite description, such as $f : n \mapsto 2n$. The functions that are explicitely defined and used in mathematics have a finite description, so you only ever meet a countable subset of functions $\mathbb{N} \to \mathbb{N}$. These functions are often computable, but not always.

2.4 Computable objects and representations

Up to now, we have defined computability only for functions. In order to talk about computability for other objects, such as sets or languages, we provide a **representation** of the object in the form of a function that contains the same information and whose input and output are countable.

Note. *Try to keep in mind the difference between:*

- *an **encoding**: representing objects as strings so that they can be handled as input–output by programs;*
- *a **representation**: representing objects as functions so that they can be considered computable (or not).*

Exercice 5. *Among the following objects, try to make sense of the question of whether they are computable. That is, can you represent them as a function whose input and output are countable?*

1. *the integer 7;*
2. *the boolean representing the truth value (0 or 1) of Riemann's conjecture;*
3. *the following problem: given a graph, find a Hamiltonian cycle?;*
4. *the language $\mathcal{L} \subset \{0,1\}^*$ of palindromes;*
5. *the subset $P \subset \mathbb{N}$ of prime numbers;*
6. *the real number π.*

1. Representations for a single integer almost don't make sense; every representation works and should show that all integers are computable. For example, the integer n can be represented by the function $\{\varepsilon\} \to \mathbb{N} : \varepsilon \mapsto n$.
2. Notice that this is not a function: this is a single boolean, even if it depends on the world in which we live. Just like an integer, a boolean can be represented by a function that doesn't read its input (always true, or always false). We don't know which function it is, but in any case there is a program that computes the correct boolean, so it is computable.

For these two examples, and more generally for individual objects with a finite representation, the notion of computability is degenerate (everything is computable). Before explaining the next examples, let me underline that, in contrast with encodings, the definition of computability can and often does depend on the choice of representation.

In practice, we agree on a standard representation to define, for each kind of object, when it is computable. In the case that another representation proves useful, this is another notion of computability with a different name. Here are some standard representations:

Definition 6. *In the following, X is a countable set.*

- *A problem (given $x \in X$, does x satisfy the property $\mathcal{P}$?) is represented by its boolean function $X \to \{0,1\}$.*
- *A set $S \subset X$ is represented by its indicator function $1_S : X \to \{0,1\}$, defined by $1_S(x) = 1 \Leftrightarrow x \in S$.*
- *A language $\mathcal{L} \in \{0,1\}^*$ is a type of set, so the previous definition applies.*
- *A real number r is represented by an approximation function $a_r : \mathbb{N} \to \mathbb{Q}$ such that $|r - a_r(n)| \leqslant 2^{-n}$.*

The definition for real numbers is perhaps the least intuitive. We will talk a bit more about computations on real numbers in a later section.

Names It is usual to say that a function is computable, that a problem or question is decidable, that a language or set is recursive. This is historical baggage, and you can consider these words as synonyms.

Now let us see another natural representation for sets that yields another definition for computability:

Definition 7. *A set $S \subset X$ is **computably enumerable**[4], c.e. for short, if there is a computable surjection $f : \mathbb{N} \to S$.*

We already called such a function an **enumeration** in Definition 4. Caution: in contrast to Definition 4, we can't use alternative definitions here.

Theorem 5. *If S is computable, then S is computably enumerable.*

Proof. Let us do the proof when $S \subset \mathbb{N}$. Since S is computable, by definition, the function 1_S is computed by some program `P_S`. Consider the following program:

```
program enum(int n){
        int count = 0;
        for i from 0 to infinity:
                if P_S(i):
                        count += 1;
                if count = n:
                        output i;
}
```

On input $n \in \mathbb{N}$, it ouputs the n-th element of S, in increasing order. It should be clear that it is a surjection on S. Notice that the program may loop if S has less than n elements. This is not a concern. □

A computably enumerable set is not necessarily computable. We will prove this later, when we have the tools for it.

3 Programs working on programs, reductions, oracles

3.1 Programs as input and output

We mentioned earlier that programs form a countable set. It follows that a program can be given as input or output to another program, encoded as a string.

You can imagine that programs are manipulated through their source code, which is a string. This is not such a strange idea: a compiler, for example, receives as input a program and outputs a program (in different languages).

Definition 8. *A **universal program** U is a program that, given as input a program P and a word x, behaves as follows:*

– *if $P(x)\!\uparrow$, then $U(P, x)\!\uparrow$;*

[4] Often called **recursively enumerable** — remember my above remark.

– *if* $P(x)\downarrow$*, then* $U(P,x)\downarrow = P(x)\downarrow$.

U is able to simulate the behaviour of any program given as input. In actual computing, U is said to execute the program, and you find such functions (often called **exec**) in various programming languages.

3.2 The halting problem

The halting problem (for programs) consists in deciding, given a program and a string as input, whether the program halts on the input. Formally:

$$\begin{array}{rccl} & \mathcal{P} \times A^* & \to & \{0,1\} \\ \text{halt}: & (P,x) & \mapsto & \begin{cases} 1 \text{ if } P(x)\downarrow \\ 0 \text{ if } P(x)\uparrow \end{cases} \end{array}$$

Theorem 6. *The halting problem is undecidable (= uncomputable).*

Proof. For the sake of contradiction, assume that the halting problem can be computed by a program `stop`. Consider the following program:

```
program f(program P){
        if stop(P, P) = 0
                output 0
        else
                loop
}
```

Now ask yourself: what is the output of $\texttt{stop}(\texttt{f},\texttt{f})$? If $\texttt{stop}(\texttt{f},\texttt{f}) = 0$, this means that $\texttt{f}(\texttt{f})\downarrow = 0$; if $\texttt{stop}(\texttt{f},\texttt{f}) = 1$, then $\texttt{f}(\texttt{f})\uparrow$. But in both cases, this is a contradiction with the definition of `stop`. □

This may seem a funny result with a magical proof, but the impossibility of deciding the halting problem has far-reaching consequences in actual problems of program verification. We will see in a moment that there is nothing specific about halting, and that most problems on programs are undecidable.

3.3 Turing reductions

To find new uncomputable functions, we develop a tool to compare the difficulty of computing different functions. Intuitively, f is "easier to compute" than g if

> If I can get the values of $g(x)$ for any x, I can write a program that computes f.

This means, in particular, that if I has a program that computes g, then I could write a program that computes f.

Let us take a toy example[5]. Imagine we are in a strange model working on integers where the allowed operations are plus, minus, and division by 2. Which is easier, multiplying or squaring?

[5]Thanks to `https://fr.wikipedia.org/wiki/Réduction_(complexité)`

– Squaring is easier than multiplying, since $\texttt{square}(x) = \texttt{mult}(x, x)$.

– Multiplying is easier than squaring, since $\texttt{mult}(x, y) = \frac{\texttt{square}(x+y)-\texttt{square}(x)-\texttt{square}(y)}{2}$.

We are only making comparative statements: we have no reason to expect that any of these functions is computable in this strange model, but we can nevertheless compare their difficulty. What did we do? We wrote a special kind of program, called a **reduction**, that gets access to an additional operation in addition to those allowed in the model. We say the other function is **given as an oracle**.

Let us do a formal definition.

Definition 9 (Turing reduction)**.** *Let f and g be two functions $A^* \to A^*$. We say that f is **Turing-reducible** to g, and we write $f \leqslant_T g$, if f can be computed given g as an oracle. In other words, f is computable if we add to the model a new operation that computes g (as a "black box").*

The notation $f \leqslant_T g$ can be thought of as "f is easier (or as difficult) than g".

We define similarly $\geqslant_T, \equiv_T, <_T \ldots$.

When writing pseudocode, receiving g as an oracle allows us to write operations such as $y = g(x)$ regardless of whether g is computable or not. On Turing machines, this can be formalised using an additional **oracle tape** and a special **oracle state**; when the computation enters the oracle state, the machine "magically" replaces the contents x of the oracle tape by $g(x)$.

The motivation for Turing reduction is the following theorem:

Theorem 7. *If $f \leqslant_T g$ with g computable, then f is computable.*

If $f \leqslant_T g$ with f uncomputable, then g is uncomputable.

Proof. $f \leqslant_T g$ means that there is a program P receiving g as an oracle that computes f. Since g is computable, there is a program P_g that computes g. Every time the operation to compute g is used in P, we replace it with a call to the program P_g. We obtain a program with no oracle that computes f. □

It is important to understand that uncomputable functions can be compared, and some could be "more uncomputable" that others. There is in fact a whole theme of research about the mathematical structure of $\leqslant_T$ and $\equiv_T$ and their equivalence classes (**Turing degrees**).

Turing reduction is just one kind of reduction, which is suited to computability and uncomputability, but not (for example) to P̨ and N̨, for the following reasons:

Exercice 6. *Prouvez que deux fonctions calculables sont toujours Turing-équivalentes ($\equiv_T$).*

In other contexts, we will introduce other kinds of reductions.

3.4 Rice's theorem

Let us put this new notion of reduction to good use.

Theorem 8 (Rice's theorem)**.** *Let T be a nontrivial property on computable functions (not always true or false).*

Then halt $\leqslant_T T$*. In particular, T in uncomputable.*

It seems interesting that halting, despite being uncomputable, is the easiest property on computable functions.

This theorem is deceptively simple-looking and there are quite a few devils in the details. Since the inputs of T are computable functions, it is natural to encode them by a program that computes them. However, T is not a property of the program, it is a property of the function. For the following examples, ascertain if they are properties of the function or of the program, and get an intuition of whether they seem undecidable.

- Do we have $f(\varepsilon) = \bot$? (the empty input)
- Is there a while loop somewhere?
- Can we find an input x such that $f(x) \neq 2$?
- Does the program loop on every input?

Another formulation is that the theorem applies to **semantic** properties — their meaning, i.e. the computed function – and not **syntactic** properties — what is actually written in the program.

Proof. Define $\infty : x \mapsto \bot$, and assume that it does not satisfy T (the other case is symmetric). Since T is not trivial, there is a function f that satisfies T.

Now, we will define a function ψ that transforms one program into another. Given as input a program P and a string x, ψ outputs the following program:

```
program (input y){
        P(x);
        output f(y);
}
```

- ψ is computable: simply write down the above code and replace x by its value and P by its program (both given as input).
- if $P(x)\downarrow$, then $\psi(P, x)$ computes f and satisfies T. If $P(x)\uparrow$, $\psi(P, x)\uparrow$, so $\psi(P, x)$ computes ∞ and does not satisfy T.

We are ready for the reduction that proves halt $\leqslant_T T$.

```
program stop(program P, string x) oracle T{
        if T(ψ(P,x)): output 1
        else: output 0
}
```

□

3.5 A few reduction exercises

Exercice 7. *Are the following problems computable or uncomputable?*

$$=_1 : \begin{array}{rcl} \mathcal{M} \times \mathcal{M} \times \{0,1\}^* & \to & \{0,1\} \\ (f, g, x) & \mapsto & \begin{cases} 1 \text{ if } f(x)\downarrow = g(x)\downarrow \\ 0 \text{ if } f(x)\downarrow \neq g(x)\downarrow \end{cases} \end{array}$$

$$=_2 : \begin{array}{rcl} \mathcal{M} \times \mathcal{M} \times \{0,1\}^* & \to & \{0,1\} \\ (f,g,x) & \mapsto & \begin{cases} 1 \text{ if } f(x)\downarrow = g(x)\downarrow \\ 0 \text{ otherwise.} \end{cases} \end{array}$$

Exercice 8. *Are the following functions computable or uncomputable?*

1. *Halting on empty input:* $P \mapsto 1$ *if* $P(\varepsilon)\downarrow$, 0 *if* $P(\varepsilon)\uparrow$.
2. *Halting for* P: $x \mapsto 1$ *if* $P(x)\downarrow$, 0 *if* $P(x)\uparrow$.
3. *Halting before* k *steps:* $(P, n, k) \mapsto 1$ *if* $P(x)$ *halted before* k *steps of computation,* 0 *otherwise.*
4. *Dead code detection: let* P *be a program in* **<your favourite language>** *and* f *the name of a subprogram. Is* f *called when* P *is executed ?*

3.6 Computability beyong the halting problem

Are there any functions strictly harder than the halting problem, or are all uncomputable functions Turing-equivalent?

Definition 10. *Let* $g : A^* \to A^*$ *be an arbitrary function. Denote* halt^g *the halting problem on programs that receive* g *as an oracle. That is, for any program* P *that receives an oracle* g *and any input* x, $\text{halt}^g(P, x) = 1$ *if* $P(x)\downarrow$, *and* 0 *if* $P(x)\uparrow$.

Theorem 9 (Turing jump)**.** *For any function* g, $\text{halt}^g >_T g$.

This means that $\text{halt}^g \geqslant_T g$ and $\text{halt}^g \not\leqslant_T g$.

Proof. We define a program p that receives as input two strings x, y and outputs a program:

```
program () oracle g{
        if g(x) = y: output 1
        else: loop
}
```

The reduction $\text{halt}^g \geqslant_T g$ is:

```
program (string x) oracle halt^g{
        for any string y:
                if halt^g(prog, x, y): output y
}
```

To show that $\text{halt}^g \not\leqslant_T g$ (strict inequality), do the same proof as the halting problem with programs that receive g as an oracle. □

4 Computable enumerations

4.1 Generalities

We saw earlier that sets, languages and decision problems (all objects that are represented by functions with image in $\{0,1\}$) have different representations that lead to different notions of computability:

– by the indicator function 1_X (if it is computable, the set is **computable**)

– by some enumeration, i.e. a surjection $enum_X : \mathbb{N} \to X$ (if one such enumeration is computable, the set is **computably enumerable**).

There are other reasonable representations but they mostly end up equivalent to these two. Notice that the second definition uses the notion of enumeration that comes from the definition of a countable set (Definition 4), so you could call this notion "computably countable" instead. Watch out: you cannot use the inverse function here (injection $X \to \mathbb{N}$).

We also made a note that we are now able to reformulate as follows:

Theorem 10. *Let X be a set, 1_X its indicator function, $enum_X$ an enumeration of X. Then $enum_X \leqslant_T 1_X$. It follows that a computable set is computably enumerable.*

Before going further, we prove an equivalent definition:

Theorem 11. *A set X is computably enumerable if, and only if, the partial indicator function $1_X^\perp$ is computable, where $1_X^\perp : x \mapsto 1$ if $x \in X$, $\perp$ if $x \notin X$.*

This theorem gives us a lot of insight into the difference between computable and computably enumerable. Before doing the proof, note that X is computable if and only if X^c (its complement) is. Indeed, you can exchange 0 and 1 in the output of a program that computes 1_X. However, this theorem shows that being computably enumerable is **not symmetrical** by exchanging 0 and 1.

If X^c is c.e., we say that X is **co-computably enumerable**.

Proof. **(⇐)** Given a computable enumeration $enum_X$, the following program computes $1_X^\perp$:

```
program 1 (input x){
        for all n from 0 to ∞:
                if enum_X(n) = x:
                        output 1
}
```

It outputs 1 if $x \in X$ (because $enum_X$ is a surjection), and loops otherwise.

(⇒) Assume that $1_X^\perp$ is computable. The following program computes an enumeration of X:

```
program enum_X (input n){
    S = []
    for all t from 0 to ∞:
        for all strings x of length ⩽ t:
            simulate 1_X^⊥(x) during t steps of computations
            if this simulation stops:
                add x to S
                if S contains n elements: output S[n]
}
```

If $x \in X$, then $1_X^{\downarrow}(x)$ will stop after some time t. When n is large enough, $enum_X(n)$ will enter the t-th step of the loop and output x. So $enum_X$ is an enumeration (= surjection on X). □

Exercice 9. *Show that the set $S = \{(P, x) \in A^* \times A^* : P(x)\downarrow\}$ is computably enumerable but is not computable.*

Theorem 12. *If X is c.e. and co-c.e., then X is computable.*

Proof. We have two computable enumerations: ϕ that enumerates elements from X, and ϕ^c that enumerates elements from X^c. This means that each element of $\{0,1\}^*$ is enumerated by exactly one of the functions. Therefore, the following program:

```
program f(input x){
    for all i from 0 to ∞:
                if x = φ(i): output 1
                if x = ψ(i): output 0
}
```

always halts and computes 1_X. □

This world of asymmetrical computability, where 0 and 1 play very different roles, can only exist for sets, languages or decision problems, and more generally functions with image in $\{0,1\}$

4.2 Strong (many-one) reductions

If we take a function $f : X \to \{0,1\}$ (a decision problem, the indicator function of a set...), and $\bar{f}$ the opposite function obtained by switching 0 and 1, then it is clear that $f \equiv_T \bar{f}$. However, it is possible that f is c.e. and not $\bar{f}$ (this is the case whenever f is not computable). This tells us that Turing reduction is too coarse to work with computable enumerations.

Definition 11 (Strong (many-one) reduction)**.** *Let f and g be two functions $A^* \to A^*$. We say that f is strongly (or many-one)-reducible to g, and we write $f \leqslant_m g$, if f is computable by a program receiving g as an oracle where the program can use g only once at the end of the program, without changing the result.*

A perhaps more formal equivalent definition is that there is a computable function p such that $f(x) = g \circ p(x)$ for all input x.

By definition it should be clear that many-one reductions are stronger than Turing reductions, that is, $f \leqslant_m g \Rightarrow f \leqslant_T g$. In particular, just like in Turing reduction, if $f \leqslant_m g$ and g is computable, then f is computable. What is new is the following result:

Theorem 13. *If g is c.e. and $f \leqslant_m g$, then f is c.e.; the same is true for co-c.e.*

This means that strong reductions are the right kind of reduction to work in asymmetrical computability.

5 Recursive and primitive recursive functions

Recursive functions are a so-called functional model of computation, instead of a "mechanical" definition where a computable function is some combination of elementary computing operations on an input. Here, the definition consists in some base functions that are assumed computable, and some operations on functions that can be "done by hand" and thus preserve computability. For this model, the functions are $\mathbb{N}^k \to \mathbb{N}$.

5.1 Primitive recursive functions

Definition 12. *Primitive recursive functions are the following base functions:*

Constant 0*;*

Successor $n \mapsto n+1$*;*

Projections $\pi_i^k : x_1, \ldots, x_k \to x_i$*,*

that can be combined with the following operations:

Composition *if* $f, g_1, \ldots, g_k$ *are primitive recursive, then* $h(x) = f(g_1(x), \ldots, g_k(x))$ *are.*

Primitive recursion *if* f *and* g *are primitive recursive, then the recursively defined function:*

$$h(0, x) = f(x)$$
$$h(n+1, x) = g(h(n, x), n, x)$$

is primitive recursive.

Exercice 10. *Show that the following functions are primitive recursive:*

1. *addition* $add : a, b \mapsto a + b$
2. *multiplication* $mult : a, b \mapsto a \times b$

Notice that primitive recursive functions always halt. Indeed, primitive recursion is "controlled" in the sense that the first argument tells you from the beginning how many iterations there will be in the loop. This is equivalent to a `for` loop (where the index can't be changed inside the loop).

It turns out that such a controlled recursion is not enough for Turing-completeness, even for total functions.

Theorem 14. *There is a computable total function that is not primitive recursive.*

Proof. First convince yourself that the set of primitive recursive functions is countable. Let $f_0, f_1 \ldots$ be a computable enumeration of this set.

It is not hard to write a universal program U for primitive recursive functions: given a primitive recursive function f and an integer n, $U(f, n)\!\downarrow = f(n)$.

By diagonalisation, the function computed by the following program:

```
program diago(int n){
        output U(f_n,n)+1
}
```

cannot be primitive recursive. □

This argument is much more general and applies to any model of computation whose programs always halts. Even more generally, any model of computation whose halting problem is decidable cannot be Turing-complete.

This has deep consequences. The undecidability of the halting problem is not a weird side effect of our models of computation but a necessary aspect. If you were given a mechanism that prevents you from writing infinite loops, you can be sure it would also wrongly prevent you from writing some programs that do halt, but where the mechanism is too coarse to understand why.

5.2 The Ackermann function

Definition 13. *The* ***Ackermann function*** $\mathrm{Ack} : \mathbb{N}^2 \to \mathbb{N}$ *is defined as follows:*

- $\mathrm{Ack}(0, p) = p + 1$
- $\mathrm{Ack}(n + 1, 0) = \mathrm{Ack}(n, 1)$
- $\mathrm{Ack}(n + 1, p + 1) = \mathrm{Ack}(n, \mathrm{Ack}(n + 1, p))$

Exercice 11. *Show that the function is well-defined (the definition doesn't loop).*

Theorem 15. *The Ackermann function is not primitive recursive.*

Proof. You can easily prove by induction the following properties: $\mathrm{Ack}(2, x) = 2x + 3$, Ack is increasing relative to both inputs, $\mathrm{Ack}(x, y + 1) \leqslant \mathrm{Ack}(x + 1, y)$.

We will show that for any primitive recursive function f there is a t such that:

$$f(x_1, \ldots, x_n) \leqslant \mathrm{Ack}(t, \max(x_i)).$$

This proves that no such f can compute the Ackermann function.

Base functions The property is true with $t = 0$ for the three base functions.

Composition Let us assume it is true for some t for functions f and $g_1, \ldots, g_n$. Then:

$$h(x) = f(g_1(x), \ldots, g_k(x)) \leqslant \mathrm{Ack}(t, \mathrm{Ack}(t, x)) \leqslant \mathrm{Ack}(t + 1, x + 1) \leqslant \mathrm{Ack}(t + 2, x)$$

Récursion primitive Let us assume that it is true for some t for functions f and g. Let us show it is true for $t + 4$ for the function h.

First, let us show by induction on n that for all n, $h(n, x) \leqslant \mathrm{Ack}(t + 1, n + x)$:

- $h(0, x) = f(x) \leqslant \mathrm{Ack}(t, x)$.
- Assume it is true for some n. Then $h(n + 1, x) = g(h(n, x), n, x) \leqslant \mathrm{Ack}(t, \mathrm{Ack}(t + 1, x + n)) = \mathrm{Ack}(t + 1, x + n + 1)$.

To conclude, denoting $m = \max(x, n)$, we have $\mathrm{Ack}(t + 1, n + x) \leqslant \mathrm{Ack}(t + 1, 2m) \leqslant \mathrm{Ack}(t + 1, \mathrm{Ack}(2, m)) \leqslant \mathrm{Ack}(t + 3, m + 1) \leqslant \mathrm{Ack}(t + 4, m)$. □

5.3 Recursive functions

A recursive function is a function obtained from the previous operations and:

Minimisation If f is a recursive function, then

$$\mu(f)(x) = \min\{n \in \mathbb{N} : f(n, x) = 0\} \quad \text{or} \quad \perp \quad \text{if min doesn't exist}$$

is also recursive.

Theorem 16. *The set of recursive functions is the set of computable functions.*

This is just another example of the Church-Turing thesis.

6 Complexity classes

6.1 General overview

Complexity theory is the study of functions that are computable with limited resources. It is motivated by the performance analysis of algorithms in a large sense. Depending on the context, this study may focus on different kinds of resources: time or number of steps before an algorithm halts, number of variables or amount of memory, number of components (logic gates, states...), amount of communication (network algorithms), how many times some operation is necessary (e.g. comparison for list sorting).

In the computability part of this course, we tried as much as possible to forget about the underlying model, because all reasonable models define the same set of computable functions (Church-Turing thesis). In the complexity part, the notion of ressource is an aspect of the model and most classes are not robust to a change of model (if the class even makes sense for another model).

There is a lot of reflection involved in which models and resources best capture real-life performance bottlenecks in actual applications, but there is also a lot of theoretical work to understand the structure of different types of classes. It is always great when two classes coming from different models end up equal. In any case, remember that all the functions we consider from now on are computable.

For objects that are not functions — sets, languages, real numbers — we study their complexity by using representations, as explained in Section 2.4. In complexity, the choice of representations **and** encodings may change the class, and it is a good habit to think about it.

6.2 Time– and space–bounded classes

Definition 14 (Notations o and O)**.** *Denote $f(n) = o(g(n))$ (and say "f is a small o of g") when $\frac{f(n)}{g(n)} \to 0$.*

Denote $f(n) = O(g(n))$ (and say "f is a big O of g") if there is a constant $C > 0$ such that $f(n) \leqslant Cg(n)$.

We begin with the classical Turing machine model.

Definition 15 (Time complexity)**.** *Let $T : \mathbb{N} \to \mathbb{N}$.*

*A function f is **computable in time** $T(n)$ if there exists a Turing machine that, on any input w of length n, halts after $T(n)$ steps of computation at most and outputs $f(w)$. This is denoted $f \in TIME(T(n))$.*

- $\mathrm{P} = \mathrm{PTIME} = \bigcup_{k\in\mathbb{N}} \mathrm{TIME}(n^k)$
- $\mathrm{EXPTIME} = \bigcup_{k\in\mathbb{N}} \mathrm{TIME}(2^{n^k})$

For real algorithms, it is also usual to talk about linear complexity $O(n)$, quadratic complexity $O(n^2)$, quasi-linear complexity $O(n \log n)\ldots$

You may wonder why we did not define a class $\mathrm{LOGTIME} = \mathrm{TIME}(\log n)$. A logarithmic time does not allow a Turing machine to read its input, which seems a bit absurd. There are ways to make sense of this notion by considering another model: RAM machines (Random Access Memory), that can access any address in memory in one step of computation. In this model, dichotomic search is in logarithmic time as one would expect. Vous vous demandez peut-être pourquoi on ne définit pas une classe $\mathrm{LOGTIME} = \mathrm{TIME}(\log n)$. Un temps logarithmique ne laisse pas à une machine de Turing le temps de lire son entrée, ce qui semble un peu absurde. Il est possible de donner un sens à cette notion dans un modèle différent : les machines RAM (Random Access Memory), qui peuvent accéder à n'importe quelle case mémoire en une étape de calcul. Dans ce modèle, la recherche dichotomique termine en temps logarithmique comme on s'y attend.

Il s'agit d'un exemple d'un phénomène plus large : la complexité en temps sous-linéaire est très sensible aux détails du modèle. La situation est meilleure pour des classes plus grandes telles que P, et on a une version plus forte de la thèse de Church-Turing :

Church-Turing thesis for time complexity All Turing-complete models with a reasonable notion of time can simulate each other with polyomial overhead. In particular, they define the same classes P and EXPTIME.

Polynomial overhead is a lot if you're doing computation on a real computer, but it's perfectly fine for theoreticians who work on $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$, for example.

Definition 16 (Space complexity)**.** *A function f is **computable in space** $S(n)$ if there exists a Turing machine that, on any input w of length n, halts and outputs $f(w)$ and never writes outside of the first $S(n)$ memory tape cells except for writing the output. This is denoted $f \in SPACE(S(n))$.*

The sentence "except for writing the output" can be formalised by having another write-only tape called "output tape" . The intuition is that space complexity measures the memory necessary to perform a computation, and having a program "pay" for the size of its output does not correspond to this intuition.

- $L = \mathrm{LOGSPACE} = \mathrm{SPACE}(\log n)$
- $\mathrm{PSPACE} = \bigcup_{k\in\mathbb{N}} \mathrm{SPACE}(n^k)$
- $\mathrm{EXPSPACE} = \bigcup_{k\in\mathbb{N}} \mathrm{TIME}(2^{n^k})$

Exercice 12. *1. Show that $TIME(T(n)) \subset SPACE(T(n))$ (if a function is computable in time $T(n)$, then it is computable in space $T(n)$).*

2. What is the time and space complexity of the function $n \mapsto n+1$? (slightly tricky)

3. What is the time and space complexity of computing $f \vee g$ (logical or) given the complexity of computing f and g?

4. More difficult: show that $PSPACE \subset EXPTIME$. Hint: show that $SPACE(S(n)) \subset TIME(C^{S(n)})$ for some constant C.

The next result is given without proof, but it is much more technical than its appearance suggests. There is a good Wikipedia page on the topic.

Theorem 17 (Hierarchy theorem)**.** *Let t_1 and t_2 be two functions such that $t_1(n) = o(t_2(n))$. Under some technical hypotheses, we have $TIME(t_1(n)) \subsetneq TIME(t_2(n))$ (the inclusion is obvious, the inequality not at all). We have a similar result for space.*

This theorem is wrong without the technical hypotheses and there are some highly aritifical cases where two functions are different but describe the same complexity classes.

6.3 Nondeterministic models of computation

Nondeterminism is the possibility of a model having different behaviours in a given configuration. This does not seem to be a natural notion if you are writing actual programs; however we will see that these classes can be interpreted as functions for which it is possible to check efficiently whether a given value is the correct output. It is also of considerable interest for theoreticians, as you surely know.

Let us start with the straightforward definition, where we extend the model of Turing machines to allow for nondeterminism in the computation. The transition function now gives a set of possible behaviours (state, symbol on the cell, movement of the head). From a given starting configuration we now have a variety of possible computations, and we must find a good notion of what is the output in this context.

Definition 17. *A function $f : A^* \to \{0,1\}$ is **computable in nondeterministic time** $T(n)$ (denoted $f \in NTIME(T(n))$) if there is a nondeterministic Turing machine M such that:*

- *M **always** stops after $T(n)$ steps of computation at most;*
- *If $f(x) = 0$, then the machine **always** outputs 0 on input x.*
- *If $f(x) = 1$, then the machine **sometimes** outputs 1 on input x (it can output 0 the rest of the time).*

First note that this is an **asymmetrical** definition: 0 and 1 do not play the same role, exactly as in computable enumerations. We can therefore speak about coNTIME, coNSPACE, etc.

For a more intuitive grasp of nondeterminism, we can use the same pseudocode language to write programs, where in addition to the usual operations we are allowed to use a function `guess()` which can output 0 or 1 nondeterministically. Such nondeterministic programs are as powerful as nondeterministic Turing machines.

Exercice 13. *The set PRIME of prime numbers is in coNP.*

There is an alternative definition of nondeterministic classes that uses standard Turing machines; this makes some proofs easier, such as comparing nondeterministic and deterministic classes. Nondeterminism is replaced by a special input called **certificate** that tells the machine which choices it should make instead of guessing.

Theorem 18. *Let $f : A^* \to \{0,1\}$. $f \in NTIME(T(n))$ if, and only if, there is a Turing machine M such that, for any input x of length n,*

- *for any certificate $y \in A^{T(n)}$, $M(x,y)$ stops after $T(n)$ steps of computation at most;*
- *if $f(x) = 0$, then $M(x,y)\downarrow = 0$ for every certificate $y \in A^{T(n)}$;*
- *if $f(x) = 1$, then there exists a certificate $y \in A^{T(n)}$ such that $M(x,y)\downarrow = 1$.*

Soit $f : A^* \to \{0,1\}$. $f \in NTIME(T(n))$ si et seulement si il existe une machine de Turing M telle que, sur toute entrée x de taille n,

- *pour **tout** certificat $y \in A^{T(n)}$, $M(x,y)$ s'arrête après au plus $T(n)$ étapes de calcul;*
- *si $f(x) = 0$, alors $M(x,y)\downarrow = 0$ pour **tout** certificat $y \in A^{T(n)}$;*
- *si $f(x) = 1$, alors **il existe un** certificat $y \in A^{T(n)}$ tel que $M(x,y)\downarrow = 1$.*

Why do we limit certificates to length $T(n)$? We saw a similar idea earlier: since the program halts in time $T(n)$, it would not have the time to read a longer certificate anyway.

Here, the alternative interpretation is that a problem is in NP if a certificate that guarantees that the answer is "yes" can be checked in polynomial time.

Definition 18 (Nondeterministic complexity classes)**.**
- *$NTIME(T(n))$: functions that are computable in nondeterministic time $T(n)$.*
- *$NSPACE(S(n))$: functions that are computable in nondeterministic space $S(n)$.*

We similarly define classes $NL = NLOGSPACE$, $NP = NPTIME$, $NPSPACE$, $NEXPTIME\ldots$

Let us just mention that PSPACE = NPSPACE (Savitch's theorem), so nondeterministic space complexity classes are only studied for polynomial functions.

Let us do one formal proof to understand why the second definition can be more convenient.

Theorem 19.

$$NP \subset PSPACE$$

Proof. If $f \in$ NP, this means that there is a program P such that $f(x) = 0$ if, and only if, $P(x,y) = 0$ for all certificates y of length $T(|x|)$ for some polynomial T. Consider the following program:

```
program (input x){
        for all words y of length T(|x|):
                if P(x,y) = 1:
                        output 1
        output 0
}
```

Check that this is a deterministic program that computes f. The loop uses a space $T(|x|)$ to write the certificates in succession, and the program P uses polynomial time, so it also needs polynomial space. Therefore the whole program uses polynomial space. □

Theorem 20 (Overview)**.**

$$L = LOGSPACE \subset NL \subset P = PTIME \subset NP \subset PSPACE \subset EXPTIME \dots$$

We have $P \subsetneq PSPACE$ (the classes are not equal). However, we do not know whether $P = NP$, if $NP = PSPACE$, or if NP is somewhere in the middle.

The next result is given with no explanation; take it as an anecdote. For a complexity class C, denote C^f the same class where all Turing machines in the definition receive the function f as an oracle.

Theorem 21. *There are two functions A, B such that*

$$P^A = NP^A \qquad \text{and} \qquad P^B \neq NP^B$$

This result immediately breaks a lot of purported proofs of P $\stackrel{?}{=}$ NP: if the proof keeps working after we add an oracle everywhere, it must be wrong.

6.4 Randomised complexity

A randomised program is a program written using the usual operations and an additional operation `random()`, which outputs 0 or 1 with probability 1/2. Programs that use randomness are common, so it makes sense to develop a framework to talk about their complexity.

To formalise the notion, the situation is very similar to nondeterminism:

- we can define a new model, probabilistic Turing machines (that have access to an additional `random` operation), or
- we find an alternative definition that only uses standard Turing machines.

Randomised complexity classes offer more variety than nondeterminism; BPP is just an example of such a class.

Definition 19. *A function f is **computable in time** $T(n)$ **with bounded probability** (write $f \in BPTIME(T(n))$) if there is a probabilistic Turing machine such that, on any input x of length n,*

- *The machine always halts in time $T(n)$*
- *$M(x)\!\downarrow = f(x)$ with probability at least $\frac{2}{3}$.*

Just as in the previous section, there is an alternative definition using certificates.

Theorem 22. *A function f is computable in time $T(n)$ with bounded probability if and only if there is a Turing machine such that, for any input x of length n,*

- *$M(x, y)$ halts in time $T(n)$ for all certificates y;*
- *$M(x, y)\downarrow = f(x)$ for at least $\frac{2}{3}$ of all possible certificates.*

As earlier, the certificates y have length $T(n)$. If the certificate is chosen at random, the machine always halts and you will get the right answer at least 2/3 of the time.

Exercice 14. *1. What if we replaced $\frac{2}{3}$ by another value, such as $\frac{1}{2}$ or $\frac{3}{4}$? (consider only functions $A^* \to \{0, 1\}$)*

2. Show that $P \subset BPP$ and that $BPP \subset PSPACE$.

The open question $\mathrm{P} \stackrel{?}{=} \mathrm{BPP}$ is known as the derandomisation conjecture and is one of the big open complexity questions; it is linked to questions about strong deterministic pseudorandom generators.

This small excursion in randomised complexity was very limited in scope. There are many other classes that use randomness:

- The answer is always correct but the time is bounded on average (ZPP).
- Asymmetric classes: the answer may be wrong on one side but not on the other (RP).
- Classes with a limited number of `random` calls, etc.

Complexity theory has a lot of different subdomains with specific problems. If you arrive in a new domain, the tools you know may no longer work, but the most important thing is that you understand why the definitions make sense.

7 Reductions and completeness in complexity

7.1 Reductions again

Remember that reductions are used to compare the computational complexity of functions.

We saw reductions used in computability in sections 3.3 and 4.2 (for general and asymmetric computability, respectively). These reductions are not suited to complexity classes: remember that two computable functions are always Turing-equivalent. To compare functions inside some class C, we need an appropriate reduction.

Definition 20. *Let C be a complexity class and $\leqslant$ be a reduction.*
A function f is C-hard for $\leqslant$ if $g \leqslant f$ for every $g \in C$.
f is C-complete if, in addition, $f \in C$.

C-complete functions are "the hardest functions in C".
We will see in a moment how to choose $\leqslant$ so that these notions are useful.

Exercice 15. *1. If C is the set of computable functions, what are the C-complete functions for $\leqslant_T$? The C-hard functions?*

2. If f is C-complete and $f \leqslant g$, what does this mean for g? What if $g \leqslant f$?

*3. A reduction $\leqslant_1$ is **finer** than a reduction $\leqslant_2$ if $f \leqslant_2 g \Rightarrow f \leqslant_1 g$. What relationship do you have between the sets of C-complete functions for $\leqslant_1$ and $\leqslant_2$?*

Let us start with a bit of completeness in computability:

Theorem 23. *The halting problem is c.e.-complete for $\leqslant_m$.*

Proof. For any c.e. function $f : A^* \to \{0,1\}$, there is a program $prog_f$ that, on input x, halts if and only if $f(x) = 1$ (Theorem 11). You can use this program to write the reduction to the halting problem. □

Notice that a computable function cannot be c.e.-complete.

7.2 Time– or space–bounded reductions

Since a reduction is a program with access to an oracle, we can define time and space bounds by considering that calling the oracles takes 1 step of computation and uses no additional memory.

Definition 21. *If f and g are two functions, g Turing-reduces to f in time $T(n)$ if there is a program receiving g as an oracle that, on any input x of length n, halts and outputs $f(x)$ in time $T(n)$, and similarly for space.*

We define many-one reductions similarly.
How can we choose a suitable reduction for each class ? There are two main aspects:

1. the reduction has to allow less power (less resources) than functions from the class itself.
2. the reduction has to be strong (many-one) if we are dealing with an asymmetrical class.

First point. If the reduction has more power than the class, then we can have $f \leqslant g$ with $g \in C$ but $f \notin C$, which goes against the intuition of "more difficult" . If the reduction allows exactly as much power, then the whole class will be C-complete: for example, you can check that all functions in P are equivalent for Turing reduction in polynomial time[6]. However this reduction is a reasonable choice for PSPACE.

Second point. This is why many-one reductions exist. Since Turing reductions allow us to exchange true and false freely, they cannot distinguish e.g. NP and co-NP. If this is not clear, refer back to Section 11.

[6]Hint: the reduction can do the computation without using the oracle.

Some examples

- P-completeness is usually relative to LOGSPACE reductions[7].
- NP-completeness is defined by strong (many-one) reductions in polynomial time, also called **Karp reductions**.
- PSPACE-completeness and EXPTIME-completeness are defined by Turing reductions in polynomial time.

Outside of standard cases most papers will explicitely mention which reduction is being used.

7.3 Natural complete problems

Up to now it is not clear if complete problems should even exist. Here we show a general technique to build a first complete problem: predicting the behaviour of a Turing machine corresponding to the definition of the class.

Here is the first P-complete prediction problem:

$$pred_P : \begin{array}{rcl} \mathcal{M} \times \{0,1\}^* \times \mathbb{N}[X] & \to & \{0,1\} \\ (M,x,p) & \to & \begin{cases} 1 \text{ if } M(x)\downarrow = 1 \\ \quad \text{in at most } p(|x|) \text{ steps of computation} \\ 0 \text{ otherwise.} \end{cases} \end{array}$$

Halting and prediction problems are equivalent but the latter make proofs slightly easier.

Theorem 24. *$pred_P$ is P-complete for LOGSPACE reductions.$pred_P$ est P-complet pour les réductions LOGSPACE.*

Proof. For clarity, take a function $f : A^* \to \{0,1\} \in \mathrm{P}$. By definition of P, there is a program M and a polynomial T such that $M(x)\downarrow = f(x)$ in time $T(n)$ for all inputs x, where n is the length of x.

To compute f with access to an oracle $pred_P$, just call $pred_P(M,x,t)$. Remember that the space used for calling the oracle (in particular, the length of x) is not counted in space complexity. □

If you go back to your notes on NP-completeness, you can check that Cook's theorem – which proves that SAT is NP-complete – is actually a reduction to the prediction problem in nondeterministic Turing machines:

$$\begin{array}{rcl} \mathcal{M} \times \{0,1\}^* \times \mathbb{N}[X] & \to & \{0,1\} \\ (M,x,p) & \to & \begin{cases} 1 \text{ if there is a certificate } y \text{ such that } M(x,y)\downarrow = 1 \\ \quad \text{in at most } p(|x|) \text{ steps} \\ 0 \text{ otherwise.} \end{cases} \end{array}$$

[7]There are other possible reductions, such as NC reductions. Whether they give a different set of P-complete functions is an open problem.

8 Computability on real numbers

This part has two objectives: learning how to do computability on uncountable spaces, and understanding how computability corresponds to classical mathematical properties.

We will work on real numbers between 0 and 1 because it makes the definitions cleaner.

8.1 Individual real numbers

We saw in Section 2.1 that our definition of computability requires us to represent objects as functions with countable inputs and outputs (objects with a finite description).

Note. *The set of such functions is not countable; it has the so-called cardinality of the continuum. The reason why real numbers can be represented as functions with countable intput–output, and therefore why we can do computability on them, is because they also have the cardinality of the continuum.*

In fact, real numbers have two main representations. We can represent $x \in \mathbb{R}$:

1. by a binary (or decimal) representation:

$$bin_x : \begin{array}{ccc} \mathbb{N} & \to & \{0,1\} \\ n & \mapsto & n-\text{th bit of } x \end{array}$$

2. by a rational approximation:

$$approx_x : \begin{array}{ccc} \mathbb{N} & \to & \mathbb{Q} \\ n & \mapsto & q \text{ such that } |q-x| \leqslant \frac{1}{2^n} \end{array}$$

In the second case, the choice of the approximation gap $\frac{1}{2^n}$ is not significant. It could be, say, $\frac{1}{n}$ instead.

Note that every real number can be approximated by rationals, and that every rational number is computable. What matters is that a single program computes the whole sequence of rational numbers. Some authors say that the sequence is **uniformly computable** to insist that there is only one program computing the whole sequence.

Theorem 25. *Both representations are equivalent. That is,*

$$\forall x \in \mathbb{R}, bin_x \equiv_T approx_x.$$

Proof. We dinstinguish the cases where x is rational or irrational.

– $x \in \mathbb{Q}$. In this case, bin_x and $approx_x$ are computable. The binary representation of x is ultimately periodic, so we can write a program that distinguishes every case and output the correct digit. For example, if $x = 0,11010010010010\ldots$, bin_x would be:

$$1,2 \mapsto 1 \quad \text{and otherwise,} \quad n \mapsto 1 \text{ if } n \equiv 1 \bmod 3, 0 \text{ otherwise,}$$

and $approx_x$ is just the constant function that outputs x.

– $x \notin \mathbb{Q}$.

1. $bin_x \geqslant_T approx_x$: if we have oracle access to bin_x, we can compute a rational approximation of x by taking the first n bits of its binary representation. Formally,

```
program approx_x (input n) oracle bin_x{
        out = 0
        for i from 1 to n:
                out = 2*out + bin_x(i)
        output out/2^n
}
```

2. $bin_x \leqslant_T approx_x$: this is the subtle direction. First notice that if you move along the real line, the n-th bit changes at 2^{-n} intervals (more precisely, the interval limits are $k2^{-n}$ with $k \in \mathbb{Z}$).
   since $x \notin \mathbb{Q}$, x is not the limit of any interval, so for a large enough k, $x - 2^{-k}$ and $x + 2^{-k}$ are in the same 2^{-n} interval. This means that we know the k-th bit of x if we have an approximation of x up to 2^{-n}.

```
program bin_x (input k) oracle approx_x{
        for n from k to infinity:
                a = approx(x, n)
        if a − 2^−k and a + 2^−k have the same n-th bit:
            output this bit
}
```

   This program may never halt if $x \in \mathbb{Q}$: when x is on an interval limit no approximation will be enough. □

8.2 Real-valued functions

In this section, we extend progressively our framework to define the computability of functions $\mathbb{R} \to \mathbb{R}$. $\mathbb{R}$ is not countable, and these functions cannot be represented as functions with countable inputs and outputs in general[8], so we need new definitions here.

First let us think about the strategies real computer programs use to handle real numbers:

– **Symbolic** approach: we fix a few symbols (π, $\log(n)$, $\cos(n)\dots$) and we only handle the real numbers that can be expressed in this way — a countable subset.

– **Approximation** approach: we work with rational numbers (e.g. fixed-precision numbers of the form $\frac{q}{2^n}$) and input and output are approximations.

The second approach will provide more robust definitions (the first one is "just" string rewriting). While we work on the definitions, we will make sure that they correspond to the practical meaning of computability, and we will find many mathematical notions that we already know.

Let us begin with a simple case.

[8] This is because this set is larger than the continuum

Definition 22. *A function $f : \mathbb{Q} \to \mathbb{R}$ is computable if there exists a program M that, given x as input, can provide an approximation of $f(x)$ at any required precision:*

$$\forall x \in \mathbb{Q}, \forall n \in \mathbb{N}, \quad |M(x,n)\downarrow - f(x)| \leqslant \frac{1}{2^n}$$

Exercice 16. *If $f : \mathbb{Q} \to \mathbb{R}$ is computable, show that $f(q)$ is computable for any $q \in \mathbb{Q}$.*

To work on functions $\mathbb{R} \to \mathbb{R}$, the input as well as the output need to be given as approximations. However, think about the function $1_{\mathbb{Q}}$ ($1_{\mathbb{Q}}(x) = 1$ if $x \in \mathbb{Q}$ and 0 otherwise). What would computing that function even mean?

If we want to approximate $f(x)$ but we only have an approximation of x (say, we have x_n such that $|x - x_n| \leqslant \frac{1}{2^n}$), we need to make sure that $f(x_n)$ is not too far from $f(x)$ – otherwise no computation can make sense.

We will have to assume that f is **continuous** in quite a strong sense:

If we want to compute an approximation of $f(x)$ with precision $\frac{1}{2^n}$,
we need an approximation of x with precision $\frac{1}{2^{g(n)}}$ where g is computable.

g is called the **modulus of continuity** in mathematics. In the end, we see that there are two error sources to handle:

- our input is an approximation of the real input.
- our program can only output an approximation of the correct result even on a rational input.

We arrive at the following definition which should correspond to the intuitive definition of computing a real function in programming:

Definition 23. *A function $f : \mathbb{R} \to \mathbb{R}$ is computable if there are two programs C (computation) and P (precision) such that:*

$$\forall x \in \mathbb{R}, \forall n \in \mathbb{N}, |x_n - x| \leqslant \frac{1}{2^{P(n)\downarrow}} \Rightarrow |C(x_n,n)\downarrow - f(x)| \leqslant \frac{1}{2^n}$$

Theorem 26. *If f is computable, then f is continuous.*

How do we use this definition to compute $f(x)$ with precision $\frac{1}{2^n}$ in practice?

1. Compute $P(n+1)$.
2. Provide a rational x_n that approximates x with precision $\frac{1}{2^{P(n+1)}}$.
3. Output $C(x_n, n+1)$.

Exercice 17.
1. *Show that if $f : \mathbb{R} \to \mathbb{R}$ is computable and x is computable, you can write a program that, on input n, outputs an approximation of $f(x)$ with precision $\frac{1}{2^n}$.*
2. *Corollary of the above: show that if x is computable, then $f(x)$ is computable.*

The whole definition works with rational approximations. We could instead require the program to output the first n bits of $f(x)$. This would not work at all. Try to prove that $x \mapsto 3x$ is computable in decimal and you will have problems.

Some examples of uncomputable real functions:

- Any discontinuous function;
- The constant function $x \mapsto r$ where r is an uncomputable real.

9 Symbolic dynamics; computability in other topics

9.1 Tiling spaces and domino problems

Up to now, undecidable or complete problems were all about programs. This is Turing FM, where computabilists talk to computabilists.

You probably already know many natural NP-complete problems; most classes also have a variety of natural problems coming from different mathematical topics. Let us meet a few of them that come from **tilings**.

Definition 24. *Given:*

- *An alphabet, that is, a finite set of symbols A.*
- *A finite set $\mathcal{F}$ of **forbidden patterns**, which are objects from $U \to \mathcal{A}$ for some finite $U \subset \mathbb{Z}^2$.*

*A colouring of $\mathbb{Z}^2$ ou **pavage** – that is, a function $\mathbb{Z}^2 \to \mathcal{A}$ – is **admissible** (for $\mathcal{F}$) if no forbidden pattern appears in it. The associated **tiling space** is the set of its admissible tilings.Étant donnés :*

- *Un alphabet A, autrement dit, un ensemble fini de symboles.*
- *Un ensemble fini $\mathcal{F}$ de **motifs interdits**, qui sont des objets de $U \to \mathcal{A}$ pour un ensemble fini $U \subset \mathbb{Z}^2$.*

*Un coloriage de $\mathbb{Z}^2$ ou **pavage** – c'est-à-dire, une fonction $\mathbb{Z}^2 \to \mathcal{A}$ – est **admissible** (for $\mathcal{F}$) si aucun motif interdit n'y apparait. L'**espace de pavage** associé à $\mathcal{F}$ est l'ensemble de ses pavages admissibles.*

The fundamental problem of the domain is the **domino problem**: given a tiling space (given A and $\mathcal{F}$), does an admissible tiling exists?

This problem is actually undecidable. Here we will make proofs for a simpler variant so that they remain accessible.

Definition 25. *The **seeded domino problem**:*

Input *An alphabet, a set of forbidden patterns, and an integer n in unary,*

Output *The answer to one of the following questions (depending on the variant)*

- *$Seed(\infty, \infty)$: can the seed be extended to an admissible tiling of $\mathbb{Z}^2$?*
- *$Seed(n, n)$: can the seed be extended to an admissible tiling of the square $[0, n]^2$?*
- *$Seed(n, \infty)$: can the seed be extended to a tiling of the band $[0, n] \times \mathbb{Z}$?*

These problems appear pretty easy at first glance, but some difficult combinatorics emerge when the number of symbols increases, as we will prove. What kind of programs can we write?

- *Seed*(n, n): it is not hard to write a brute force program in polynomial space that tests all possibilities. By guessing the tiling and then checking it in polynomial time, we can see that the problem is actually in NP.
- *Seed*(n, ∞): for clarity, assume that the forbidden patterns are only adjacency constraints (two symbols). Try to tile the rectangle $n \times |\mathcal{A}|^n$. If you cannot find an admissible tiling, then the answer must be false. If you find such a tiling, then one line has to appear twice (pigeonhole principle) and we can tile the band by reapeating the periodic part, so the answer is true.

  By guessing the lines of the tiling in succession we can turn this idea into a NPSPACE = PSPACE program.
- *Seed*(∞, ∞): we can try to find tilings on squares $[0, n]^2$ by brute force as before. If some square does not have an admissible tiling, the answer is certainly false; if we find a periodic admissible tiling, the answer is certainly true; but it is not certain that we will always find such a tiling. We can say that it is co-computably enumerable, because we will find a square that cannot be tiled if the answer is "false".

9.2 Simulating universal computation

We can build an alphabet, a set of forbidden patterns and a seed such that any admissible tiling represents the space–time diagram of some computation of a Turing machine (that can be nondeterministic if needed):

(todo pictures)

Thus, any admissible tiling that extends the seed must contain the computation of a Turing machine.

This simulation leads to the following completeness results:

Theorem 27. *1. Seed*(∞, ∞) *is co-c.e.-complete.*

2. Seed(n, n) *is NP-complete.*

3. Seed(n, ∞) *is PSPACE-complete.*

Proof. 1. We make a reduction to the halting problem by adding the tile representing the halting state of the Turing machine to the set of forbidden patterns. There is an admissible tiling if and only if the machine never stops.

2. We make a reduction to the prediction problem by adding to the forbidden patterns the tile representing the halting state with the tape value 0 (false). There is an admissible tiling if and only if there is a computation that answers 1 in time n.

3. Same reduction as above. □

1. On fait une réduction au problème de l'arrêt en ajoutant la tuile représentant l'état d'arrêt de la machine de Turing à l'ensemble des motifs interdits. Il y a un pavage admissible si, et seulement si, la machine ne s'arrête pas.

2. On fait une réduction au problème de prédiction en ajoutant aux motifs interdits la tuile représentant l'état d'arrêt de la machine de Turing si la réponse est 0 (faux).

Il y a un pavage admissible si, et seulement si, il existe un calcul qui répond 1 après au plus n étapes de calcul.

3. Même réduction que ci-dessus.

I will mention with no proof that $Seed(1, n)$ is NL-complete; you can look for information on the reachability problem in graphs.

9.3 Meaning of the results

In this example, as is often the case, completeness results in complexity and computability go together depending on whether we consider the finite or infinite versions of problems, and they come from some simulation of universal computation in an apparently nice combinatorial setting.

First remember that we are working on worst-case complexity / computability. Those problems might be solvable in 99.9% of the cases by heuristics — this is a big problem in cryptography, for example. It is possible to provide better guarantees, but this is yet another domain of computability.

Still, these results tell us that we will not find nice, general methods to solve the problem (or to solve it much faster): these are impossibility statements or lower bounds for computational, but also mathematical complexity.

Intellectually, impossibility results should have the same value as positive results, but you might find them a bit arid[9]. A more positive point of view is that a nice general method "kills" the problem — there is nothing more to do — whereas impossibility results give a good justification for considering easier variants, approximations, additional hypotheses, etc., and in a way give more value to the partial results.

If you find yourself nostalgic for the Hic Svnt Dracones of ancient maps, now that there is nothing more to explore, uncomputability results are your dragons — and they are here, provably, forever.

[9] if you are looking for funding...