

The Art of Computer Programming 3

Lecture Notes

Lecture 1.	Computable and uncomputable functions	2
Lecture 2.	Decidable and enumerable languages	6
Lecture 3.	Turing machines	12
Lecture 4.	A brief history of computer programming	—
Lecture 5.	Lambda calculus	20
Lecture 6.	More lambda calculus	26
Lecture 7.	Time complexity, classes P and EXP	30
Lecture 8.	Class NP	34
Lectures 9 & 10.	Reductions and NP-completeness	36
Lecture 11.	Space complexity	44

Lecture 1. Computable and uncomputable functions

1.1 Alphabets and words

At the low level, information inside computers is represented by 0s and 1s. Every program takes as input a finite sequence of bits and returns another such sequence, and all computation as we know it, in the end, is manipulation of bit-strings. Our formalisation begins here.

Definition 1 (Alphabets and words). An **alphabet** Σ is a non-empty finite set of symbols. A **word** (or **string**) over Σ is a finite sequence of symbols $w = a_1 a_2 \dots a_m$, where each $a_i \in \Sigma$. The unique sequence of length zero is the **empty word**, denoted ϵ . The set of all finite words over Σ is denoted Σ^* .

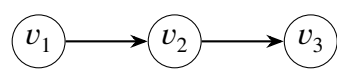
Given two words $u = a_1 \dots a_n$ and $v = b_1 \dots b_m$, their **concatenation** is $uv = a_1 \dots a_n b_1 \dots b_m$. Concatenation is associative: $(uv)w = u(vw)$, and ϵ is the two-sided identity: $u\epsilon = \epsilon u = u$.¹

Binary encoding. The ‘canonical’ alphabet for computers is the *binary alphabet* $\{0, 1\}$. Words in any other alphabet Σ can be *encoded* by binary words as follows. We fix a length k such that $2^k \geq |\Sigma|$, and assign a distinct k -bit binary word to each symbol a_i . For example, if $\Sigma = \{a, b, c\}$, we can choose $k = 2$ and map

$$a \mapsto 00, \quad b \mapsto 01, \quad c \mapsto 10.$$

A word over Σ is then encoded by replacing each symbol with its code: for instance, cab becomes 100001. Since every symbol uses exactly 2 bits (or, in general, k), decoding is unambiguous: we simply read the string two bits at a time.²

Other finite objects that programs work with are encoded similarly. For example, a directed graph G on vertices $\{v_1, \dots, v_n\}$ can be represented by its $n \times n$ adjacency matrix A , where $A_{i,j} = 1$ if there is an edge $v_i \rightarrow v_j$ and 0 otherwise. Flattening this matrix row by row yields a binary word of length n^2 . For instance, the graph on 3 vertices with edges $v_1 \rightarrow v_2$ and $v_2 \rightarrow v_3$ and its corresponding adjacency matrix are:


 $A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix},$

which is encoded by the word 010001000.

1.2 Computable functions

Fix your favourite general-purpose programming language (we will use Python in these notes). Consider programs that take one input word and produce one output word over some alphabet Σ . As explained above, multiple arguments, numbers, graphs, or structured data can always be encoded into a single word in Σ^* (and decoded from the output).

Thus, each program π naturally computes a (partial!) function $f_\pi : \Sigma^* \rightarrow \Sigma^*$.³ This is how it is defined.

¹If you like algebra, you may note that Σ^* with concatenation and ϵ is thus the free monoid generated by Σ .

²In ASCII, a standard encoding in real computers, each character is an 8-bit byte.

³Note the arrow! It is $\rightarrow$, not $\mapsto$. This is how we denote that the function is *partial*.

Definition 2 (Function computed by a program π). Given $u \in \Sigma^*$, run π on u .

- If π halts (i.e., terminates, concludes the execution), then $f_\pi(u) = v$, where $v \in \Sigma^*$ is the word it outputs.
- If π does not halt, then $f_\pi(u)$ is undefined.

In real life, programs may also crash with an error. In theory, we treat crashes as equivalent to non-halting.

Definition 3 (Computable function). A partial function $f : \Sigma^* \rightarrow \Sigma^*$ is called **computable** if there exists a program π such that $f = f_\pi$.

Note that when $f : \Sigma^* \rightarrow \Sigma^*$ is **total**, i.e., defined on all words in Σ^* , the corresponding program π must halt on every input.

Let us consider some examples of computable functions.

Example 4. The following functions are all computable:

- *reverse*: given $w \in \Sigma^*$, outputs its reverse (e.g., $\text{reverse}(abc) = cba$);
- *convert*: interprets $w \in \{0, 1, \dots, 9\}^*$ as an integer n , outputs a word in $\{0, 1, 2, 3\}$ —the same number, but in 4-ary notation;
- *double*: interprets w as an integer n , outputs $2n$;
- *multiply*: given a pair of integers $\langle n, k \rangle$, outputs their product $n \cdot k$;
- *shortestPath*: given a directed graph and two vertices $\langle G, s, t \rangle$, outputs the length of the shortest path from s to t (undefined if no path exists);
- *deanon*: given w , if w encodes the real name of Banksy, $\text{deanon}(w)$ encodes the real name of Satoshi Nakamoto; for all other w , *deanon* is undefined.

1.3 Uncomputable functions

Are there any functions that are not computable? Yes, and it is easy to prove: there are only countably many programs, but uncountably many functions on $\{0, 1\}^*$. Let us see this argument in detail.

Recall that a set S is called *countable* if its elements can be indexed by natural numbers. The set $\{0, 1\}^*$

is, of course, countable—here is a possible indexing of words in $\{0, 1\}^*$:

$$\begin{aligned}
 w_0 &= \varepsilon \\
 w_1 &= 0 \\
 w_2 &= 1 \\
 w_3 &= 00 \\
 w_4 &= 01 \\
 w_5 &= 10 \\
 w_6 &= 11 \\
 w_7 &= 000 \\
 &\dots
 \end{aligned} \tag{1}$$

That is, we list the words by increasing length, and order words of the same length lexicographically. Similarly, Σ^* is countable for any alphabet Σ .

Programs are words themselves. Indeed, programs are just sequences of symbols that we write, and thus can be seen as words. In fact, the converse is also true: we can try to execute any word as if it was a program and see what happens—a crash would just mean non-halting. So every program is a word and every word encodes some (maybe nonsensical) program. The set of all programs, and hence of all computable functions, is therefore countable.

In contrast, the set of all partial functions $\{0, 1\}^* \rightarrow \{0, 1\}^*$ is uncountable. In fact, this is already true for the set of all functions $\{0, 1\}^* \rightarrow \{0, 1\}$.

Proposition 5 (Cantor, 1891). *The set of all functions $\{0, 1\}^* \rightarrow \{0, 1\}$ is uncountable.*

Proof. For clarity of notation, we use the correspondence (1) and consider the set of functions $\mathbb{N} \rightarrow \{0, 1\}$. Assume for contradiction that this set is countable. Then we can index all functions in this set: $f_0, f_1, f_2, \dots$. Each function f_i is defined by its values on each $k \in \mathbb{N}$. Let us write them out:

$f_0(\mathbf{0})$	$f_0(1)$	$f_0(2)$	$f_0(3)$	...
$f_1(0)$	$\mathbf{f_1(1)}$	$f_1(2)$	$f_1(3)$	...
$f_2(0)$	$f_2(1)$	$\mathbf{f_2(2)}$	$f_2(3)$	...
$f_3(0)$	$f_3(1)$	$f_3(2)$	$\mathbf{f_3(3)}$	...
...				

Consider the diagonal of this infinite table, in bold, and the function:

$$d(n) = \begin{cases} 1, & \text{if } f_n(n) = 0, \\ 0, & \text{if } f_n(n) = 1. \end{cases}$$

Then, d is a function $\mathbb{N} \rightarrow \{0, 1\}$, so it must appear in the above list $f_0, f_1, f_2, \dots$ of all such functions. However, by construction, d disagrees with f_0 on 0, with f_1 on 1, and, in general, with f_n on n . A contradiction. $\square$

Corollary 6. *There exist uncomputable functions from $\{0, 1\}^*$ to $\{0, 1\}$.*

In fact, there are uncountably many.

And so what? No concrete example was given so far. Could it be that all *useful* functions that we may ever want to compute are countable? Unfortunately, not. Consider the following very useful *halting function*:

$$h(\langle \pi, w \rangle) = \begin{cases} 1, & \text{if } \pi \text{ halts on input } w, \\ 0, & \text{if } \pi \text{ loops forever on input } w. \end{cases}$$

Its argument is a *pair*: a program π and a word w , put in the $\langle \rangle$ -brackets and separated by a comma—this is just a way to ‘pack’ two different things into one argument of a function. We assume that $\langle, \rangle$ and the comma are all symbols of Σ that are not used in programs and input words.⁴ Note that h is a total function.

Theorem 7 (Turing, 1936). *The halting predicate is (alas!) uncomputable.*

Proof. Recall that we have an indexing of words in Σ^* : $w_0, w_1, w_2, \dots$, each of which can be seen as a program: $\pi_0, \pi_1, \pi_2, \dots$. Each π_i can (potentially) be run on each w_j (if $i = j$, we run π_i on its own code!)

We can imagine a table of results for the halting predicate:

	w_0	w_1	w_2	w_3	$\dots$
π_0	$h(\pi_0, w_0)$	$h(\pi_0, w_1)$	$h(\pi_0, w_2)$	$h(\pi_0, w_3)$	$\dots$
π_1	$h(\pi_1, w_0)$	$h(\pi_1, w_1)$	$h(\pi_1, w_2)$	$h(\pi_1, w_3)$	$\dots$
π_2	$h(\pi_2, w_0)$	$h(\pi_2, w_1)$	$h(\pi_2, w_2)$	$h(\pi_2, w_3)$	$\dots$
π_3	$h(\pi_3, w_0)$	$h(\pi_3, w_1)$	$h(\pi_3, w_2)$	$h(\pi_3, w_3)$	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

We are going to mess with the diagonal again! Suppose for contradiction that h is computed by some program η . We define the following *diagonal* program δ :

```
def  $\delta(w)$ :
    if  $\eta(w, w) == 1$ :
        while True: pass      # loop forever
    else:
        return 0              # halt immediately
```

Since δ is a valid program, we could ask what happens when δ is run on its own code. If δ halts when run on the word encoding δ , then $\eta(\delta, \delta)$ must return 1, and so δ , according to its own code, should actually loop forever. If it does not halt, $\eta(\delta, \delta)$ returns 0, and then δ should halt. A contradiction. $\square$

⁴This creates a problem: we need to define h also on ‘malformed’ inputs—such as $\langle \langle, , \rangle$ —which is formally a word from Σ^* . But we can agree that h will be equal to 0 on such words and forget about them. What we really care about is the behaviour of h on properly formed words: a $\langle$, then a program, then a comma, then an input word, then a $\rangle$.

Lecture 2. Decidable and enumerable languages

2.1 From functions to languages

Let $f : \Sigma^* \rightarrow \{0, 1\}$ be a total function that outputs a single bit. It can be thought of as classifying input words into two categories: those on which f returns 1 (the ‘yes’ words) and those on which f returns 0 (the ‘no’ words).

This is the same as specifying a subset $L \subseteq \Sigma^*$ —namely, the set $L = \{w \in \Sigma^* \mid f(w) = 1\}$.

Conversely, every set $L \subseteq \Sigma^*$ defines a one-bit function, called its **characteristic function**:

$$\chi_L(w) = \begin{cases} 1, & \text{if } w \in L, \\ 0, & \text{if } w \notin L. \end{cases}$$

Every set $L \subseteq \Sigma^*$ is associated with the following problem of binary classification, called the **decision problem**⁵ for L :

Input: $w \in \Sigma^*$

Question: Does w belong to L ?

Definition 8. A set of words $L \subseteq \Sigma^*$ is called a **language**. It is **decidable** when its characteristic function is computable, and **undecidable** otherwise.

A program that computes the characteristic function χ_L is said to **decide** the language L : it outputs 1 on words from L (accepts them) and 0 on words not from L (rejects them). Note that such a program must halt on every input—and give an answer.

We thus have a correspondence between one-bit functions and languages:

	Function $f : \Sigma^* \rightarrow \{0, 1\}$	Language $L \subseteq \Sigma^*$
Task (‘problem’)	given w , compute $f(w)$	given w , decide whether $w \in L$
If some π does that	computable	decidable
This π	computes the function	decides the language

Example 9. The empty language $L = \emptyset$ is decidable. It is decided by the following program:

```
def reject_all(w):
    return 0
```

Example 10. Any finite language $L = \{w_0, w_1, \dots, w_n\}$ is decidable:

⁵The term is historical: basically, we are asked to ‘decide’ if the input word belongs to the set.

```
def decide_L(w):
    words_L = [w0, w1, ... , wn]
    if w in words_L:
        return 1
    else:
        return 0
```

Example 11. The language

$$\text{PRIMES} = \{w \in \{0, 1\}^* \mid w \text{ is the binary encoding of a prime number}\}$$

is decidable:

```
def is_prime(w):
    n = int(w, 2)          # interpret binary string as integer
    if n < 2:
        return 0
    for d in range(2, n):
        if n % d == 0:
            return 0
    return 1
```

Example 12. The language

$$\text{HALT} = \{\langle \pi, w \rangle \mid \pi \text{ halts on input } w\}$$

is undecidable, by Theorem 7.

It is a tradition in the theory of computing to name languages in ALL CAPS. It might be slightly ANNOYING, but that's the tradition, and we will RESPECT IT.

2.2 Semantic properties of programs

We have seen that HALT is undecidable, which is sad for those who would like to *verify* that their programs behave in the expected manner—at least, halt on a given word. Are there properties of programs that one can effectively check?

Of course there are. For example, this one:

Input: program code

Question: Does this program use a variable named x ?

It should be clear that the property above is qualitatively different from HALT. The former asks about the program's *syntax*, the latter—about the *function* that the program computes, i.e. the program's *semantics*. To highlight this distinction, we introduce some terminology.

Recall that every $w \in \Sigma^*$ can be treated as a program: feed it to Python's interpreter, specify an input, and it will do something—maybe crash immediately, maybe output Banksy's real name, or maybe output the input backwards...—it will compute some partial function $\Sigma^* \rightarrow \Sigma^*$.

Any language $P \subseteq \Sigma^*$ defines a **property of programs**: a program π has property P if $\pi \in P$, and does not have it if $\pi \notin P$.

Definition 13. A property of programs $P \subseteq \Sigma^*$ is called *semantic* if for any two programs $\pi_1, \pi_2 \in \Sigma^*$, if $f_{\pi_1} = f_{\pi_2}$, then either both $\pi_1, \pi_2 \in P$ or both $\pi_1, \pi_2 \notin P$.

In other words, a semantic property is a property of *functions*: for every function, either all programs that compute it have the property, or none does.

A property P is **non-trivial** if $P \neq \emptyset$ and $P \neq \Sigma^*$. That is, there are some programs that have it, and some that do not.

Example 14. Consider the property

$$\text{HALT-ON-EMPTY} = \{\pi \mid \pi \text{ halts on input } \varepsilon\}.$$

It is semantic: $\pi \in \text{HALT-ON-EMPTY}$ iff $f_\pi(\varepsilon)$ is defined. It is non-trivial: a program that always outputs 1 regardless of the input has this property, and one that loops forever on all inputs does not have it.

The following theorem is the saddest theorem in our course.

Theorem 15 (Rice, 1953). *Every non-trivial semantic property of programs is undecidable.*

Proof. The proof was not covered in the lecture, so it is here just for the curious.

Let $P \subseteq \Sigma^*$ be a non-trivial semantic property, and consider the following program:

```
def never_stop(w):
    while True: pass    # loops forever regardless of the input
```

The function that it computes, $f_{\text{never_stop}}$, is the (unique) function that is undefined on all inputs.

Suppose that $\text{never_stop} \notin P$. (If $\text{never_stop} \in P$, consider the complement $\overline{P} = \Sigma^* \setminus P$ instead, which is also a non-trivial semantic property. Properties P and $\overline{P}$ are either both decidable or both undecidable, so this is without loss of generality.)

Since P is non-empty, there is some program $\sigma \in P$. Since P is semantic and $\text{never_stop} \notin P$, we have that $f_\sigma \neq f_{\text{never_stop}}$, meaning that σ halts on at least one input.

We reduce the Halting Problem to P . Given a pair $\langle \pi, w \rangle$, construct the following program $\rho_{\pi, w}$:

```
def rho_pi_w(x):
    pi(w)                # run pi on w (ignore x)
    return sigma(x)      # if pi halted, behave like sigma
```

If π halts on w , then $\rho_{\pi,w}$ behaves exactly like σ , so $f_{\rho_{\pi,w}} = f_{\sigma}$ and $\rho_{\pi,w} \in P$. If π does not halt on w , then $\rho_{\pi,w}$ never reaches the second line, so $f_{\rho_{\pi,w}} = f_{\text{never_stop}}$ and $\rho_{\pi,w} \notin P$.

Therefore, $\langle \pi, w \rangle \in \text{HALT}$ if and only if $\rho_{\pi,w} \in P$. If P were decidable by some program `decide_P`, we would be able to decide HALT as follows:

```
def decide_halt( $\pi$ , w):
     $\rho_{\pi,w}$  = construct( $\pi$ , w)    # construct the code of  $\rho_{\pi,w}$  as above
    return decide_P( $\rho_{\pi,w}$ )      # check whether  $\rho_{\pi,w} \in P$ 
```

This contradicts Theorem 7. □

Example 16. The following languages are all undecidable by Rice’s theorem:

- $\{\pi \mid \pi \text{ halts on all inputs}\};$
- $\{\pi \mid \pi \text{ halts on some input}\};$
- $\{\pi \mid \pi \text{ accepts the empty word}\};$
- $\{\pi \mid f_{\pi} = f_{\pi_0}\}$ for any fixed π_0 — the set of programs equivalent to π_0 .

Remark 17. There is one hidden detail in the proof of Rice’s theorem—the *construction* of the program $\rho_{\pi,w}$ from π and w . For a proper reduction, we need this construction to be computable: given the code of π and the word w , we must be able to algorithmically produce the code of $\rho_{\pi,w}$. In Python (and any real programming language), this is obvious: $\rho_{\pi,w}$ is just a piece of source code assembled by string concatenation. Formally, a programming language (‘encoding’) is *admissible* if it has an interpreter and ensures the ability to algorithmically hard-code arguments into a given program’s source code. Rice’s theorem holds for any admissible encoding, but can fail in artificial, pathological programming languages where programs cannot be algorithmically constructed (imagine, for example, a language where *somehow* every program has an additional flag—whether it halts on the empty input or not).

2.3 Enumerable languages

Definition 18. A language $L \subseteq \Sigma^*$ is called **recursively enumerable** (or simply **enumerable**) if there exists a program that runs indefinitely and prints a (possibly infinite) list of words, such that:

- (1) every word printed by the program belongs to L ;
- (2) every word belonging to L is eventually printed by the program (possibly with repetitions).

Example 19. The set $\text{EVEN} = \{n \in \mathbb{N} \mid n = 2k, k \in \mathbb{N}\}$ is enumerable:

```
def enum_even():
    x = 0
    while True:
        print(x)
        x = x + 2
```

Example 20. The language $\{0, 1\}^*$ is enumerable:

```
def enum_bin_words():
    words = [""]      # create an array with just ε
    for w in words:
        print(w)
        words.append(w + "0")
        words.append(w + "1")
```

If we replace `print(w)` with `yield w`, `enum_bin_words()` becomes an infinite generator, allowing us to write `for w in enum_bin_words():` to iterate over all words in $\{0, 1\}^*$ just like a Python range.

By analogy, Σ^* is enumerable and can be iterated through, for any alphabet Σ .

Example 21. The language $\text{HALT} = \{\langle \pi, w \rangle \mid \pi \text{ halts on input } w\}$ is enumerable.

Since $\{0, 1\}^*$ is enumerable, we can enumerate (binary encodings of) all programs and all input words. But to enumerate the pairs $\langle \pi, w \rangle \in \text{HALT}$, we would need to know if π halts on w . How can we tell, if HALT is undecidable?

Imagine we had infinitely many processors, one for each pair $\langle \pi_i, w_j \rangle$. Then, for every $\langle \pi_i, w_j \rangle$ we can run a process p_{ij} , all of them in parallel, and wait. When a process terminates, we print the respective pair to the output. And those processes that never terminate will never have their pairs printed—that's exactly what we want.

The hard part is how to simulate infinitely many processes on a normal computer. We will rely on the following consideration. At the lowest level, a program does not execute all at once: its execution proceeds as a discrete sequence of elementary *machine instructions*—simple primitive steps: fetching a byte from memory, performing an arithmetic operation, or jumping to another location in code. Each instruction corresponds to a single unit of execution time.

So we can simulate the ‘parallelisation’ as follows. First, we arrange the infinite two-dimensional grid of processes p_{ij} into a single linear sequence: $p_0, p_1, p_2, \dots$ (for instance, ordering by the sum $i + j$). Then, we share the processor by gradually allocating more instructions to more processes:

- On step 0, run p_0 for 1 instruction.
- On step 1, run p_0 for a 2nd instruction, and p_1 for 1 instruction.
- On step 2, run p_0 for a 3rd instruction, p_1 for a 2nd instruction, and p_2 for 1 instruction.
- In general, on step s , run p_i for its $(s - i + 1)$ -th instruction for each $i \in \{0, 1, \dots, s\}$.

Our program `enum_halt` will maintain a list of running processes and for each store their current progress. It will iterate through this list, pushing each process forward by one step, and appending newly launched processes. When a process terminates, the respective pair $\langle \pi_i, w_j \rangle$ is printed out.

This technique has a vintage name, *dovetailing*. The metaphor comes from carpentry: a *dovetail joint* joins two boards by interlocking alternating fingers of wood. Here, we similarly interleave small slices of execution from different processes.

Theorem 22 (Post, 1944). *A language $L \subseteq \Sigma^*$ is decidable if and only if both L and its complement $\overline{L} = \Sigma^* \setminus L$ are enumerable.*

Proof. ($\Rightarrow$) Suppose L is decided by a program `decide_L`. Let `enum_words()` be a generator for Σ^* , analogous to the `enum_bin_words` from Example 20. We can enumerate L simply by filtering:

```
def enum_L():
    for w in enum_words():      # iterate over all words in  $\Sigma^*$ 
        if decide_L(w) == 1:
            print(w)
```

For the complement $\overline{L}$, we do the exact same thing, replacing `== 1` with `== 0`.

($\Leftarrow$) Let `enum_L` and `enum_not_L` enumerate L and $\overline{L}$, respectively. On input w , run `enum_L` and `enum_not_L` in parallel until one of them outputs w (on two processors, or on one, via dovetailing). If `enum_L` outputs w , return 1; if `enum_not_L` outputs w , return 0. Since w must belong to either L or $\overline{L}$ (and never both), exactly one will output w at some point, so this decides L . $\square$

Corollary 23. *The complement of the Halting Problem,*

$$\overline{\text{HALT}} = \{\langle \pi, w \rangle \mid \pi \text{ does not halt on } w\}$$

*is **not** enumerable.*

Lecture 3. Turing machines

3.1 The 1930s: What is computation?

Until now, we have used Python to write programs—our statement of Turing’s theorem is that no Python program computes the halting function (in the terminology of Lecture 1) or decides the language HALT (in the terminology of Lecture 2). However, Turing proved his theorem in 1936, before electronic programmable computers existed and when pythons were more dangerous to work with. This deserves an explanation.

The story begins in the 1920s with David Hilbert’s grand programme (not program!) to establish the foundations of mathematics. A central question of this programme was the *Entscheidungsproblem* (the ‘decision problem’), formulated by Hilbert and Wilhelm Ackermann in 1928:

Does there exist an effective procedure that takes as input any mathematical statement (encoded in a special notation called ‘first-order logic’) and decides whether the statement is true or false?

No electronic computers existed at the time, so it was supposed that such a ‘procedure’ would be executed by a person with pen and paper. For a procedure to be ‘effective’, it must be applicable mechanically, by following clear instructions step by step, without creative thinking or other usual attributes of a mathematician’s work.

Example 24. Consider solving a quadratic equation $ax^2 + bx + c = 0$, where $a \neq 0$, over the real numbers. Compute the discriminant $\Delta = b^2 - 4ac$. If $\Delta < 0$, there are no real solutions; otherwise, evaluate $x = \frac{-b \pm \sqrt{\Delta}}{2a}$. This is an effective procedure: the steps are explicit and require no mathematical insight or intuition.

Hilbert believed mathematics was decidable: that any well-posed mathematical question could be decided mechanically. To prove this belief rigorously—or to disprove it, as eventually happened—one needed a formal definition of an effective procedure, i.e. of an *algorithm*.

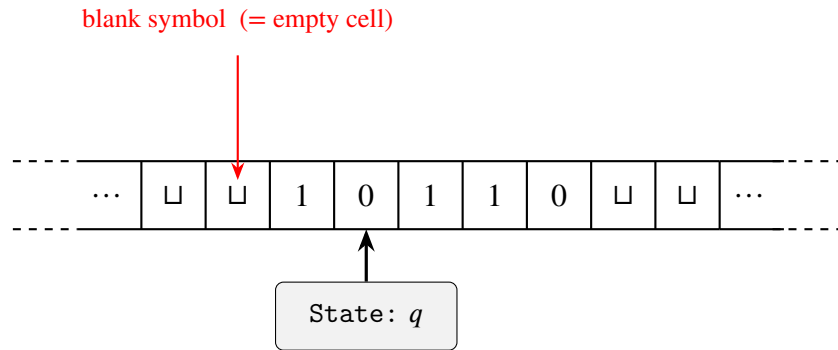
Several definitions of an algorithm were proposed in the 1930s. Although developed by different people, they all turned out to be equivalent in terms of which functions are computable. This is strong evidence that they capture the ‘nature of computation’. We will consider some of these definitions in our course.

Probably the most famous definition, and the first that we will study (although not the first historically), was given by Alan Turing in 1936.

3.2 The Turing machine

Informally speaking, a Turing machine is a device consisting of three components:

1. An **infinite tape** divided into discrete cells, extending indefinitely in both directions. Each cell can hold exactly one symbol from a finite **tape alphabet** Γ .
2. A **read/write head** that points to a single cell on the tape at any moment.
3. A **finite control** that stores the internal state q from a finite set of states Q .



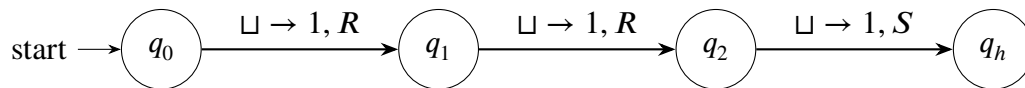
The machine works in steps. At each step, it reads the symbol written in the current cell, and then, depending on that symbol and on the state it is in:

1. overwrites the cell with a new symbol;
2. moves one cell to the right, or to the left, or stays put;
3. switches to another state (of the finitely many available).

A concrete machine is defined by specifying its behaviour for each combination of the observed symbol and current state, and a *halting state*, upon entering which the machine turns off.

Example 25. Consider a machine that, starting on an empty tape (where every cell contains $\square$), prints three 1s and stops.

We can achieve this using four states: q_0 (initial), q_1 , q_2 , and a halting state q_h . The machine starts in q_0 scanning a blank cell, overwrites it with 1, moves right, and switches to q_1 . In q_1 , it writes another 1, moves right, and switches to q_2 . Finally, in q_2 , it writes a third 1 and enters the halting state q_h :



Here, a label $\square \rightarrow 1, R$ means “if reading $\square$, write 1 and move right”, while S means “stay put”.

Formally, we define a Turing machine as follows.

Definition 26. A *Turing machine* (TM) is a tuple of six components:

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_h \rangle$$

where:

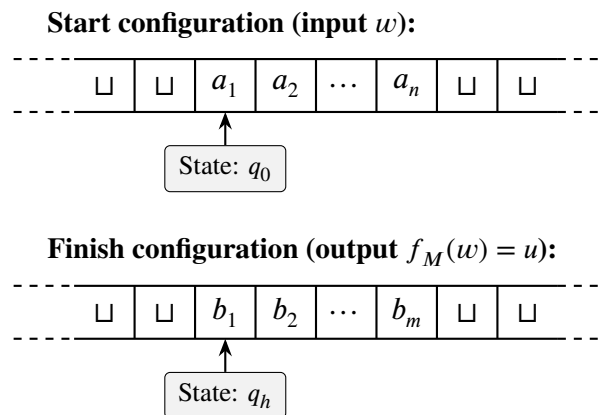
- Q is a finite, non-empty set of *states*;
- Σ is an *input alphabet*;
- $\Gamma \supseteq \Sigma$ is a *tape alphabet* with a special *blank symbol* $\sqcup \in \Gamma \setminus \Sigma$;
- $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$ is the *transition function*;
- $q_0 \in Q$ is the *initial state*;
- $q_h \in Q$ is the *halting state*;

The webpage <https://turingmachine.io/> contains several examples and visualisations of Turing machines, and allows you to program your own. Try it!

3.3 Computing functions and deciding languages on Turing machines

A Turing machine M computes a partial function $f_M: \Sigma^* \rightarrow \Sigma^*$, defined as follows:

1. Write the input word $w = a_1 a_2 \dots a_n$ in n consecutive cells, with blanks in all other cells.
2. Place M 's head on the first symbol of w (or on a blank cell if $w = \epsilon$).
3. Start M from its initial state q_0 and let it run:
 - if M ever reaches the halting state (i.e. *halts*), and the resulting tape contains a word $u \in \Sigma^*$ with all other cells blank, then $f_M(w) = u$, provided that the head is observing the first symbol of u when $u \neq \epsilon$, or a blank cell when $u = \epsilon$;
 - in all other cases (no halting, symbols not from $\Sigma \cup \{\sqcup\}$ on the tape, more than one word on the tape), $f_M(w)$ is undefined.



Here $u = b_1 b_2 \dots b_m$; when $u = \epsilon$, the finish configuration contains only blanks.

Clearly, if $\{0, 1\} \subseteq \Sigma$, we can program a machine M computing some function $f_M: \Sigma^* \rightarrow \{0, 1\}$. This machine will classify words of Σ^* into two classes and **decide** the language $\{w \in \Sigma^* \mid f_M(w) = 1\}$, as we have seen with programs before.

Decide-by-state version. Consider an equivalent definition of a Turing machine, specifically designed to decide languages. Instead of one halting state it has two: an **accepting state** q_a and a **rejecting state** q_r . Run such a machine on a tape with a word w —if it ever halts, look at the state: q_a means $f_M(w) = 1$, q_r means $f_M(w) = 0$. If it never halts, we will assume that $f_M(w)$ is undefined.

It is widely accepted that Turing machines capture our intuitive notion of an algorithm.

The Church–Turing Thesis. *Every function that can be computed by an effective procedure (in the intuitive sense: by a human following explicit rules, or by any physical computing device) can be computed by a Turing machine.*

This statement is not a mathematical theorem and cannot be proved formally. It says that Turing machines provide a good formal definition of the informal notion of an algorithm.

What gives us confidence in the thesis is the remarkable convergence of different models of computation. In the 1930s, several other logicians independently proposed their own definitions of computability: Alonzo Church formulated the λ -calculus (we will see it in Lecture 5), Kurt Gödel and Jacques Herbrand developed recursive functions, and Emil Post introduced production systems. Despite their very different styles, all of these models turned out to compute exactly the same class of functions. Later formalisations—such as Markov normal algorithms (we will meet them in Exercise Session 4), register machines, and modern programming languages—likewise proved equivalent.

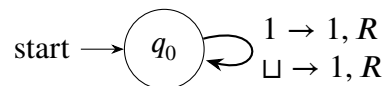
3.4 Examples

As before, we visualise Turing machines as directed graphs where vertices are states and edges represent transitions. A transition $\delta(q, a) = (q', b, D)$ is drawn as an edge from q to q' labelled

$$a \rightarrow b, D$$

meaning: “read a , write b , move in direction D ”.

Example 27. Consider a machine with one work state q_0 over $\Sigma = \{1\}$ and $\Gamma = \{1, \sqcup\}$:



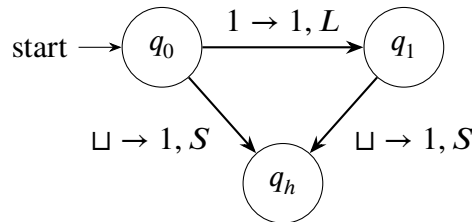
On any input word w , the machine scans rightward indefinitely without ever halting, filling the tape with 1s.⁶

⁶Strictly speaking, we should also specify a halting state q_h —by definition, every machine must have it. But in this machine, it is *unreachable*—no sequence of transitions starting from the initial state ever leads to it, so we omit it from the diagram.

Example 28 (Unary +1). We represent a natural number $n \in \mathbb{N}$ in unary as a sequence of n ones (with 0 represented by the empty word ε).

Given n consecutive 1s on the tape, we wish to produce $n + 1$ consecutive 1s. Rather than walking to the right end of the block, the machine can write 1 to the left of the input: if the head reads 1, it steps one cell left onto the adjacent blank, writes 1, and halts. If the input is empty (the head reads $\sqcup$), it writes 1 in place and halts. In both cases, the machine halts observing the first 1 of the output.

Let $Q = \{q_0, q_1, q_h\}$ with initial state q_0 and halting state q_h :



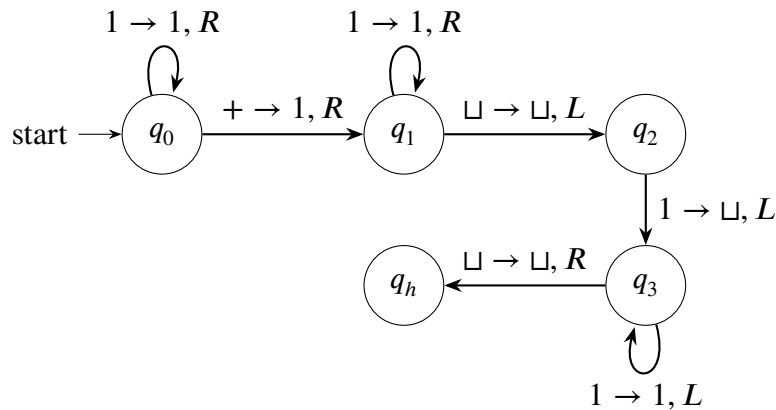
Example 29 (Unary addition). Consider computing the sum of two numbers $m, n \geq 1$ given as two blocks of ones separated by a + symbol:

$$\dots \sqcup \underbrace{11 \dots 1}_m + \underbrace{11 \dots 1}_n \sqcup \dots$$

The procedure is as follows:

1. Walk right across the first block of 1s until reaching the + symbol.
2. Replace + with 1, joining the two blocks into a single block of $m + 1 + n$ ones.
3. Walk to the right end of this block, step back onto the rightmost 1, and erase it (write $\sqcup$), leaving $m + n$ ones.
4. Walk left across all 1s until reaching the blank cell $\sqcup$ to the left of the block, then step one cell right onto the first 1 and halt.

Here is the machine $Q = \{q_0, q_1, q_2, q_3, q_h\}$ with initial state q_0 and halting state q_h :

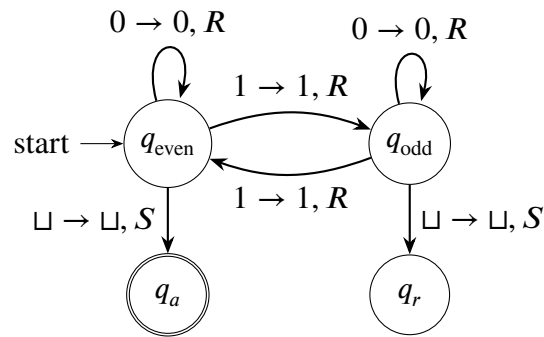


Example 30 (Parity checker). Let us construct a machine with accepting and rejecting states that decides whether a binary string $w \in \{0, 1\}^*$ contains an *even* number of 1s.

We use two work states to track the parity of the number of 1s seen so far: q_{even} (initial state) and q_{odd} . While scanning w , reading 0 keeps the state unchanged, while reading 1 toggles between q_{even} and q_{odd} .

When the machine hits the blank cell $\sqcup$ at the end of the input word:

- from q_{even} , it transitions to the accepting state q_a ;
- from q_{odd} , it transitions to the rejecting state q_r .



3.5 The universal Turing machine

A Turing machine is a single-purpose device: its behaviour on every input is fixed by its transition function. One machine computes binary addition; another decides whether a binary word represents an even number. A computer, however, is more versatile: we program it to perform different tasks. This familiar idea was far from obvious in the 1930s, and we owe its mathematical formulation to Alan Turing.

Turing imagined a universal machine that can be programmed to simulate any other Turing machine. Before we define it, let us remove an apparent complication: different Turing machines may have different tape alphabets.

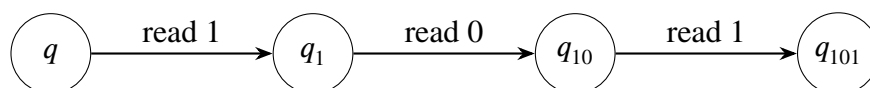
Simulating an alphabet in binary. Let M have tape alphabet Γ . Choose k such that $2^k \geq |\Gamma|$, and assign a different block of k bits to every symbol of Γ . For example, if

$\Gamma = \{\sqcup, a, b, c, d, e, 0, 1\},$	symbol	$\sqcup$	a	b	c	d	e	0	1
	code	000	001	010	011	100	101	110	111

then three cells of M are represented by three blocks on a binary tape:

a	e	c
0 0 1	1 0 1	0 1 1

The simulating machine reads one block at a time. For every state q of M , it has separate states $q_1, q_{10}, q_{101}, \dots$ that record both q and the prefix read so far. While reading the block 101, for instance, it follows:



In state q_{101} , the simulator knows both that M is in state q and that the scanned symbol is e . It can therefore apply the transition prescribed for (q, e) . Further intermediate states remember the new symbol and direction while the simulator rewrites the three bits and moves to the appropriate neighbouring block.

For k fixed in advance, there are only finitely many positions within a block and finitely many prefixes to remember, so all this information fits into finitely many states. We may therefore replace any fixed tape alphabet by a binary one (note that even the blank symbol $\sqcup$ can be encoded as k consecutive 0s, and thus not needed—an empty tape is just a tape filled with 0s).

Consequently, we may freely use finitely many auxiliary tape symbols. For example, we can mark a symbol a by replacing it with a decorated copy $\hat{a}$, or add separators and flags: these are merely extra symbols, and a larger fixed block size encodes them all in binary.

Encoding a machine. It now suffices to encode machines whose tape alphabet is $\{0, 1\}$, with 0 serving as the blank symbol. Suppose such a machine M has m states. Number them

$$q_0, q_1, \dots, q_{m-1},$$

renaming if necessary so that q_0 is initial and q_{m-1} is halting. Begin the code with m ones followed by a zero. From this prefix, a decoder learns m and hence the fixed number of bits needed to write any state number.

Next list the two transitions from each state, first the one for reading 0 and then the one for reading 1. A transition record contains the fixed-width binary number of the new state, the bit to write, and two bits for the direction (say, 00 for L , 01 for R , and 10 for S). There are exactly $2m$ such records, all of the same known length. Their concatenation after the prefix is therefore a binary word which unambiguously encodes the machine M . We will denote this word by $[M]$.

The input w is already binary. We encode the pair consisting of M and w , for example, as a word $\langle [M], w \rangle$, where $[M]$ is the binary word above, while $\langle, \rangle$ and the comma are special symbols that can also be encoded in binary. The particular convention is unimportant; what matters is that a fixed Turing machine can decode it and determine where $[M]$ ends and w begins.

Simulating an encoded machine. A **universal Turing machine** U receives $\langle M, w \rangle$ and maintains three pieces of information on its tape:

- the description $[M]$;
- the current state of the simulated machine;
- an encoding of its tape, with one marked cell (say $\hat{0}$ instead of 0, or $\hat{1}$ instead of 1) indicating the position of the head of M .

Initially these contain $[M]$, q_0 , and the input w . To simulate one step, U reads the marked cell, scans $[M]$ for the transition with the current state and symbol, writes the new symbol, moves the marker to the neighbouring cell, and records the new state. If the marker moves beyond the represented part of the tape, U adds a cell containing the blank symbol 0. Each operation is a finite combination of scans, comparisons, and copies, so a fixed Turing machine can perform it. Repeating this procedure simulates every step of M .

Theorem 31 (Turing, 1936). *There exists a Turing machine U such that, for every Turing machine M and input w ,*

$$U(\langle M, w \rangle) = M(w).$$

That is, U halts exactly when M halts, and in that case produces the same output.

The universal machine is the theoretical model of a general-purpose computer: $[M]$ is a program, w is its input, and U executes the program. We can now restate Theorem 7 for Turing machines:

Theorem 32 (Turing, 1936). *The language*

$$\text{HALT}_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ halts on } w\}$$

is undecidable.

Lecture 5. Lambda calculus

So far, we have emphasised that programs can be treated as data. Now, we will study another formal definition of an algorithm, historically one of the earliest, where the tables are turned. In Alonzo Church's lambda calculus, everything is a program.

5.1 Functions, in math and in lambda calculus

We are used to defining functions by writing $f(x) = x + 1$, or, sometimes, by $f : x \mapsto x + 1$. Applying f to 4 gives 5: $f(4) = 5$.

In lambda calculus, the same function is denoted by $\lambda x.x + 1$ —literally, “a function that takes x and returns $x + 1$ ”.⁷ Application goes like this:

$$(\lambda x.x + 1)4 = 4 + 1 = 5$$

This is not pure lambda calculus: to keep this first example simple, we use numbers and the notation $+1$ as if they were defined. Soon we will construct these things, and many more.

A function may return another function. For example,

$$(\lambda x.\lambda y.x + y)3 = \lambda y.3 + y$$

Applying the result to 4 gives 7. Thus, a function of many arguments can be represented by a function receiving the first argument and returning a function of the rest of the arguments, with the first one fixed. This is called **currying**.⁸

Definition 33. A *lambda term* is a variable x , an application (MN) of one term to another, or an abstraction $(\lambda x.M)$.

Application associates to the left: MNP means $(MN)P$; the parentheses around terms are omitted where unnecessary. The body of an abstraction extends as far to the right as possible: $\lambda x.MN$ means $\lambda x.(MN)$.

5.2 Computation by substitution

The basic computation rule is **beta reduction**.⁹

$$(\lambda x.M)N \longrightarrow_{\beta} M[x := N]$$

Read this as: replace the occurrences of the parameter x in the body M by the argument N . It may be performed inside a larger term, including inside the body of an abstraction. We write $\longrightarrow_{\beta}^*$ for zero or more applications of beta reduction.

⁷Why λ ? Church never explained clearly. Wikipedia's page on lambda calculus contains some details.

⁸Chicken curry is not relevant here, Haskell Curry is.

⁹Why β ?! Because there is a simpler one, called α , so it's just numbering (and Church's love for Greek letters).

Example 34. Let $ID = \lambda x.x$ and $K = \lambda x.\lambda y.x$. Then:

$$IDA \longrightarrow_{\beta} A \quad KAB \longrightarrow_{\beta} (\lambda y.A)B \longrightarrow_{\beta} A$$

The first function returns its argument; the second keeps its first argument and discards the second.

A variable occurrence is **bound** if it is introduced by a surrounding λ ; otherwise it is **free**. In $\lambda x.xy$, the occurrence of x is bound and that of y is free. A term with no free variables is **closed**. Substitution replaces only free occurrences: in $\lambda x.x$, the x is bound, it is a local name for $\lambda x.x$.

Python has a similar thing, when one function is defined inside another:

```
def outer(x):
    def inner(x):          # a different parameter
        return x + 10     # inner's x
    return inner(x)       # outer's x

outer(5)  # 15
```

Calling `outer(5)` does not change `inner`'s definition; its parameter receives a value only when `inner(5)` is called. Similarly,

$$(\lambda x.x(\lambda x.x))y \longrightarrow_{\beta} y(\lambda x.x)$$

The name of a local parameter does not matter: $\lambda x.x$ and $\lambda z.z$ describe the same function. Consistently renaming a bound variable to a fresh name, meaning one not already occurring in the term, is called **alpha conversion**: $\lambda x.x \equiv_{\alpha} \lambda z.z$. We will identify terms that differ only by such renaming.

Mind the bounds! The term $K = (\lambda x.\lambda y.x)$ of Example 34 says: "take x , remember it, then take whatever and return x ". Imagine we apply it to y : $(\lambda x.\lambda y.x)y$. A 'dumb' reduction substitutes y for x in the body and produces $\lambda y.y$: we were supposed to remember the first argument (which was y), but it is now lost! Indeed, $\lambda y.y$ will take an argument and output *that argument*, not the term y passed as the first argument to K . So this is a wrong way of doing beta reductions. The correct way is: when substituting M for the free occurrences of x in N , first rename any bound parameters of N that would otherwise bind free occurrences of variables in the inserted M , using names fresh for both terms:

$$(\lambda x.\lambda y.x)y \equiv_{\alpha} (\lambda x.\lambda z.x)y \longrightarrow_{\beta} \lambda z.y$$

Now the answer is a constant function returning the external y , and we are happy. This convention is called **capture-avoiding substitution**.

Once again, let us consider an example from Python.

```
y = 10

def add_to_y(x):
    return x + y

def calculate(y):
    return add_to_y(3)

calculate(100)  # 13
```

Inside `add_to_y`, `y` refers to the global variable, regardless of where the function is called. Suppose we replace the call by its body, substituting 3 for `x`:

```
def calculate(y):
    return 3 + y

calculate(100)  # 103: wrong!
```

The copied occurrence of `y` now refers to `calculate`'s parameter: it has been accidentally captured. Renaming that parameter before copying the body preserves the meaning:

```
def calculate(t):
    return 3 + y

calculate(100)  # 13
```

Python itself does not substitute text when calling functions, but in λ -calculus, which computes exactly by substitutions, we must use α -conversion to avoid problems as above.

5.3 Functions as arguments of other functions

Function composition is itself a function:

$$\text{COMP} = \lambda f. \lambda g. \lambda x. f(gx)$$

What does it do? It takes the first argument, f , then the second, g , and returns a function that takes x and applies g to x , then f to $g(x)$ (in standard math notation), then returns the result $f(g(x))$. Indeed, $\text{COMP}FGX \rightarrow_{\beta}^* F(GX)$.

Example 35. Let $\text{DBL} = \lambda f. \lambda x. f(fx)$, the operation that applies a function twice. Then

$$\text{DBL}FX \rightarrow_{\beta}^* F(FX) \quad \text{DBL}(\text{DBL}F)X \rightarrow_{\beta}^* F(F(F(FX)))$$

Note the parentheses. If we write without them, left-associativity will enforce another reading: $\text{DBL DBL}FX$ is $((\text{DBL DBL})F)X$. In this case, however, the result is the same. Apply the definition of the *outer* DBL first: it gives $\text{DBL}(\text{DBL}F)X$, so again F is applied four times.

5.4 Data as functions

5.4.1 Booleans

What must a Boolean do? It must choose between two alternatives. Define

$$\text{TRUE} = \lambda x. \lambda y. x \quad \text{FALSE} = \lambda x. \lambda y. y$$

So TRUE is our old friend K , and FALSE is the dual: “Take the first argument, forget it, take the second and return it”.

Then $\text{TRUE } AB \longrightarrow_{\beta}^* A$ and $\text{FALSE } AB \longrightarrow_{\beta}^* B$. A conditional needs no additional mechanism:

$$\text{IF} = \lambda p. \lambda x. \lambda y. pxy$$

That is, $\text{IF } ABC$ is the λ -way of writing “if A is true, then B , else C ”. Indeed:

$$\text{IF TRUE } BC \longrightarrow_{\beta}^* \text{TRUE } BC \longrightarrow_{\beta}^* B$$

and

$$\text{IF FALSE } BC \longrightarrow_{\beta}^* \text{FALSE } BC \longrightarrow_{\beta}^* C$$

To negate a Boolean, exchange its two alternatives:

$$\text{NOT} = \lambda p. \lambda x. \lambda y. pyx$$

For example,

$$\text{NOT TRUE} \longrightarrow_{\beta} \lambda x. \lambda y. \text{TRUE } yx \longrightarrow_{\beta}^* \lambda x. \lambda y. y = \text{FALSE}$$

Conjunction can be implemented by

$$\text{AND} = \lambda p. \lambda q. pqp$$

That is, if p is true, return q ; otherwise return p , which is already false.

5.4.2 Natural numbers

If the only thing we can do is apply functions, how can we simulate a number $n \in \mathbb{N}$? Well, the number n can tell us to apply a function n times. Define the **Church numerals**

$$\bar{0} = \lambda f. \lambda x. x \quad \bar{1} = \lambda f. \lambda x. fx \quad \bar{2} = \lambda f. \lambda x. f(fx) \quad \dots$$

In general, $\bar{n} = \lambda f. \lambda x. f^n(x)$, where $f^0(x) = x$ and $f^{n+1}(x) = f(f^n(x))$. The superscript here is an abbreviation for repeated application. In particular, DBL from Example 35 is $\bar{2}$.

To obtain $n + 1$, the successor of n , perform its repetitions and apply f once more. That is, $\overline{n+1}$ can be represented as $\lambda f. \lambda x. f(\bar{n}fx)$. Literally: “take f , take x , apply f to x exactly n times, and then once more”.

So the function that takes $\bar{n}$ and returns $\overline{n+1}$ is then:

$$\text{SUCC} = \lambda n. \lambda f. \lambda x. f(nfx)$$

For example,

$$\text{SUCC } \bar{2} \longrightarrow_{\beta}^* \lambda f. \lambda x. f(f(fx)) = \bar{3}$$

To add m and n , apply successor m times, starting at $\bar{n}$:

$$\text{ADD} = \lambda m. \lambda n. m \text{ SUCC } n$$

A Church numeral can iterate any function, including a function on Church numerals. Here $\bar{m}$ receives SUCC as the function to repeat and $\bar{n}$ as the starting value. For example,

$$\text{ADD } \bar{2} \bar{3} \longrightarrow_{\beta}^* \text{SUCC}(\text{SUCC } \bar{3}) \longrightarrow_{\beta}^* \bar{5}.$$

For multiplication, add n a total of m times, starting at zero:

$$\text{MULT} = \lambda m. \lambda n. m (\text{ADD } n) \bar{0}$$

Note the parentheses! $\text{ADD } n$ is a function that takes a numeral and adds n to it. Repeating this function m times from $\bar{0}$ gives $\bar{n}, \bar{2n}, \dots, \bar{mn}$; if $m = 0$, the result is simply $\bar{0}$.

Remark 36. Note that $\bar{0}$ and FALSE are the same term up to parameter names.

5.5 Halting and nontermination

What would be the λ -counterpart of our `never_stop` program? Of the Turing machine that runs forever to the right?

A term is in **normal form** if it contains no subexpression of the form $(\lambda x. M)N$. For example, $\lambda x. x$ and all Church numerals are normal forms. A term *has* a normal form if some finite sequence of reductions reaches one.

Let

$$\Delta = \lambda x. xx \quad \Omega = \Delta \Delta$$

Then $\Omega \longrightarrow_{\beta} \Omega \longrightarrow_{\beta} \dots$. Its only possible step reproduces the same term, so it has no normal form. Notice that Δ itself is a normal form.

Remark 37. Compare Ω to the ‘diagonal’ program δ of Theorem 7. The former takes the input and applies it to itself. The latter checked whether the input program halts on its own code.

Different subexpressions may be ready to reduce:

$$(\lambda x. \text{ID}) \Omega \longrightarrow_{\beta} \text{ID} \quad \text{and} \quad (\lambda x. \text{ID}) \Omega \longrightarrow_{\beta} (\lambda x. \text{ID}) \Omega$$

That is, reducing the outer application gives ID immediately. Reducing inside the argument can continue forever. Thus, the existence of an infinite reduction does *not* prove that a term has no normal form.

Theorem 38 (Church–Rosser). *If $M \longrightarrow_{\beta}^* P$ and $M \longrightarrow_{\beta}^* Q$, then there is a term R such that $P \longrightarrow_{\beta}^* R$ and $Q \longrightarrow_{\beta}^* R$, allowing renaming of bound variables (alpha conversions).*

This property is called **confluence**. In particular, a term cannot have two different normal forms: neither could reduce further to a common result. Confluence does not say that every choice of reductions terminates. There is, however, a strategy for choosing what to reduce first.

Theorem 39 (Normalisation). *If a term has a normal form, repeatedly reducing the leftmost outermost reducible application reaches it.*

‘Outermost’ means that the application is not inside another reducible application; among those, take the leftmost one.

5.6 Trying it in Scheme

Scheme is a *functional programming language* in which functions can be passed around just as in our examples. An online interpreter, such as <https://try.scheme.org/>, lets us try:

```
(define dbl (lambda (f) (lambda (x) (f (f x)))))  
(define succ (lambda (x) (+ x 1)))  
  
(( dbl succ) 10)           ; 12  
((( dbl dbl) succ) 10)    ; 14
```

Here `define` names a value, `(f x)` applies it, and semicolons introduce comments. Scheme supplies built-in numbers and evaluates arguments before function calls.

Lecture 6. More lambda calculus

We have represented Booleans and numbers by lambda terms. We now turn to *data structures*: pairs and lists. To work with lists, we will need *recursion*. Fasten your seatbelts.

6.1 Pairs

To store two values, make a function that passes both to whoever wants to use them:

$$\text{PAIR} = \lambda a.\lambda b.\lambda f.f\ ab \quad \text{FST} = \lambda p.p\ \text{TRUE} \quad \text{SND} = \lambda p.p\ \text{FALSE}$$

Thus $\text{FST}(\text{PAIR } AB) \rightarrow_{\beta}^* A$, and similarly for the second component.

Embrace the λ -philosophy: a pair is a function; to get its first or second value, *apply it* to the right argument. If you want a sum of the two components, apply your pair to the function of addition: $(\lambda f.f\ ab)\ \text{ADD} \rightarrow_{\beta} \text{ADD } ab$.

Example 40. We use pairs to compute the predecessor function (i.e. $\max\{x - 1, 0\}$) for Church numerals. Keep the previous and current counter values in a pair. Starting from $(0, 0)$, the update $(\overline{m}, \overline{n}) \mapsto (\overline{n}, \overline{n + 1})$ gives $(0, 1), (1, 2), (2, 3), \dots$. It is implemented by

$$\text{UPD} = \lambda p.\text{PAIR}(\text{SND } p)(\text{SUCC}(\text{SND } p)).$$

After n updates, the first component is $\max\{n - 1, 0\}$, so

$$\text{PRED} = \lambda n.\text{FST}(n\ \text{UPD}(\text{PAIR } \overline{0}\ \overline{0})).$$

We can now implement *truncated subtraction*: $m \div n = \max\{m - n, 0\}$. Apply predecessor n times to m . Once zero is reached, it stays zero:

$$\text{SUBT} = \lambda m.\lambda n.n\ \text{PRED } m$$

6.2 Recursion in lambda calculus

In Python, we can define a function that calls itself:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

In pure lambda calculus, a term is a *finite syntactic expression*. Writing $\text{TRUE} = \lambda x.\lambda y.x$ is just a naming convention: “wherever you see TRUE , substitute $\lambda x.\lambda y.x$ ”. We cannot define a term literally as, say, $A = \lambda x.Ax$. It’s like saying that inside the text of ‘The Three Musketeers’ there is a chapter containing

exactly the text of the full ‘The Three Musketeers’.¹⁰ An equation describing how a term *computes*, however, is a different matter.

Two terms A and B are **β -equal**, written $A =_{\beta} B$, if one can pass from one to the other by finitely many beta reductions, reversed beta reductions, and alpha conversion. By Theorem 38, two β -equal terms either both lack a normal form or have the same normal form, up to alpha conversion. This remains true when β -equal terms are placed in the same surrounding term, so either can replace the other in a program without changing its normal-form result. Their reduction sequences, however, need not be the same.

Suppose we want a term FACT implementing factorial. Its body must test for zero and, in the other case, use FACT on the predecessor. First write this body with a parameter f in place of the recursive function:

$$G = \lambda f. \lambda n. \text{IF} (\text{ISZERO } n) \bar{1} (\text{MULT } n (f (\text{PRED } n))).$$

Here ISZERO is the zero test from Exercise sheet 5, and PRED is the predecessor function from Example 40. Note that $G f$ performs one layer of the computation, delegating the smaller factorial to the function f that it takes as a parameter. It is not yet factorial, because that function still has to be supplied.

We want the resulting function to use *itself* for the delegated calls. Thus we seek a term FACT satisfying

$$\text{FACT} =_{\beta} G \text{ FACT}.$$

Such a term FACT is called a **fixpoint**¹¹ of G . The goal is to find a term satisfying this equation for our chosen body G .

Theorem 41 (Fixpoint theorem). *There is a closed term Θ such that, for every term F ,*

$$\Theta F \longrightarrow_{\beta}^* F(\Theta F).$$

Proof. Put $U = \lambda u. \lambda f. f(uuf)$ and $\Theta = UU$. Then

$$\Theta F = UUF \longrightarrow_{\beta} (\lambda f. f(UUf))F \longrightarrow_{\beta} F(UUF) = F(\Theta F). \quad \square$$

Set $\text{FACT} = \Theta G$. Unfolding FACT gives $G \text{ FACT}$: the parameter f in the body is replaced by FACT itself. For factorial, this gives

$$\text{FACT } \bar{0} \longrightarrow_{\beta}^* \bar{1}, \quad \text{FACT } \bar{3} \longrightarrow_{\beta}^* \text{MULT } \bar{3} (\text{FACT } \bar{2}).$$

Reducing further similarly reaches $\text{FACT } \bar{1}$ and then $\text{FACT } \bar{0}$, which is the base of recursion. We therefore obtain $\bar{6}$. Reduction produces each new call as needed; after expanding the abbreviations, ΘG itself is still a finite term.

Remark 42. Theorem 41 finds a fixpoint of a function F among all lambda terms, not necessarily the terms that F was ‘supposed to’ work on. For example, ΘSUCC , the fixpoint of the successor function, cannot be a Church numeral: SUCC sends every numeral to a different one.

¹⁰Formally, let $|B|$ denote the length of a term B , i.e. the number of symbols used to write it down. Literal equality would give $|A| = |\lambda x. Ax| > |A|$, which is impossible.

¹¹A grammatically correct way of writing is ‘fixed-point’, while ‘fixpoint’ is slang in theoretical computer science.

6.3 Scott lists

To inspect and modify sequences, we will use Scott lists (named after Dana Scott, Church's student and a star theoretician himself), which are different from Church lists from Exercise sheet 6. A Church list supplies the result of folding its tail; a Scott list supplies the tail itself, which is convenient for inspecting a list one element at a time.

A Scott list has two possibilities: it is empty, or it consists of a head and a tail. Encode these alternatives as

$$\text{NIL} = \lambda z. \lambda c. z, \quad \text{CONS} = \lambda h. \lambda t. \lambda z. \lambda c. cht.$$

Here CONS stands for 'constructor': given a head (an element) and a tail (a list), it constructs a new list, with the given head followed by the given tail.

Given an empty-case value Z and a nonempty-case function C , a list L behaves as follows:

$$\text{NIL } Z C \longrightarrow_{\beta}^* Z, \quad (\text{CONS } H T) Z C \longrightarrow_{\beta}^* C H T.$$

A nonempty list passes its head and its *actual tail* to the selected function. For example,

$$\text{TAIL} = \lambda l. l \text{ NIL } (\lambda h. \lambda t. t)$$

returns the tail of a nonempty list, and NIL on an empty list.

Appending one list to another

We want concatenation to satisfy:

$$\text{APP NIL } b =_{\beta} b, \quad \text{APP (CONS } h t) b =_{\beta} \text{CONS } h (\text{APP } t b).$$

The first equation handles the empty list. The second keeps the head and concatenates the tail with b . Turn this description into a function of the recursive call:

$$F = \lambda f. \lambda l_1. \lambda l_2. l_1 l_2 (\lambda h. \lambda t. \text{CONS } h (f t l_2)), \quad \text{APP} = \Theta F.$$

Unfolding the fixpoint once gives

$$\text{APP } l_1 l_2 \longrightarrow_{\beta}^* l_1 l_2 (\lambda h. \lambda t. \text{CONS } h (\text{APP } t l_2)).$$

What is going on here? The first list, supplied for l_1 , is itself a case selector: if $l_1 = \text{NIL}$, it returns the empty-case value l_2 ; if $l_1 = \text{CONS } h t$, it calls the second argument with h and t , giving $\text{CONS } h (\text{APP } t l_2)$. No separate emptiness test is needed.

Example 43. Write $[A, B]$ for the Scott list with elements A , B , and $[]$ for NIL. Let us concatenate $[A, B]$ and $[C]$:

$$\begin{aligned} \text{APP } [A, B] [C] &\longrightarrow_{\beta}^* \text{CONS } A (\text{APP } [B] [C]) \\ &\longrightarrow_{\beta}^* \text{CONS } A (\text{CONS } B (\text{APP } [] [C])) \\ &\longrightarrow_{\beta}^* [A, B, C]. \end{aligned}$$

Each recursive call shortens the first list, so the computation terminates on finite lists (unless a list itself contains a non-terminating element such as Ω).

6.4 Computing functions on words

Lambda calculus is Turing-complete: any function $\Sigma^* \rightarrow \Sigma^*$ computable by a Turing machine can be computed by a lambda term. This statement may not be clear yet—in lambda calculus, we apply terms to other terms. So how can a term ‘compute’ a function on words?

For example, we can encode every word $w = a_1 \dots a_n$ by the normal-form Scott-list encoding $L_w = [A_1, \dots, A_n]$, where the terms A_i encode the letters a_i , in some way.¹² Then, for every computable partial function $f : \Sigma^* \rightarrow \Sigma^*$ there exists a term F such that:

1. if $f(w) = u$, then $F L_w \rightarrow_{\beta}^* L_u$;
2. if $f(w)$ is undefined, then $F L_w$ does not have a normal form.

To have an idea of how this is possible, recall that Markov normal algorithms can already simulate Turing machines (Exercise sheet 4). To simulate a step of a given normal algorithm on a list L_w , inspect it one element at a time, choose the *first applicable rule*, and replace its *leftmost occurrence*. Recursive functions on lists can compare symbols, search for the pattern, and rebuild the word with the replacement, using concatenation as above. The fixpoint operator lets us repeat these steps, restarting the rule search after each replacement. A terminal rule stops immediately; if no rule applies, return the current word. We omit the full construction.

6.5 Scheme again

In Scheme, the recursive equations for concatenation can be written directly:

```
(define join
  (lambda (xs ys)
    (if (null? xs)
        ys
        (cons (car xs) (join (cdr xs) ys)))))

(join '(a b) '(c))    ; (a b c)
```

Here `car`, `cdr`, and `cons` give the head, tail, and list constructor. The quote marks list data; `'()` is empty. Scheme supplies recursive definitions directly and allows several arguments in one call.

¹²Say, if $\Sigma = \{b_1, \dots, b_m\}$ has m letters, the term for the letter b_i could be $\lambda x_1 \dots \lambda x_m. x_i$, the *selector* of the i th argument.

Lecture 7. Time complexity, classes P and EXP

We now turn from “what can computers do?” to “what can computers do efficiently?” A program that takes a thousand years to finish is of little practical use. How fast can an algorithm be? This is a question for computational complexity theory.

7.1 Running time of an algorithm

The running time of an algorithm usually depends on the size of its input. Finding shortest paths in a large graph, for example, may take longer than in a small one. Here is a more direct example: if a Turing machine working over unary alphabet takes a natural number n and must output 2^n , it needs at least 2^n steps just to write down the answer.

To measure efficiency, we therefore ask *how running time grows with input size*.

In our formalisation, inputs are words, and the size of an input word w is its length, denoted $|w|$. The empty word ε has length 0. Representation matters: the number 5 takes one digit in decimal, three bits in binary (101), and five symbols in unary (11111). Still, any fixed finite input alphabet Σ can be encoded in binary using $\max\{1, \lceil \log_2 |\Sigma| \rceil\}$ bits per symbol. This increases word length by at most a constant factor.

For precise definitions, we fix a model of computation: multitape Turing machines (see Exercise 3.6). For simplicity, we focus on decision problems, or deciding languages. A decider has two halting states, accepting and rejecting, and reaches one of them on every input (see Section Lecture 3.).

Definition 44. Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a function. A Turing machine M runs in time $O(f(n))$ if it halts on every input and there are constants $c > 0$ and k such that, for every input word w with $|w| \geq k$, M halts within $c \cdot f(|w|)$ steps.

This is a worst-case bound for a particular algorithm represented by M . The notation $O(f(n))$ ignores a constant factor and finitely many small inputs. That is useful when comparing growth rates, but the constant may matter greatly in practice: $10^{10}n$ is still $O(n)$.

Complexity theory also asks how difficult a computational task is, regardless of the algorithm chosen to solve it. We start with the following definition for decision problems.

Definition 45. A language $L \subseteq \Sigma^*$ is decidable in time $O(f(n))$ if there exists a Turing machine that decides L and runs in time $O(f(n))$.

We can now classify languages by how fast they can be decided. For every $f : \mathbb{N} \rightarrow \mathbb{N}$, we obtain a *complexity class*:

$$\text{TIME}(f(n)) = \{L \subseteq \Sigma^* \mid L \text{ is decidable in time } O(f(n))\}$$

Example 46. Class $\text{TIME}(1)$ comprises languages decidable in constant time. For example, $\emptyset \in \text{TIME}(1)$ (reject immediately) and $\Sigma^* \in \text{TIME}(1)$ (accept immediately). Here is a less trivial example:

$$\text{REV_EVEN} = \{w \in \{0, 1\}^* \mid \text{the reverse of } w \text{ represents an even number in binary}\}$$

A binary number is even precisely when its last bit is 0. Since a Turing machine starts at the first bit of w , it can accept when this bit is 0 (or when $w = \varepsilon$) and reject when it is 1. It need not scan the rest of w , so this really is a constant-time decision procedure.

Example 47. Class TIME (n) contains languages decidable in linear time. For example,

$$\text{PARITY} = \{w \in \{0, 1\}^* \mid w \text{ contains an even number of 1s}\}$$

A Turing machine scans the word once, keeping the parity of the number of 1s in its state. Here is another example:

$$\text{EVEN} = \{w \in \{0, 1\}^* \mid w \text{ represents an even number in binary}\}$$

A machine decides this by scanning to the end and checking the last bit.

7.2 Polynomial and exponential time

For the language EVEN from Example 47, one can prove that $\text{EVEN} \notin \text{TIME}(1)$: a machine that runs in constant time cannot reach the last bit of a sufficiently long word. This proof depends on the model. A computer with random-access memory, given the input length, could jump to the last bit. Exact time classes such as TIME (1) are therefore sensitive to the underlying model.

Lower bounds for familiar problems are usually much harder to prove. A slow algorithm tells us only how long *that* algorithm takes. To show that a language has no faster algorithm, we must rule out every possible algorithm, including ones we have not discovered.

We therefore consider larger classes that give a more robust view of running time. The class P contains languages decidable in polynomial time; EXP contains those decidable in exponential time:

$$P = \bigcup_{k \geq 0} \text{TIME}(n^k), \quad \text{EXP} = \bigcup_{k \geq 0} \text{TIME}(2^{n^k}).$$

These larger classes smooth over many differences between reasonable models of computation. A multitape Turing machine can simulate a standard computer program with at most polynomial overhead, provided we count the bit cost of operations on large numbers; the converse also holds.

Example 48. Consider the language

$$\text{PALINDROME} = \{w \in \{0, 1\}^* \mid w \text{ reads the same forwards and backwards}\}.$$

A one-tape machine can repeatedly compare and cross off the leftmost and rightmost unmatched symbols. Each round scans at most $O(n)$ cells, and there are at most $n/2$ rounds. Thus $\text{PALINDROME} \in \text{TIME}(n^2) \subseteq P$.

Example 49. In SUBSET-SUM, the input is a list of nonnegative binary integers $a_1, \dots, a_m$ and a target t . The language is

$$\text{SUBSET-SUM} = \left\{ \langle a_1, \dots, a_m, t \rangle \mid \exists I \subseteq \{1, \dots, m\} : \sum_{i \in I} a_i = t \right\}.$$

We can test each of the 2^m subsets I . For each one, add the selected numbers and compare the sum with t . These operations take polynomial time in the input length n . Since $m \leq n$, the whole search takes $2^m \text{ poly}(n) = 2^{O(n)}$ time. Thus $\text{SUBSET-SUM} \in \text{EXP}$.

We should keep in mind that the classes P and EXP hide important details. As noted above, $O(f(n))$ hides a constant factor. The class P, moreover, puts linear-time languages in the same class as those for which only, say, $O(n^{20})$ algorithms are known. We nevertheless treat polynomial time as a theoretical notion of *efficient computation*.

This convention can seem paradoxical. Why is n^{100} ‘efficient’, while $n^{\log \log \log n}$ is not? The latter function eventually exceeds every polynomial, but its exponent is small on inputs we could realistically store. Polynomial time describes long-run growth, not a guarantee of practical speed.

Still, exponential running time often comes from checking every candidate in a large set (as in Example 49). Polynomial algorithms, on the other hand, usually exploit some ‘smart ideas’. It is not rare that an initially high-degree polynomial algorithm is improved over time by refining the underlying idea.

7.3 Time hierarchy

Example 49 gave an upper bound on the complexity of SUBSET-SUM. Can we do better? Or can we prove that there is no polynomial-time algorithm for this problem, i.e. that SUBSET-SUM $\notin$ P? Actually, do we know *any* language outside P? The next theorem says yes, although its proof constructs a rather artificial example. For SUBSET-SUM, membership in P is open.

Theorem 50. *For all $k \geq 0$, $\text{TIME}(n^k) \subset \text{TIME}(n^{k+1})$. Moreover, $P \subset \text{EXP}$.*

Proof. The non-strict inclusions are immediate: for every fixed k , $n^k = O(n^{k+1})$ and $n^k = O(2^n)$. For $k = 0$, strictness follows from $\text{EVEN} \in \text{TIME}(n) \setminus \text{TIME}(1)$. We give the diagonal argument for $\text{TIME}(n) \subset \text{TIME}(n^2)$; the other positive integer exponents work similarly.

Consider words of the form $\langle M \rangle \# 0^r$, where $\langle M \rangle$ is an encoding of the Turing machine M suitable for the simulation of M by the universal machine, $\#$ is a separator symbol (only given here for visual convenience), and 0^r is a sequence of $r \geq 2$ zeros.

On input $w \in \{0, 1, \#\}^*$, a machine M' rejects if w is not of the form above or if $|\langle M \rangle| > \log r$. Otherwise, M' simulates M on *the whole input* w for at most $\lfloor n^{3/2} \rfloor$ steps, where $n = |w|$. If M halts, M' gives the opposite answer (i.e. rejects if M accepts and accepts if M rejects); if M does not halt within the limit, M' rejects. Let L' be the language decided by M' .

First, $L' \in \text{TIME}(n^2)$. This uses a standard fact about universal simulation: a fixed multitape machine can simulate T steps of a machine whose description has $O(\log n)$ bits in $O(n + T(\log n)^c)$ time, for some fixed c , when T is polynomial in n .¹³ Parsing w and maintaining the time limit also fit within $O(n^2)$. With $T = \lfloor n^{3/2} \rfloor$, the total is $O(n^2)$.

Second, $L' \notin \text{TIME}(n)$. Suppose a machine M decided L' in at most cn steps for all sufficiently long inputs. Choose r large enough that $|\langle M \rangle| \leq \log r$ and $cn < \lfloor n^{3/2} \rfloor$ for $w = \langle M \rangle \# 0^r$. On this input, M' sees M halt and returns the opposite answer, contradicting the assumption that they decide the same

¹³These details are very technical, but interested readers can consult [Arora & Barak, Computational Complexity: A Modern Approach]

language. For any fixed $k \geq 1$, a limit of $n^{k+1/2}$ steps gives the analogous separation between $\text{TIME}(n^k)$ and $\text{TIME}(n^{k+1})$.

The same argument proves $P \subset \text{EXP}$: simulate $M(w)$ for up to 2^n steps, where $n = |w|$, and give the opposite answer if M halts. Even with the overhead of universal simulation, this machine runs in exponential time, while 2^n eventually exceeds the running time of any polynomial-time M . $\square$

Lecture 8. Class NP

We will use Turing machines that decide languages by halting in one of two states: accepting or rejecting (see Section Lecture 3.).

Definition 51. A language $L \subseteq \Sigma^*$ is in NP if there are a polynomial $p(n)$ and a Turing machine V (the verifier), running in polynomial time in the length of its input pair, such that, for every $w \in \Sigma^*$,

$$w \in L \iff \exists s \in \Sigma^* \text{ such that } |s| \leq p(|w|) \text{ and } V \text{ accepts } (w, s).$$

The word s is called a certificate for w .

In other words, a word w belongs to a language in NP when its membership can be ‘justified’ by a not-too-long proof (at most polynomial in the length of w). This proof must also be ‘checkable’ in polynomial time. It need not be unique: every $w \in L$ has at least one proof, while no $w \notin L$ has a proof accepted by the verifier. We do not require the verifier to find a proof.

Example 52. Boolean formulas are built from the constants T (true) and F (false), variables $x_1, x_2, \dots$, and the Boolean operations negation, conjunction, and disjunction. A formula $\varphi(x_1, \dots, x_n)$ is called *satisfiable* if some assignment of Boolean values to its variables makes it true. Thus we have a language:¹⁴

$$\text{SAT} = \{\varphi \text{ a Boolean formula} \mid \varphi \text{ is satisfiable}\}$$

As usual, we assume that formulas are encoded as words in some alphabet, say $\{T, F, x, 0, 1, \neg, \wedge, \vee, (,)\}$, and thus can be given as input to a Turing machine.

We prove that $\text{SAT} \in \text{NP}$. List the variables that occur in φ as $x_1, \dots, x_n$. Let V take a pair (φ, s) , where $s \in \{T, F\}^n$ is a word of n letters, and interpret its i th letter as the value assigned to x_i . The machine evaluates φ on this assignment, accepting if it becomes true and rejecting otherwise. It also rejects malformed inputs: for example, φ may not be a well-formed formula, or s may not be a sequence of n Boolean values. Parsing and evaluating the formula take polynomial time. For each satisfiable formula, a satisfying assignment is the required polynomial-size certificate: there are at most $|\varphi|$ distinct variables in the encoded formula. For a non-satisfiable formula, or an input that is not a formula, no certificate is accepted.

Example 53. In a graph, a *clique* is a set of pairwise adjacent nodes. Consider the following language:

$$\text{CLIQUE} = \{\langle G, k \rangle \mid G \text{ is an undirected graph, } k \geq 0, \text{ and } G \text{ has a clique of size at least } k\}$$

Again, $\text{CLIQUE} \in \text{NP}$. If the input is malformed, or if $k > |V(G)|$, the verifier rejects immediately. Otherwise, the certificate lists k distinct nodes of G . The verifier checks that every pair in the list is adjacent. Since $k \leq |V(G)|$, the certificate has polynomial length, and the checks take polynomial time.

In the two examples above, membership in the language is decided by establishing the existence of something, and this something itself plays the role of the certificate: it is the witness whose existence is at issue. This is a common pattern when talking about class NP, and we will see it again and again. In fact, the usual decision problem associated with a language $L \in \text{NP}$ —“decide whether $w \in L$ ”—is reformulated by the definition of NP as “decide whether there exists a certificate for w that the verifier accepts”. This, in turn, connects it to a *search problem*—“find the certificate”. Class NP is important because many practical search problems (find this or that in a graph, find an assignment of values to variables that satisfies certain constraints, etc.) can be associated with languages in NP; the answers one is looking for play the role of certificates.

¹⁴A standard variant, CNF-SAT, restricts the input formulas to conjunctive normal form, see Lecture 9.

Class NP in relation to classes P and EXP

Proposition 54. $P \subseteq NP \subseteq EXP$

Proof. Suppose $L \in P$, so there is a machine M that runs in polynomial time and accepts w if and only if $w \in L$. We construct a verifier V for L : given (w, s) , it ignores s and runs M on w . Take $p(n) = 0$, so the only possible certificate is the empty word. Then V accepts a pair exactly when $w \in L$, as required.

Now suppose $L \in NP$, with verifier V and a polynomial bound $p(n)$ on certificate length. Imagine the following machine M' . Given w , it considers *every* word s of length at most $p(|w|)$ and runs V on (w, s) , accepting if and only if V accepts at least one pair. It recognises L : if there is a certificate for w , it will find it in its long search. It also runs in exponential time. Let $q(n)$ be a polynomial upper bound on V 's running time. On an input of length $|w| = n$, each run takes at most $q(n + p(n)) = n^{O(1)}$ steps. Since Σ is fixed, there are at most $2^{O(p(n))} = 2^{n^{O(1)}}$ candidate certificates. The enumeration and simulation overhead is polynomial per candidate, so the total running time is still $2^{n^{O(1)}}$. Thus $L \in EXP$. $\square$

By Theorem 50, $P \subset EXP$, strictly. What about the two inclusions in Proposition 54? Are they strict? It is *widely believed* that they are, but this remains unproved.

Indeed, P is the class of problems that can be solved efficiently; NP is the class of problems for which a proposed yes-answer (a certificate) can be checked efficiently. It is counter-intuitive that being able to check something quickly might mean being able to find it quickly, when the search space can be exponentially large. Yet so far nobody has found a precise mathematical argument that rules out this possibility. This is where the theory of computation starts to feel less like mathematics and more like science: we hypothesise certain ‘laws of computation’, but have not proved them mathematically.

In 2000, the Clay Mathematics Institute selected seven long-standing, difficult problems in mathematics as the Millennium Prize Problems. Its total prize fund was US\$7million, with US\$1 million allocated for a solution to each problem; the P-versus-NP question is one of them.

Lectures 9 & 10. Reductions and NP-completeness

Complexity theory classifies computational problems into complexity classes (we have seen some: P, NP, and EXP). It can also compare the difficulty of problems and select ‘the most difficult’ in a class.

9.1 Polynomial-time reductions

Definition 55. Let $A, B \subseteq \Sigma^*$. A function $r : \Sigma^* \rightarrow \Sigma^*$ is a polynomial-time reduction from A to B if it is computable by a Turing machine in polynomial time and preserves the answer: for every $w \in \Sigma^*$,

$$w \in A \iff r(w) \in B.$$

We write $A \leq_p B$.

Example 56. Consider the following two graph-related languages:

UNDIRECTED-REACHABILITY = $\{\langle G, s, t \rangle \mid \text{there is a path from } s \text{ to } t \text{ in the undirected graph } G\}$

DIRECTED-REACHABILITY = $\{\langle H, s, t \rangle \mid \text{there is a path from } s \text{ to } t \text{ in the directed graph } H\}$

Here s and t are nodes of the given graph.

We show that $\text{UNDIRECTED-REACHABILITY} \leq_p \text{DIRECTED-REACHABILITY}$. By definition, we need to give a polynomial-time algorithm that, given any input word w , produces an output word $r(w)$ such that w encodes a triple $\langle G, s, t \rangle$ with a corresponding path if and only if $r(w)$ encodes a triple $\langle H, s, t \rangle$ with a directed path.

We describe the algorithm in words. If the input w is malformed (does not encode any graph-node-node triple), output the empty word (it does not encode any triple either). If w encodes some $\langle G, s, t \rangle$, construct H as follows. Copy all the nodes of G . For every edge $u-v$ of G , add to H two directed edges $u \rightarrow v$ and $v \rightarrow u$.

Every undirected path in G is then a directed path in H , and every directed path in H follows edges that came from edges of G . Thus the answer is preserved. The construction only copies nodes and replaces each edge by two edges. For an input encoding of length n , even copying the endpoint names by repeated scans takes at most $O(n^2)$ time, so this is a polynomial-time reduction.

Proposition 57.

1. If $A \leq_p B$ and $B \in \text{P}$, then $A \in \text{P}$.
2. If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.

Proof. For the first statement, take $k, m \geq 1$. Let r be a reduction from A to B , computed in time $O(n^k)$, and let B be decided by a machine M running in time $O(n^m)$. We can decide A in polynomial time as

follows. On input w , compute $r(w)$ and give it to M as input. The length $|r(w)|$ is $O(|w|^k)$, because the machine computing r cannot write more symbols than it makes steps. The total time is thus at most

$$O(|w|^k) + O((|w|^k)^m) = O(|w|^{km})$$

which is polynomial in the input length $|w|$. The answer is correct because $w \in A$ exactly when $r(w) \in B$.

For the second statement, again take $k, m \geq 1$. Let r_1 reduce A to B and be computable in time $O(n^k)$, and r_2 reduce B to C and be computable in time $O(n^m)$. Let $r(w) = r_2(r_1(w))$. Since $|r_1(w)| = O(|w|^k)$, the total computation time is again $O(|w|^{km})$. Finally,

$$w \in A \iff r_1(w) \in B \iff r_2(r_1(w)) \in C,$$

so r is a reduction from A to C . □

9.2 NP-hardness and NP-completeness

Reductions let us express that one problem is at least as hard as every problem in NP.

Definition 58. A language B is NP-hard if, for every language $A \in \text{NP}$, $A \leq_p B$. It is NP-complete if, additionally, $B \in \text{NP}$.

Thus, NP-complete problems are ‘the hardest’ problems in class NP (if we neglect the cost of computing polynomial-time reductions). By Proposition 57, giving a polynomial-time algorithm for any NP-complete problem (or any NP-hard problem) would prove that $P = \text{NP}$.

Indeed, such an algorithm would give a polynomial-time algorithm for every language in NP, by first applying its reduction. Thus $\text{NP} \subseteq P$, and the reverse inclusion is already known.

Reductions preserve NP-hardness:

Proposition 59. If A is NP-hard and $A \leq_p B$, then B is NP-hard.

Proof. Let L be any language in NP. Since A is NP-hard, we have $L \leq_p A$. Since $A \leq_p B$, we get $L \leq_p B$ from Proposition 57. Since this holds for every $L \in \text{NP}$, the language B is NP-hard. □

9.3 The Cook–Levin theorem

We will need some terminology on Boolean formulas.

Definition 60. A literal is either a Boolean variable x or its negation $\neg x$. A clause is a disjunction of literals: $\alpha_1 \vee \alpha_2 \vee \dots \vee \alpha_n$. A CNF (conjunctive normal form formula) is a conjunction of clauses:

$$(\alpha_{11} \vee \alpha_{12} \vee \dots \vee \alpha_{1n_1}) \wedge (\alpha_{21} \vee \alpha_{22} \vee \dots \vee \alpha_{2n_2}) \wedge \dots \wedge (\alpha_{m1} \vee \alpha_{m2} \vee \dots \vee \alpha_{mn_m})$$

where each α_{ij} is a literal (different α_{ij} may correspond to different variables!)

Consider the language:¹⁵

$$\text{CNF-SAT} = \{\varphi \mid \varphi \text{ is a satisfiable CNF}\}$$

Theorem 61 (Cook—Levin). *The language CNF-SAT is NP-complete.*

Proof. First, $\text{CNF-SAT} \in \text{NP}$, just like SAT of Lecture 8. A certificate gives a satisfying assignment, and the verifier checks whether the formula is true on this assignment. It also checks that the input is a CNF. There are at most as many variables as symbols in the input, and checking each clause takes polynomial time, so both the certificate length and verification time are polynomially bounded.

The difficult part is to prove NP-hardness. Fix any language $L \in \text{NP}$. By the definition from Lecture 8, there are a polynomial-time verifier V and a polynomial bound on certificate length, say $p_1(n)$, such that $w \in L$ exactly when V accepts (w, s) for some certificate s of length at most $p_1(|w|)$. Suppose V runs in time $p_2(n)$.

We will make several assumptions on V , without loss of generality:

1. Let Σ be the input alphabet (of words and certificates) and let $\Gamma \supseteq \Sigma \cup \{\sqcup\}$ be the tape alphabet, with $\sqcup \notin \Sigma$. The input pair is written as $w \sqcup s$.
2. Suppose V has a tape that is infinite to the right only, and starts at its leftmost cell. We have seen in Exercise 3.6(a) that this does not change the computational power of Turing machines. It is also easy to see that what could be done on a two-way infinite tape in polynomial time is still doable in polynomial time on a one-way infinite tape.
3. Furthermore, assume V accepts by halting in state q_a and rejects by halting in state q_r , and that upon halting it actually continues to work but does nothing (remains in the same state in the same position for eternity). Formally, for every $a \in \Gamma$, add the instructions $(q_a, a) \mapsto (q_a, a, S)$ and $(q_r, a) \mapsto (q_r, a, S)$.

The computation table. Fix an input word $w = a_1 \dots a_n$. A certificate $s = b_1 \dots b_m$ has length $m \leq p_1(n)$, so the initial tape is

$$a_1 \dots a_n \sqcup b_1 \dots b_m \sqcup \sqcup \dots \quad (2)$$

Recall that $n = |w|$. The input pair occupies at most $n + 1 + p_1(n)$ cells. If V takes at most $p_2(k)$ steps on an input pair of length k , then its running time on any such pair is at most

$$p(n) = p_2(n + 1 + p_1(n)).$$

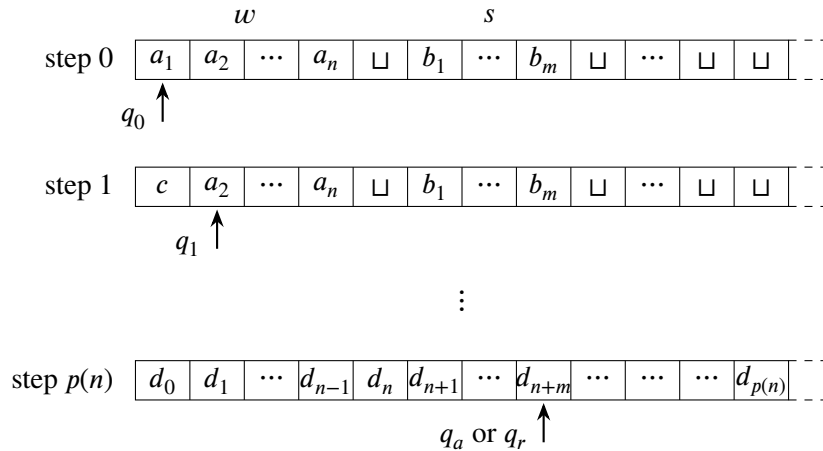
Here we take p_2 to be nondecreasing, and enlarge it if necessary so that $p_2(k) \geq k + 1$. It remains a polynomial bound. With our ‘loop-in-your-halting-state’ rule, we can record exactly $p(n)$ steps, even when V accepts or rejects earlier.

This time bound also limits how far the head can go. It starts at cell 0 and moves by at most one cell per step, so it cannot go beyond cell $p(n)$. These $p(n) + 1$ cells also contain the whole input and a following blank, by our choice of p_2 .

¹⁵In the literature, this language is usually denoted as just SAT, but we used this name for the language of all satisfiable Boolean formulas, not only CNFs. Of course, $\text{CNF-SAT} \subset \text{SAT}$.

Now write the successive tape snapshots one below another. They form a table: row 0 is the initial tape, row 1 is the tape after one step, and so on up to row $p(n)$. Each row has columns $0, \dots, p(n)$, so the table has $(p(n) + 1) \times (p(n) + 1)$ cells. To describe the computation fully, also mark the head position and the current state in each row.

For example, suppose $n \geq 2$ and the first instruction is $(q_0, a_1) \mapsto (q_1, c, R)$. It replaces a_1 by c , moves the head to a_2 , and enters state q_1 . All other cells remain unchanged:



In the final row, d_j is the symbol left in cell j after $p(n)$ steps; the state is q_a or q_r . The first transition above is only an illustration. The construction below uses the actual instructions of V .

Given w , we can describe how a correct table must start and how each row must follow from the previous one, even though we do not know the certificate. We will express these requirements by a CNF φ_w , and require the final state to be accepting. More precisely, we will have Boolean variables for the cells in the table, for the head position, and for the machine's state. An assignment satisfying the clauses below will describe a table of size $(p(n) + 1) \times (p(n) + 1)$, together with its head positions and states. The clauses of φ_w will put constraints on this assignment, so that φ_w is true on it if and only if it describes a correct table: one which starts from the configuration (2), has an accepting state in the final row, and in which each transition from one row to the next follows an instruction of V .

Thus, our goal is

$$w \in L \iff \varphi_w \text{ has a satisfying assignment}$$

We must also show that the formula can be constructed in polynomial time. Then $w \mapsto \varphi_w$ will be the required reduction.

Encoding the table by Boolean variables. We will use the following Boolean variables:

- x_{ija} for every cell (i, j) and every $a \in \Gamma$: $x_{ija} = T$ if and only if that cell contains a . Here $i, j \in \{0, \dots, p(n)\}$.
- y_{ij} , also for every cell: $y_{ij} = T$ if and only if V was looking at the j th cell after step i .
- z_{iq} , for every step i and every state q of V : $z_{iq} = T$ if and only if V was in state q after step i .

Here, 'after step 0' means 'at the start'.

We build the formula in pieces. Each piece is a conjunction of clauses, so the final formula will be in conjunctive normal form (CNF).

One symbol in every cell. For each cell (i, j) , add the clauses

$$\bigvee_{a \in \Gamma} x_{ija} \quad \text{and} \quad (\neg x_{ija} \vee \neg x_{ijb}) \quad \text{for all distinct } a, b \in \Gamma$$

The first requires at least one symbol, and the others forbid two different symbols. Thus, every cell contains exactly one symbol.

The initial row. We fix $w \sqcup$, reserve $p_1(n)$ cells for the certificate, and fill the rest with blanks:

$$\left(\bigwedge_{j=0}^{n-1} x_{0,j,a_{j+1}} \right) \wedge x_{0,n,\sqcup} \wedge \left(\bigwedge_{j=n+1+p_1(n)}^{p(n)} x_{0,j,\sqcup} \right) \wedge y_{00} \wedge z_{0q_0}$$

The last two clauses fix the initial head position and state.

In the certificate cells, for every $n+1 \leq j \leq n+p_1(n)$, add

$$\left(x_{0,j,\sqcup} \vee \bigvee_{a \in \Sigma} x_{0ja} \right) \quad \text{and} \quad (\neg x_{0,j,\sqcup} \vee x_{0,j+1,\sqcup})$$

These cells contain letters from Σ or blanks, and after the first blank all following cells are blank. Thus, the initial tape contains $w \sqcup s$ for some $s \in \Sigma^*$ of length at most $p_1(n)$. We do not know the certificate; we only want it to exist—in the end, it is chosen by the satisfying assignment.

One head and one state in every row. Let Q be the set of states of V . For every row $i \in \{0, \dots, p(n)\}$, add the clauses

$$\bigvee_{j=0}^{p(n)} y_{ij} \quad \text{and} \quad \bigvee_{q \in Q} z_{iq}$$

They say that the head is somewhere and the machine is in some state. We also need to rule out two head positions or two states at once. For every pair of distinct positions j, k , and every pair of distinct states q, q' , add

$$(\neg y_{ij} \vee \neg y_{ik}) \quad \text{and} \quad (\neg z_{iq} \vee \neg z_{iq'})$$

Together, these clauses say that every row has exactly one head position and exactly one state.

The final row. We want the computation to accept, so add the clause

$$z_{p(n),q_a}$$

Recall that once V accepts, it stays in state q_a . Thus, acceptance earlier than step $p(n)$ also gives an accepting final row.

Correct transitions. We now connect consecutive rows. Consider an instruction

$$(q, a) \mapsto (q', b, d),$$

where $a, b \in \Gamma$ and $d \in \{-1, 0, 1\}$ describes the head movement. If the machine is in state q , looking at cell j , and that cell contains a , then in the next row it must be in state q' , looking at cell $j + d$, with b written in cell j .¹⁶

For every $i < p(n)$, every cell j , and every instruction, we require

$$(z_{iq} \wedge y_{ij} \wedge x_{ija}) \implies (z_{i+1,q'} \wedge y_{i+1,j+d} \wedge x_{i+1,j,b}).$$

The implication is equivalent to three clauses, one for each required change:

$$(\neg z_{iq} \vee \neg y_{ij} \vee \neg x_{ija} \vee z_{i+1,q'}) \wedge (\neg z_{iq} \vee \neg y_{ij} \vee \neg x_{ija} \vee y_{i+1,j+d}) \wedge (\neg z_{iq} \vee \neg y_{ij} \vee \neg x_{ija} \vee x_{i+1,j,b})$$

If the machine is in state q , at cell j , reading a , the first three literals in each clause are false. The clauses then force the next state, head position, and written symbol. For every other instruction the condition is false, so its clauses impose no restriction. This is why we can include clauses for all instructions without knowing which will be used.

For $j = 0$ and $d = -1$, use $y_{i+1,0}$, since a move left at the boundary leaves the head in place. If a rule would move beyond the rightmost column, forbid its condition by the clause $\neg z_{iq} \vee \neg y_{ij} \vee \neg x_{ija}$. This cannot exclude a real computation: starting at cell 0, the head cannot move beyond cell $p(n)$ in $p(n)$ steps.

There is one more requirement: all cells other than the one under the head must keep their symbols. For every $i < p(n)$, every j , and every $a \in \Gamma$, require

$$\neg y_{ij} \implies (x_{i+1,j,a} \leftrightarrow x_{ija}).$$

When y_{ij} is false, this requires every symbol variable for cell j to keep its value. When the head is at j , the transition rule instead determines the new symbol. In CNF, the requirement is

$$(y_{ij} \vee \neg x_{i+1,j,a} \vee x_{ija}) \wedge (y_{ij} \vee x_{i+1,j,a} \vee \neg x_{ija}).$$

Now every part of the next configuration is determined: the instruction fixes the next state, the head position, and the symbol written, while these last clauses preserve all the other symbols.

Let φ_w be the conjunction of all the clauses we have described.

Why is the construction polynomial-time? There are $(p(n) + 1)^2$ cells in the table and fixed numbers of symbols in Γ and states in Q . Thus, there are $O((p(n) + 1)^2)$ variables. The largest family of clauses is the one forbidding two head positions: for each row, we consider every pair of columns, giving $O((p(n) + 1)^3)$ clauses. All the other families have at most this many literal occurrences in total. The indices take $O(\log(p(n) + 2))$ bits to write.

The algorithm simply runs through the rows, columns, pairs of columns, and instructions, writing the corresponding clauses. For an explicit polynomial bound, choose constants $c, k \geq 1$ such that $p(n) + 1 \leq c(n+1)^k$. Even allowing $O(p(n)+1)$ time to write each index, the construction takes $O((n+1)^{4k})$ elementary operations. The loop counters and indices need only polynomially many tape cells; updating them by tape scans therefore also takes polynomial time per clause. Thus, $w \mapsto \varphi_w$ is computable in polynomial time.

¹⁶We denoted the direction by L , S or R before. Since here we use numerical indices of cells, -1 , 0 and 1 are more convenient.

Why is the answer preserved? If $w \in L$, take a certificate s on which V accepts. Fill the table with the computation on (w, s) , continuing in the accepting state until the last row. Assign the variables according to the tape symbols, head positions, and states in this table. The initial row has the required encoding, every cell has one symbol, every row has one head and one state, every transition is correct, and the final state is q_a . Thus, all clauses of φ_w are true.

Conversely, suppose an assignment satisfies φ_w . ‘Read’ the table, head positions, and states from this assignment. The symbol clauses give one symbol per cell, and the initial clauses make the first row $w \sqcup s$ followed by blanks, for some $s \in \Sigma^*$ of length at most $p_1(n)$. The head and state clauses give one head and one state in each row. Starting from row 0, the transition clauses force each successive row to describe exactly the next configuration of V . Finally, the last row has state q_a , so this is an accepting computation. Thus, V accepts (w, s) , and hence $w \in L$.

In short,

$$w \in L \iff \text{there is an accepting computation table for } w \text{ and some } s \iff \varphi_w \text{ is satisfiable.}$$

We have constructed a polynomial-time reduction from an arbitrary language $L \in \text{NP}$ to CNF-SAT. Therefore, CNF-SAT is NP-hard and, since it also belongs to NP, it is NP-complete. $\square$

Corollary 62. SAT is NP-complete.

Proof. In Lecture 8, we defined SAT as the language of arbitrary satisfiable Boolean formulas, and proved $\text{SAT} \in \text{NP}$. To reduce CNF-SAT to SAT, output the input unchanged if it is a CNF, and output F otherwise. This check takes polynomial time. We need the check because a satisfiable non-CNF formula belongs to SAT but not to CNF-SAT. Thus, the reduction preserves the answer on every input, proving $\text{CNF-SAT} \leq_p \text{SAT}$. By Proposition 59, SAT is NP-hard and hence NP-complete. $\square$

9.4 The NP-completeness of CLIQUE

Theorem 63. The language CLIQUE is NP-complete.

Proof. We already know from Lecture 8 that $\text{CLIQUE} \in \text{NP}$: a certificate lists the nodes of a clique, and the verifier checks that they are distinct and pairwise adjacent. It remains to prove NP-hardness. By Theorem 61 and Proposition 59, it is enough to show that

$$\text{CNF-SAT} \leq_p \text{CLIQUE}.$$

We need a polynomial-time algorithm that, given a CNF φ , constructs a graph G_φ and a number k such that φ is satisfiable if and only if G_φ has a clique of size at least k .

Suppose $\varphi = C_1 \wedge \cdots \wedge C_m$, where each C_i is a clause—a disjunction of literals. To satisfy φ , we need to make at least one literal in each clause true. These choices must agree: we cannot choose both x and $\neg x$. The graph will express exactly this requirement.

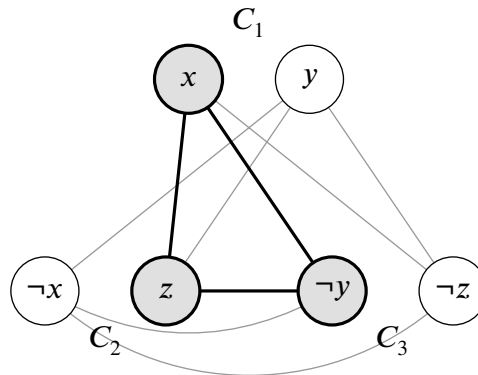
Create one node for each occurrence of a literal in φ . Thus, even if the same literal occurs in several clauses, these occurrences give different nodes. Join two nodes by an edge if they come from different

clauses and their literals are not negations of each other. Finally, set $k = m$, the number of clauses. This defines the output $\langle G_\varphi, m \rangle$.

For a small example, take

$$\varphi = \underbrace{(x \vee y)}_{C_1} \wedge \underbrace{(\neg x \vee z)}_{C_2} \wedge \underbrace{(\neg y \vee \neg z)}_{C_3}.$$

There are six literal occurrences, so the graph has six nodes. The target clique size is 3.



The highlighted nodes x , z , and $\neg y$ form a clique, with one node from each clause. They give the assignment $x = T$, $y = F$, $z = T$, which satisfies φ . There is no edge, for example, between x and $\neg x$, or between the two nodes of C_1 .

Let us prove that the construction works for every CNF. If φ is satisfiable, choose one true literal occurrence from each clause. These m nodes come from different clauses, and no two are labelled by a variable and its negation: both could not be true under the same assignment. Hence every pair is joined by an edge, giving a clique of size m .

Conversely, suppose G_φ has a clique of size at least m . No two nodes from the same clause are adjacent, so the clique has at most one node from each clause. It must therefore have exactly one from each of the m clauses. Their literals do not contradict each other. Make them all true: set a variable to T if a selected literal is positive and to F if it is negative; assign all other variables arbitrarily. Every clause now has a true literal, so φ is satisfied.

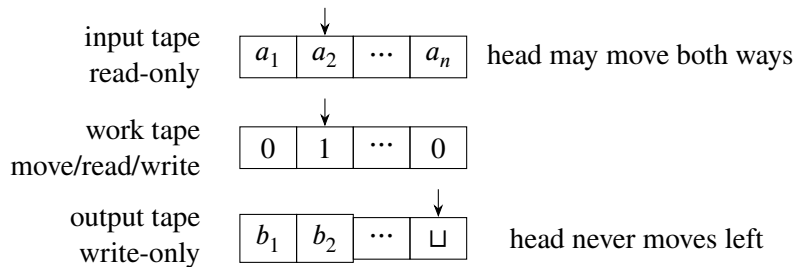
Finally, the construction takes polynomial time. (This is usually left in such proofs as ‘easy to see’—make sure you have an intuitive understanding of what a polynomial-time algorithm, say in Python, would look like.)

For completeness, malformed inputs are sent to a fixed no-instance: a single node with target size 2. An empty clause also gives a no-instance, since no node can be chosen from it. If there are no clauses, the formula is constant T , so output a fixed yes-instance: a single node with target size 1. $\square$

Lecture 11. Space complexity

So far, we have measured the time needed to solve a problem. Today we ask a different question: how much working memory does a computation need?

We use Turing machines with three tapes: a read-only input tape; a work tape, on which the machine can read and write, as usual; and a write-only output tape, whose head can move right but never left.



Definition 64. A Turing machine uses space $f(n)$ if on every input w it visits at most $O(f(|w|))^{17}$ cells on its work tape. A language is decidable in space $f(n)$ if some machine deciding it uses that much space.

We define

$$\text{SPACE}(f(n)) = \{L \subseteq \Sigma^* \mid L \text{ is decidable in space } f(n)\}$$

and

$$\text{PSPACE} = \bigcup_{k \geq 0} \text{SPACE}(n^k)$$

Let us relate PSPACE to the time classes we know.

Proposition 65.

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXP}.$$

Proof. We already know from Lecture 8 that $\text{P} \subseteq \text{NP}$. A polynomial-time machine can visit only polynomially many work-tape cells, so $\text{P} \subseteq \text{PSPACE}$.

Now let $A \in \text{NP}$, with verifier V and a polynomial bound $p(n)$ on certificate length. Imagine a machine that tries all certificates of length at most $p(n)$, one after another, and accepts if one is accepted by V . It stores only the current certificate (of length polynomial in n) and provides the workspace for one run of V (which runs in polynomial time and thus uses polynomial space). After that run, it reuses the same space for the next. So $A \in \text{PSPACE}$.

Finally, suppose A is decided by a machine M with state set Q , work-tape alphabet Γ , and a polynomial bound $p(n)$ on the number of work-tape cells it uses. Recall the notion of *configuration*: the input and work-tape head positions, the work-tape contents, and the current state. The output tape is write-only, so it cannot affect future moves. On an input w of length n , our machine has $|Q|$ possible states, at most

¹⁷As usual, the notation $O(f(n))$ means ‘ $f(n)$ up to a constant factor’. Here, this factor may depend on the machine but not on the input word.

$n + 2$ possible input-head positions, at most $p(|w|)$ possible work-tape head positions, and at most $|\Gamma|^{p(|w|)}$ possible work-tape contents. Thus it has at most

$$|Q| \cdot (n + 2) \cdot p(|w|) \cdot |\Gamma|^{p(|w|)} = O(2^{n^m})$$

configurations (for some $m \in \mathbb{N}$). Since M decides A , it halts on every input. It cannot repeat a configuration, since after a repetition it would repeat the same computation forever. Therefore it runs in time $O(2^{n^m})$, so $A \in \text{EXP}$. $\square$

The inclusions in Proposition 65 are conjectured but not known to be strict. For example, it is open whether $P = \text{PSPACE}$. The only strict inclusion here is $P \subset \text{EXP}$, by Theorem 50, the time hierarchy theorem.

By the way, there is a hierarchy also for space classes. Let $\text{EXPSPACE} = \bigcup_{k \geq 0} \text{SPACE}(2^{n^k})$.

Theorem 66. *For all $k \geq 0$, $\text{SPACE}(n^k) \subset \text{SPACE}(n^{k+1})$. Moreover, $\text{PSPACE} \subset \text{EXPSPACE}$.*

The proof of Theorem 66 essentially repeats that of Theorem 50—given enough space, one can simulate and trick every machine that uses considerably less space.

Logarithmic space

Polynomial space can still be *a lot*. As we have seen, all the NP-complete problems belong to PSPACE, and these are regarded as hard problems. In fact, many more computational tasks which are considered even harder, such as temporal planning, are doable in polynomial space. If we want, analogously to class P, to have a class of ‘tractable’ problems, in terms of space, we should set a lower limit. Say, logarithmic:

$$L = \text{SPACE}(\log n)$$

An input word w already takes $|w|$ cells on the read-only input tape; the space bound counts only the cells visited on the work tape. The output tape is useful when a computation must produce a string, for example in a reduction, but due to the restrictions it cannot serve as extra workspace.

The following proposition is proved analogously to the part $\text{PSPACE} \subseteq \text{EXP}$ in Proposition 65.

Proposition 67. $L \subseteq P$.

We do not know whether logarithmic space suffices for every problem in P.

Let $A, B \subseteq \Sigma^*$. A function $r: \Sigma^* \rightarrow \Sigma^*$ is a *logspace reduction* from A to B if it is computed using $O(\log n)$ space, and for every word $w \in \Sigma^*$:

$$w \in A \iff r(w) \in B$$

We write $A \leq_{\log} B$.

Nondeterministic computations

We will now consider a notion which has no direct correspondence in real computing machinery, but which is useful (sometimes) for theoretical analysis.

Consider the problem from Lecture 9:

$$\text{DIRECTED-REACHABILITY} = \{ \langle G, s, t \rangle \mid \text{there is a directed path from } s \text{ to } t \text{ in } G \}.$$

Imagine yourself walking around the graph of m nodes, starting at s , trying to check if there is a path to t . If you have reached t , you can accept immediately. Otherwise, you have a choice of where to go next, out of the potentially many neighbours of the current node. If you have no further information on the graph, you may try to guess which direction to choose. In this procedure, the following characterisation is important: if a path from s to t exists, you *may* get lucky and reach it *within* m steps; if there is no path, you *will never* reach t , however lucky you are. So one may say: $\langle G, s, t \rangle \in \text{DIRECTED-REACHABILITY}$ if one can guess the path from s to t within n steps, where n is the size of input (so $n \geq m$).

Formally, a *nondeterministic Turing machine* may have several possible next moves:

$$\begin{aligned} (q, a) &\mapsto (q_1, b_1, R) \\ (q, a) &\mapsto (q_2, b_2, L) \\ &\dots \\ (q, a) &\mapsto (q_3, b_3, S) \end{aligned}$$

Each possible move creates a branch of the computation. To decide a language, every branch must halt, and acceptance is defined by the existence of an accepting branch.

Definition 68. A language is in $\text{NSPACE}(f(n))$ if some nondeterministic Turing machine decides it using $O(f(n))$ space on every branch. The class $\text{NTIME}(f(n))$ is defined in the same way, with a bound of $O(f(n))$ steps on every branch. We set

$$\text{NL} = \text{NSPACE}(\log n) \quad \text{and} \quad \text{NP} = \bigcup_{k \geq 0} \text{NTIME}(n^k).$$

The latter gives an alternative equivalent definition of NP (see Exercise 11.6). For example, a nondeterministic machine can guess a truth assignment for SAT or a set of nodes for CLIQUE, then check the guess in polynomial time.

The reachability machine uses logarithmic space: it stores its current node, the next guessed node, and a counter up to $|V(G)|$. Each of these takes $O(\log n)$ bits, where n is the input length. It can check whether a proposed move follows an edge by scanning the input graph. Thus $\text{DIRECTED-REACHABILITY} \in \text{NL}$.

Since a standard Turing machine as we know it (called *deterministic*) is a special case of a nondeterministic machine limited to a single choice for each transition, a deterministic complexity class is always a subclass of its nondeterministic counterpart: $\text{P} \subseteq \text{NP}$, $\text{L} \subseteq \text{NL}$.