Prelude

Francis Lazarus

September 22, 2020

The first part of this course deals with graphs and surfaces from the combinatorial viewpoint. Graphs are ubiquitous in mathematics and computer science. At a very crude level a graph simply records relationships between objects, in the most general sense of connections. Viewing each relation as a wire connecting the associated objects, one may endow a graph with a topological structure, namely a one-dimensional complex. This one-dimensional complex can further be embedded on a topological surface, yielding the notion of combinatorial map. This very rich theory culminates with the Groethendieck's *dessins d'enfants*. At a more pragmatic scale, combinatorial maps allows to develop most of the theory of topological surfaces in a simplified framework. Graphs are also intimately related to groups in several ways. First, we have the usual homotopy and homology functors that apply to graphs as topological objects. Studying coverings of graphs and surfaces through this lens happens to be already fruitful. Maybe more importantly, groups act on graphs, among which their Cayley graphs, and such actions can provide useful information on the groups. Free groups are for instance characterized by the fact they act freely on a tree.

Finally, we can further enrich graphs with an obvious metric structure. The geometric properties of the Cayley graphs of a group may then provide information on the group itself. This is the subject of the second part of this course...

This introductory first part is thus the opportunity to introduce basic concepts in topology and in combinatorial group theory. The combinatorial point of view often allows to provide simple proofs of topological results that would otherwise require more advance concepts. The main reason is that working with the most general topological setting, one may encounter wild objects such as fractals or other monstrous constructs. Such bad encounters do not happen in the combinatorial world. One might complain that this simplification has a cost: the loss of some generality. However, the objection is not so relevant when topology is just an intermediate tool to study objects that are fundamentally discrete such as finitely generated groups. Moreover, manipulating combinatorial objects naturally leads to a constructive point of view and often leads to the design of algorithms. This algorithmic aspect opens the door to experimentation, one of the opportunities for mathematicians to benefit from the progresses in computer science.

Contents

1 What is an Algorithm

This first lecture is thus devoted to the basic notions of algorithm and computation. Concerning the theory of computation and complexity, I recommend the surveys of Avi Widgerson [Wig06] and Peter van Emde Boas [vEB90], and the book by Christos Papadimitriou [Pap94]. Concerning the fundamentals of algorithms I recommend the thin book of Robert Tarjan [Tar83] which is concise but rather restricted to graph algorithms, and the thick book of Cormen et al. [CLRS09] which is more recent and complete but also quite verbose.

Intuitively, an algorithm is a recipe that one should follow to perform a particular computation, such as computing the *gcd* of two numbers or deciding if a given knot diagram represents the unknot. The recipe should not assume any knowledge or initiative from its performer so that it could actually be performed by some mechanical process. This naturally leads to the question of *what* can be computed by a mechanical process. Of course, this looks too vague to receive a definite answer. However, the question became fundamental for logicians and mathematicians, especially after they realized

that performing a computation or writing down the proof of a theorem using the very basic rules of logic was essentially the same thing. The famous *Entscheidungsproblem* posed by David Hilbert in 1928 was asking whether every statement could be proved or disproved by a *systematic* application of logical rules. A couple years later, Kurt Gödel provided a negative answer to the fact that every statement could be proved or disproved, without even taking into account the systematic (or algorithmic) aspect of the question. It was not long after, in 1936, that Alonzo Church [Chu36] and Alan Turing [Tur36], independently came up with a universal notion of computability. They showed that there exists computational tasks for which no algorithm can exist, among which proving theorems, and thus answering negatively to the Entscheidungsproblem.

Church's model of computation is based on λ-*calculus*, a formal language to which apply formal reduction operations. A succession of reductions intuitively amounts to a computation. When interpreted adequately, Church [Chu36] proved that λ-calculus allows to compute the set of *partial recursive functions* formally defined by Gödel and Herbrand. Those functions, also called μ-recursive, or general recursive functions, correspond to the intuitive notion of computable functions. They include basic functions, namely the constant zero function as well as the successor and coordinate (projection) functions, and are globally stable by compositions and two constructive processes including recursion and some kind of minimization (whence the prefix μ). Turing [Tur36] proved that his machines are able to compute the same set of functions as Church's λ-calculus. In the end, all those elaborate models of computation turned out to be equivalent. This lead to what is known as the **Church-Turing thesis** expressing that anything that can be computed by a mechanical process can actually be computed by one of those equivalent formal models.

Turing machines and the related Random Access Machines (RAM) happen to be more intuitive and more amenable to the notion of complexity. For this reason, we shall restrict ourselves to these models. For us, the notion of algorithm can thus be identified with that of a Turing or a RAM machine.

2 Turing Machines and Decision Problems

2.1 Turing machines

In his foundational paper [Tur36] Turing justifies his model of computation by an intuitive argumentation that *naturally* leads to his machine. Since intuition is a rather relative concept it is not so surprising that the formal definition of a Turing machine enjoys (slight) variations in the literature. In accordance with the Church-Turing thesis all those definitions however lead to the same notion of computability. We will use the following definition.

Definition 2.1. A **Turing machine** is a triple $(\mathscr{A}, \mathscr{Q}, \mathscr{T})$, where $\mathscr{A}$ is a finite alphabet including a special **blank** character denoted by $\emptyset$, $\mathscr{Q}$ is a finite set of **states**, and $\mathscr{T} \subseteq \mathscr{A} \times \mathscr{Q} \times \mathscr{A} \times \mathscr{Q} \times \{L, S, R\}$ is a **transition table** specifying how the machine operates on **configurations**. Here, L, S, R are reserved symbols standing respectively for *Left, Stay, Right* and configurations are words of the form $u\,q\,v \in \mathscr{A}^* \times \mathscr{Q} \times \mathscr{A}^*$, where $\mathscr{A}^*$ denotes the set of words (*i.e.*, finite sequences) over $\mathscr{A}$.

Intuitively, the machine can be represented as on Figure 1 by a linear **tape** composed of a bi-infinite sequence of **cells** that each contains one alphabet symbol, and by a read/write head pointing to one cell and containing the machine state. Configuration uqv then corresponds to a tape marked with the word uv and otherwise with blanks and whose read/write head points to the first letter in v (the empty word is interpreted as a blank). Transition $aqbpD \in \mathcal{T}$ applies to any configuration uqv such that a is the first letter in v. It transforms uqv replacing a with b, the state q by p, and moves the head one step to the left or right according to whether D equals L or R, respectively. The head stays where it is when $D = S$. A Turing machine is

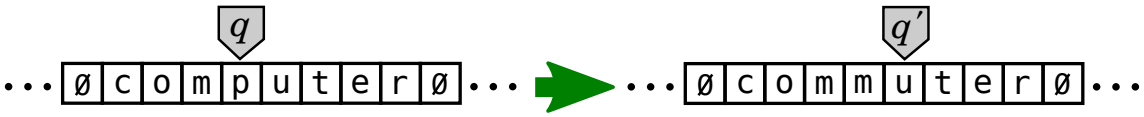

Figure 1: Illustration of the transition pqm$q'R$ applied to configuration com$q$puter on a Turing machine operating on the Latin alphabet.

deterministic if at most one transition applies to a given configuration: $aqbpD \in \mathcal{T}$ and $aqb'p'D' \in \mathcal{T}$ implies $b' = b$, $p' = p$ and $D' = D$. As opposed to a deterministic machine, a nondeterministic Turing machine may lead to several computations starting from a same configuration. The machine is **halting** in a given configuration when no transition applies. Given an initial configuration, we can inductively apply the transitions of a Turing machine. The machine may halt or run forever. The content of the tape at the initial configuration is the **input** of the machine. A same machine may halt starting with some inputs and run forever for some other inputs.

2.2 Problems and decidability

In full generality a **decision problem** is defined by a certain property that should be tested on the set of words $\mathcal{A}^*$ over a fixed alphabet $\mathcal{A}$. The words satisfying the property are the *YES instances* of the problem. Usually, the YES instances correspond to the encoding of objects – such as numbers, graphs, knot diagrams or group presentations – satisfying a certain property. For instance, one may consider the problem of primality testing where the YES instances are the binary encoding of prime integers over the alphabet $\mathcal{A} = \{0,1\}$. A decision problem can thus be identified with the **language** of its YES instances. Conversely, any language, *i.e.* any subset of $\mathcal{A}^*$, can be considered as the set of YES instances of a decision problem. We will thus write $w \in I$ for a YES instance w of a problem denoted by I.

A decision problem I is **decidable**, or (effectively) **solvable**, or **recursive**, if there exists a turing machine $M = (\mathcal{A}, \mathcal{Q}, \mathcal{T})$ with three states $q_i, q_a, q_r \in \mathcal{Q}$, respectively called **initial, accepting** and **rejecting**, such that for every $w \in \mathcal{A}^*$ the machine M starting from configuration $q_i w$ reaches a halting configuration in state q_a if $w \in I$ and in state q_r otherwise. In particular M always reaches a halting configuration. In this case, M is said to solve or decide[1] problem I. An *algorithm* for problem I is just another name for a Turing machine solving I.

[1] or, referring to the language terminology, to *recognize*.

A decision problem I is **semi-decidable** if there exists a Turing machine halting in an accepting state if and only if it is given as input a word of I. Although this definition does not require any behavior for words not in I, it is equivalent to assume that the machine never stops given such words. Note that I is decidable if and only if both I and its complement $\mathscr{A}^* \setminus I$ are semi-decidable. Unfortunately, the same definition of semi-decidable was given many names such as **semi-recursive**, **recursively enumerable**, **computably enumerable**, **listable** or **Turing recognizable**. Some of the names comes from the fact that semi-decidability is equivalent to require the existence of a Turing machine that enumerates I, *i.e.*, outputs all its words one after the other.

2.3 The halting problem

2.3.1 Standard coding of Turing machines

A Turing machine M is in **standard form** if its alphabet is a finite subset of $\{\text{blank}, 1, 1', 1'', 1''', \ldots\}$ and its set of states is a finite subset of $\{q, q', q'', q''', \ldots\}$. Up to a bijection between the sets of states and the alphabets, any Turing machine has an equivalent standard form. One can encode the transition table of a standard form M on the seven letter alphabet $\{\text{blank}, 1, q, ', R, S, L\}$ by simply concatenating its transitions $1^{(i)}q^{(j)}1^{(k)}q^{(\ell)}D$, where each of the $i+j+k+\ell$ prime symbols is considered as a separate symbol. Finally, replacing $q, ', R, S, L$ by the respective letters $1', 1'', 1''', 1'''', 1'''''$, we obtain a coding of the transition table over the finite alphabet $\{\text{blank}, 1, 1', 1'', 1''', 1'''', 1'''''\}$. This coding is the **standard code** of M and is denoted by $\lceil M \rceil$. Other definitions of standard codes are possible. Most importantly the alphabet used for the encoding is finite, independent of M, and is a subset of the alphabet of M (after enlarging its alphabet if necessary).

2.3.2 Undecidability of the halting problem

Consider the **self-halting problem** of deciding if a Turing machine M given as input its own standard code, *i.e.* starting with the configuration $q_i\lceil M \rceil$, will eventually reach a halting configuration in the accepting state.

Theorem 2.2. *The self-halting problem is semi-decidable but not decidable.*

PROOF. That the self-halting problem is semi-decidable is quite clear. Given the standard code of a Turing machine, it is enough to simulate the corresponding machine on this same input. The notion of universal Turing machine (see below) provides such a simulation. By way of contradiction, suppose that the self-halting problem is decidable. Hence, there exists a Turing machine, say S, that recognizes the complementary language. In other words, S halts in the accepting case if the input *does not* correspond to the standard code of a Turing machine that halts in the accepting state on its own input, and runs forever otherwise. Let us run S with the initial configuration $q_i\lceil S \rceil$. If S halts in the accepting state, this means that S does not halt in the accepting state on its own input, a contradiction. So S must run forever, meaning that S does halt in the accepting state on its own input, and we have again reached a contradiction. □

The general **halting problem** is to decide, given a machine M and a starting configuration if M reaches a halting configuration. Since the self-halting problem is a particular case of the halting problem, we obtain:

Corollary 2.3. *The halting problem is unsolvable.*

2.3.3 Universal Turing machine

A Turing machine T is said **universal** if for any Turing machine M and any initial configuration C, starting from configuration $q_i[M]C$ the machine T simulates the computation of M from C and halts in its accepting state if and only if this computation eventually stops. Though fastidious, one can write a program in his favourite language, say in C++, to simulate a universal Turing machine. This proves a fortiori its existence. Basically, T operates by first traversing the initial configuration C to "read" its state and the current symbol (the one that should lie under the reading head of M). Then, T needs to traverse $[M]$ in order to find the transition that applies. This transition transforms C into a configuration C' and we obtain the configuration $q_i[M]C'$ on T. The universal machine T proceeds this way until some configuration $q_i[M]C''$ is reached, where C'' is a halting configuration for M. In this case, T should stop in its accepting state. Otherwise, T runs forever.

Theorem 2.4. *The halting problem for the universal machine T is unsolvable.*

In other words, there is no Turing machine that can decide for any initial configuration if T eventually stops starting from this configuration. Indeed, such a Turing machine would solve the general halting problem by considering initial configurations of the form $q_i[M]C$.

3 Tractability and Complexity

By acknowledging a universal notion of algorithm, the Church-Turing thesis answers to the question of what is computable and more precisely of what is decidable. Quite often, a mathematician will be satisfied as soon as he knows whether a problem, such as the word problem in combinatorial group theory, is decidable or not. Computer scientists have developed a theory of computational complexity that explores with a quantitative perspective the *tractability* of an algorithm. The very famous $P \neq NP$? question is a byproduct of this theory.

The **time complexity** of a computation of a Turing machine starting with a given input is the number of transitions that apply until the machine halts. Similarly, the **space complexity** is the maximal length of a configuration, *i.e.*, the total number of cells scanned at least once during this computation[2]. When the machine is deterministic, the input can be associated with the complexity of the unique computation it triggers. For nondeterministic machines, one should be more careful. See the section

[2]There is a little ambiguity here, on how to count the input cells. One should be more specific on that point in practice.

below on the class NP. We say that a Turing machine, or an algorithm, has time complexity $f : \mathbb{N} \to \mathbb{N}$ if $f(n)$ is an upperbound for the time complexity of any computation starting with an input of length n, that is occupying n cells of the tape in the initial configuration. An analogous definition holds for the space complexity of the machine. Sometimes, in order to define complexities that are less than linear, one excludes the time and space for reading the input during the computations.

A decision problem has time or space complexity f if it can be solved by a Turing machine with time or space complexity f. For efficiency reasons one usually extends the Turing machine model to multitape models when studying complexity of problems. See Section 3.1 for the description of multitape turing machines. However, a phenomenon known as *linear speedup* appears with these multitape models: If a problem can be solved by a (multitape) Turing machine with time complexity f, then it can be solved in time $\varepsilon f(n)+n+2$ by another multitape Turing machine obtained by enlarging the alphabet and possibly adding one more tape. For this reason, the time complexity is often defined up to a multiplicative factor. Hence, with the usual Landau notations (Big O, little o,...), it is common practice to write that the time complexity of an algorithm *is* $f(n)$ or $O(f(n))$ when we really mean that there exists a multitape Turing machine solving the problem whose time complexity belongs to the class of functions $O(f)$.

The time complexity of a problem is thus intimately related to the multitape Turing machine model. However, this dependency is weaker than it seems at first sight. Indeed, an extension of the Church-Turing thesis states that the time complexity of a problem using any *reasonable*[3] model of computation is equivalent to the complexity for Turing machines within a polynomial. In other words, a problem of time complexity f with respect to Turing machines has time complexity $p \circ f$ for any other reasonable model of computation, where p is a polynomial function depending on the computation model only. Reciprocally, a problem of time complexity f with respect to a reasonable model of computation has time complexity $p \circ f$ on Turing machines. In particular, a problem has polynomial time complexity for Turing machines if and only if it has polynomial time complexity in any other reasonable model of computation.

It is thus legitimate to define a **tractable** problem, or algorithm, as one with polynomial time complexity. This is coherent with the fact that a problem, say with exponential time complexity, can not be solved in practice even for small inputs as it would take too long for a human being to wait for the end of the computation. This definition can however be too strict in practice. A polynomial time complexity with a high degree polynomial, where high is often no larger than 2, is also unfeasible in practice. Yet, a finer notion of tractability would have to deal with the problem that algorithms of linear or quadratic time complexity are irremediably attached to the precise model of computation considered.

3.1 Multitape Turing machines

In order to support the thesis of a universal notion of polynomial time complexity we investigate two other models of computations and show that they are indeed

[3] The notion of reasonable model is quite intuitive but should not be underestimated as it is easy to design unreasonably powerful model that operates drastically faster than Turing machines.

equivalent to Turing machines up to a polynomial cost. The first model, the multitape Turing machine, allows for some kind of parallelism. The second model, the random access machine, is studied in Section 3.2. A **k-tape Turing machine** has k tapes instead of a single one, and one read/write head per tape. It is otherwise working similarly to a (single tape) Turing machine. It is defined by a triple $(\mathscr{A}, \mathscr{Q}, \mathscr{T})$ with an alphabet $\mathscr{A}$, a set of states $\mathscr{Q}$, exactly as for a Turing machine, and a transition table $\mathscr{T} \subseteq \mathscr{A}^k \times \mathscr{Q} \times \mathscr{A}^k \times \mathscr{Q} \times \{R, L, S\}^k$. A configuration is defined by the current state $q \in \mathscr{Q}$ and the content of the k tapes with a marker to indicate the position of the head on each tape. The k heads point to k letters, *i.e.*, to an element of $\mathscr{A}^k$. Transition $XqYpD \in \mathscr{T}$, with $X, Y \in \mathscr{A}^k$ and $D \in \{R, L, S\}^k$, applies to any configuration whose state is q and whose heads point to X. It replaces the pointed cells by the letters in Y, moves head i, $1 \leq i \leq k$, according to the ith symbol in D and changes the state to p. Note that Turing machines as defined in Section 2.1 are the same as a 1-tape Turing machines.

By breaking the word written on a tape at the position of the head, a configuration can be seen as an element[4] of $\mathscr{Q} \times (\mathscr{A}^* \times \mathscr{A}^*)^k$. Consider the projection $\pi_k : \mathscr{Q} \times (\mathscr{A}^* \times \mathscr{A}^*)^k \to (\mathscr{A}^* \times \mathscr{A}^*)^k$ from the set of configurations to the set of **tape contents**, where the tape content is the collection of words on each tape together with the positions of the heads in these words. Said differently, π_k just forgets the state in a configuration.

Theorem 3.1. *For every k-tape Turing machine M_k, there exists a 1-tape Turing machine M with an injection ι of the tape contents of M_k into the tape contents of M such that for every computation using n transitions on M_k, starting and ending with configurations c_s and c_e, there is a computation with $O(n^2)$ transitions on M such that $\iota(\pi_k(c_s))$ and $\iota(\pi_k(c_e))$ are the tape contents of M at the start and end of the computation.*

PROOF. We shall simulate $M_k = (\mathscr{A}, \mathscr{Q}, \mathscr{T})$ on a single tape machine M. We view the k tapes of M_k as a table indexed by $\mathbb{Z} \times [k]$. We assume that index $(0, i)$ corresponds to the scanned cell of tape i at the start of the computation. We may further interpret the content of the k tapes as the content of a single tape with k tracks. To this end, we choose the alphabet $\mathscr{B} = (\mathscr{A} \times \{0,1\})^k$ for M. A letter in $\mathscr{B}$ is to represent a column in the $\mathbb{Z} \times [k]$ table together with k bit markers to indicate if the head of the ith tape of M_k, $1 \leq i \leq k$, is scanning the ith cell in the represented column. Every tape content of M_k thus corresponds to a word in $\mathscr{B}^*$. Putting the head of M at the first letter of this word, we obtain the injection ι.

We now explain how M simulates M_k. The states of M will be of the form $\mathscr{Q} \times \mathscr{Q}'$, where $\mathscr{Q}'$ is to be defined. Suppose that M_k is in configuration c and that a transition $t = XqYpD$ applies. In particular, M_k is in state q and its heads point to X. We consider M with the content $\iota(\pi_k(c))$ in state (q, q_i), where q_i is a chosen initial state in $\mathscr{Q}'$. In a first phase, M scans its tape from left to right in order to detect the k letters pointed by the heads of M_k. We record the currently discovered pointed letters by adding partial states $(\mathscr{A} \cup \{\circ\})^k$ to $\mathscr{Q}'$ ($\circ$ is any unused symbol to mark undiscovered letters). At the end of the scan, M is in state (q, X) and points to the right of the

[4] Formally, the content of a tape is a bi-infinite word with a finite number of nonblank letters. We can cut the tail and head of blanks to the left and to the right of the tape head to obtain a finite word that canonically represents the content of the tape.

columns containing at least one positive head marker component. In a second phase, M scans its tape from right to left, performing the necessary change of its content to replace X by Y with well positioned bit markers. Each time M scans a symbol with a positive marker it may have to move one step back to correctly modify its content. All the necessary moves are obtained by adding appropriate states to $\mathcal{Q}'$ and the corresponding transitions to the table of M. The details are left to the reader.

It remains to analyse the number of transitions used by M. Since M essentially scans its tape twice between the first and last head positions in M_k, the number of transitions of M corresponding to the $(i+1)$st transition of M_k is proportional to the maximal (index) distance m_i between the heads of M_k in its ith configuration. By our indexing assumption, $m_0 = 0$ at the start of the computation. After n transitions of M_k, we thus have $m_n \leq 2n$. It follows that the total number of transitions of M is $O(\sum_i m_i) = O(n^2)$. □

Corollary 3.2. *For every fixed k, a decision problem with time complexity f with respect to k-tapes Turing machines has complexity $O(f^2)$ with respect to (single tape) Turing machines.*

3.2 Random access machines

Turing machines and their multitape avatars, though they have a rather simple formal description, do not have a handy organisation of the stored data. As seen for the simulation of a multitape by a single tape machine, the machine needs to scan its whole content to retrieve the content of a single cell. This somehow hides the true algorithmic nature of a problem into functional details, such as accessing to a memory cell. The random access machine (RAM), which is an example of a *von Neumann architecture*, has a more convenient memory unit that leads to more intuitive implementations and complexity analyses of algorithms. This is the most common used model for analysing combinatorial algorithms, in particular because its description is closer to actual computers. There exists even more variants of RAM than variants of Turing machines, sometimes leading to drastically different complexity analysis. In general RAMs operate on integers rather than alphabets and their words. Since every finite alphabet and its words can be encoded by numbers or list of numbers, this is not a real limitation. We will present some kind of standard model and show that up to a polynomial gain, this model is again equivalent to the basic Turing machine.

a **random access machine** consists of four units

- a *memory unit* composed of a infinite sequence of *cells*, or *registers*, that may contain arbitrarily large integers. Each cell $R[i]$ is identified by its index in $i \in \mathbb{N}$.
- An *accumulator*, which is an isolated cell α, distinct from the memory unit, where all the computations take place.
- A *program unit*, which is the core of the machine. This is a finite sequence of instructions indexed by natural numbers.
- A *controller*, that reduces to a single cell κ, distinct from the memory unit, storing the index of the program instruction to be executed.

Except for the halting and halving instructions below, every program instruction has one integral parameter. The program instructions can be divided into three groups.

- **Transfers between memory and accumulator**. Specifically, instruction LOAD j has the effect of writing (the content of) $R[j]$ into α. There is another analogous instruction with *indirect addressing*: iLOAD j causes α to contain $R[R[j]]$. In the other direction, STORE j writes α into $R[j]$, while iSTORE j writes α into $R[R[j]]$
- **Operations on the accumulator**. WRITE x makes α to contain x. The instructions ADD j and SUB j respectively adds and subtracts the content of $R[j]$ to α. Instruction HALF has no parameter and divides the content of the accumulator by two: $\alpha \leftarrow \lfloor \alpha/2 \rfloor$.
- **Program control** After all the above instructions are executed, the controller κ is implicitly incremented by one. There are three other instructions to modify the flow of the program. JZERO i sets κ to i (so that the program *jumps* to instruction i) if the accumulator contains zero and otherwise increments the controller. Similarly, JPOS i sets κ to i if the accumulator is positive and otherwise increments the controller. Finally, instruction HALT causes the program to stop.

It is assumed that the program stops whenever an instruction has no interpretation, as in 'iLOAD j' when $R[j]$ is negative. We summarize the set of instructions below. The notations $\alpha \leftarrow \beta$, $\beta \rightarrow \alpha$, and $\alpha := \beta$ all have the same meaning[5] of writing β into α.

Instruction	**Semantics**	
LOAD j	$\alpha := R[j]$	$\kappa := \kappa + 1$
iLOAD j	$\alpha := R[R[j]]$	$\kappa := \kappa + 1$
STORE j	$R[j] := \alpha$	$\kappa := \kappa + 1$
iSTORE j	$R[R[j]] := \alpha$	$\kappa := \kappa + 1$
WRITE x	$\alpha := x$	$\kappa := \kappa + 1$
ADD j	$\alpha := \alpha + R[j]$	$\kappa := \kappa + 1$
SUB j	$\alpha := \alpha - R[j]$	$\kappa := \kappa + 1$
HALF	$\alpha := \lfloor \alpha/2 \rfloor$	$\kappa := \kappa + 1$
JZERO i		if $(\alpha = 0)$ then $\kappa := i$ else $\kappa := \kappa + 1$
JPOS i		if $(\alpha > 0)$ then $\kappa := i$ else $\kappa := \kappa + 1$
HALT		Stop the program

Note that the instruction set does not include multiplication or division operations apart from the HALF operation. Including those would not change the set of computable functions but would drastically change the complexity of problems. For instance the function $n \mapsto n^n$ can easily be implemented to run in $O(\log n)$ steps if multiplication is allowed. However, just writing the result requires $O(n \log n)$ bits. Allowing multiplication in the RAM model would therefore lead to an exponential speedup compared to Turing machines. The choice of instructions should thus be large enough to simulate Turing machines but not too large. Some instructions may be redundant when they facilitate the proof of equivalence with Turing machines. This

[5] Beware that in Computer Science, := does not denote a relation but the act of writing the *value* identified by β into the *container* named α.

is for instance the case of the ADD instruction. Indeed, assuming that $R[1]$ and $R[2]$ are unused registers, instruction 'ADD j' is the result of the following combination of instructions:

STORE 1
WRITE 0
SUB j
STORE 2
LOAD 1
SUB 2

The presence of an accumulator is not necessary to the model as one could perform arithmetic operations directly on the memory cells. This would however require two parameters per instruction instead of one and the model would be somehow less realistic. Intuitively, the accumulator plays the role of the CPU (Central Processing Unit) in a real computer.

3.2.1 Simulating Turing machines with RAMs

Given a deterministic Turing Machine $M = (\mathscr{A}, \mathscr{Q}, \mathscr{T})$, there is a simple representation of its tape content in the memory R of a RAM. First, encode every letter $a \in \mathscr{A}$ by an integer n_a chosen once for all. Then, index the cells of M's tape by $\mathbb{Z}$ and store the content a of cell i as n_a in register $R[I(i)]$, where $I(i) = 2i + 4$ if $i \geq 0$ and $I(i) = -2i + 3$ if $i < 0$. Also store the index of the cell pointed by the head of M in $R[0]$. This representation defines an injection of the set of tape contents of M into the set of contents of a R. Thanks to this representation, RAMs can faithfully and efficiently simulate Turing machines.

Theorem 3.3. *For every deterministic Turing Machine $M = (\mathscr{A}, \mathscr{Q}, \mathscr{T})$ there exists a RAM M' such that starting from a memory content representing an initial tape content of M, every transition of M corresponds to the execution of a constant number of instructions of M'. The content of the memory of M' after this constant number of instructions represents the tape content of M after the transition is applied.*

PROOF. We decompose the program defining M' into blocks, each indexed by a transition of M. If $t = aqbpD$ is such a transition, its corresponding block is

Line number	Instruction	Semantics
N_t	WRITE n_a	First check if the scanned symbol is a.
N_t+1	STORE 1	$R[1] := n_a$
N_t+2	LOAD 0	Start computing $I(R[0])$ from $R[0]$
N_t+3	JPOS N_t+9	
N_t+4	JZERO N_t+9	
N_t+5	WRITE 3	If $R[0]<0$ then …
N_t+6	SUB 0	
N_t+7	SUB 0	… $\alpha := I(R[0]) = 3-2R[0]$
N_t+8	JPOS N_t+12	Goto N_t+12
N_t+9	WRITE 4	Else …
N_t+10	ADD 0	
N_t+11	ADD 0	… $\alpha := I(R[0]) = 4+2R[0]$
N_t+12	STORE 2	$R[2] := I(R[0])$
N_t+13	iLOAD 2	$\alpha := R[I(R[0])]$
N_t+14	SUB 1	$\alpha := R[I(R[0])]-n_a$
N_t+15	JZERO N_t+18	If the scanned symbol is indeed a, apply transition t.
N_t+16	WRITE 0	Else check if scanned symbol is a' where $n_{a'} = n_a+1$ …
N_t+17	JZERO $N_{t'}$	…and $t' = a'qb'p'D' \in \mathscr{T}$
N_t+18	WRITE n_b	
N_t+19	iSTORE 2	Write n_b in place of n_a in $R[I(R[0])]$
N_t+20	LOAD 0	increment head index by d, where…
N_t+21	ADD d	… $d=-1,0,1$ according to whether $D=R,S,L$.
N_t+22	STORE 0	$R[0] := R[0]+d$
N_t+23	WRITE 0	
N_t+24	JZERO $N_{t''}$	Apply next transition $t'' = bpb''p''D'' \in \mathscr{T}$.

There should also be special blocks for the case of a halting state and when no transition applies, *i.e.* when the first part of the block from N_t to N_t+15 fails for all transitions starting with state q and any of the alphabet symbols. It is readily checked that the program defined by all those instruction blocks simulates M in the sense of the theorem. □

3.2.2 Simulating RAMs with Turing machines

Quite surprisingly, while RAMs permit the handling of arbitrarily large integers in a single instruction, it is no more powerful than Turing machines. In particular, we shall prove that the complexity of a decision problem under Turing machines is no more than polynomial in terms of its complexity under the RAM model.

We first observe that the size of the registers can grow at most linearly as instructions are executed. More precisely, consider a RAM with memory unit R. We view the initial content of R as the *input* of the RAM. To simplify the analysis, we assume without loss of generality that the input is stored in an initial segment of the memory that is never modified. We define the **size** of the input as the sum of the bit lengths of its registers. Also denote by s_P the maximal bit length of any integer that appears in the program instructions of the RAM.

Lemma 3.4. *After executing k instructions, starting with an input of size n and a zero accumulator, the bit length of the accumulator or of any register is at most $k+n+s_P$.*

PROOF. By induction on k. This is obvious for $k=0$. Suppose that the lemma is true after k instructions. If the next instruction to be executed is LOAD, iLOAD, STORE, iSTORE, HALF, JZERO, JPOS, HALT then the maximal bit length of the registers or accumulator cannot increase. If the next instruction is ADD, SUB or WRITE, then the length may increase by at most one bit since the sum (difference) of two numbers of bit length at most $k+n+s_P$ has length at most $k+n+s_P+1$. The lemma thus remains true after $k+1$ instructions are executed. □

We can easily represent the status of a RAM on a deterministic 3-tape Turing machine. Let $b(i)$ denotes the binary representation of integer i with a minus sign in front if negative (zero is encoded with the single 0 bit). The alphabet of the Turing machine includes the symbols $\emptyset, 0, 1, (,)$ as well as the coma "," and two delimiters, say $>$ and $<$. The first tape shall represent the memory R of the RAM as follows. If the input is a contiguous sequence of m integers, the tape contains the binary encoding $b(m)$ followed by the sequence of m input integers in binary, separated by comas. Any other register that is modified during the execution of the program is appended to this sequence as a pair $(b(i), b(R[i]))$ where i is the index of the register. The pairs may be possibly separated by blanks and the whole memory representation is surrounded by the delimiters to mark the beginning and end of the sequence. The second tape plays the role of the accumulator and holds its content in binary. The third tape is to contain a cell index in order to facilitate the search of the corresponding register content.

Theorem 3.5. *Given a RAM M, there exists a deterministic 3-tape Turing machine M_T simulating M as follows. For any input of M of size n, the execution of k instructions on M corresponds to $O(k^2(k+n))$ transitions of M_T starting with a representation of M's input on its first tape. After those transitions are applied, this first tape of M_T represents the memory of M at the end of the k instructions.*

PROOF. Denote by R the memory of M. We decompose the set of states of M_T into groups, one for each instruction line in the program of M. The group corresponding to an instruction allows its simulation thanks to an appropriate transition table. Consider for example the instruction iSTORE j. In a first step, the binary encoding of j is copied on the third tape and M_T scans its first tape until the jth input entry or the pair $(b(j), b(R[j]))$ is found, depending on whether j is smaller or larger than the length m of the input sequence. Then $b(j)$ is erased from the third tape and replaced by $b(R[j])$. In a second step, M_T scans once again its first tape in search of a pair $(b(R[j]), b(R[R[j]])$. (If $R[j]$ is smaller or equal to m the machine halts since it is not supposed to overwrite the input.) If it is found, M_T erases this pair by overwriting it with blanks. In any case M_T goes on until the end of the memory (indicated by the symbol $<$) and appends the new pair $(b(R[j]), b(\alpha))$ to the sequence, pushing the symbol $<$ accordingly, where α is the content of the accumulator supposedly copied on the second tape. The other instructions can be simulated analogously by introducing new states and extending the transition table. A full description of M_T would take

many pages but it should be clear from the above description that the number of transitions used to simulate an instruction is at most proportional to the size of the content of the three tapes. The total number of transitions necessary to simulate k instructions is thus proportional to the sum of the tape sizes at each transition in the simulation. The initial size for representing the m input integers is bounded by $n+m+\log m = O(n)$, where the $\log m$ term[6] stands for the encoding of m and the m term accounts for the comas in the input representation. The simulation of the jth transition elongates the first tape by at most one register pair of the form $(a, R[a])$, whose length is $O(j+n+s_P)$ by Lemma 3.4. By the same lemma, the length of the two other tapes is $O(j+n+s_P)$. It follows that the size of the tapes after simulating i transitions is $O(n+\sum_{j=0}^{i}(j+n+s_P)) = O(i(i+n))$. We conclude that the total number of transitions required by M_T to simulate k instructions of M is $O(\sum_{i=0}^{k} i(i+n)) = O(k^2(k+n))$. □

Say that a RAM solves a decision problem in time f if for every input of size n it decides (halts) after at most $f(n)$ transitions if the input represents a YES instance of the problem by writing a 1 in the accumulator, otherwise writing a zero.

Corollary 3.6. *Every decision problem with at least linear time complexity f with respect to the RAM model has time complexity $O(f^6)$ with respect to (single tape) deterministic Turing machines.*

PROOF. Let M be a RAM solving the given decision problem in time f. From the preceding theorem, there exists a deterministic 3-tape Turing machine solving the problem in time $O(f^2(n)(f(n)+n)) = O(f^3)$. By corollary 3.2, this implies the existence a single tape Turing machine solving the problem in time $O(f^6)$. □

Other RAM variants include the more realistic word-RAM for which each register can hold integers of bounded bit length, say w. (This word length is usually set to 32 or 64 in real computers.) Larger integers should be represented as sequences of words representing the digits in base 2^w. The accumulator may only operate on such bounded integers so that arithmetic operations on larger integers should be implemented by appropriate sequences of instructions. Another variant is to analyze the execution time of a RAM by assigning a cost to each instruction. A common cost accounts for the bit length of the integers manipulated by an instruction. Hence the cost of instruction 'iLOAD j' would be the sum of the binary lengths of j and $R[j]$.

3.3 Fundamental complexity classes

We now assume that the complexity of a problem is measured in terms of the Turing machine or any polynomially equivalent model such as the above standard RAM model.

[6] As often in Computer Science the logarithm is in base 2.

Polynomial and exponential classes. An algorithm has **polynomial time complexity** if for every $n \in \mathbb{N}$ and every input of length n the computation on this input has time complexity at most $p(n)$, where p is a polynomial that only depends on the algorithm. The set of problems admitting algorithms of polynomial time complexity is denoted by **P**. Replacing $p(n)$ by $2^{p(n)}$ we obtain the class **EXP** of problems with exponential time complexity. Analogously, the set of problems solved by Turing machines whose space complexity is polynomial is denoted by **PSPACE**. It is believed, but not known, that **EXP** $\not\subseteq$ **PSPACE**.

Exercise 3.7. Show that **PSPACE** $\subseteq$ **EXP**.

The class NP. The acronym **NP** stands for the class of *nondeterministic polynomial time* algorithms. A problem I is in **NP** if there is a nondeterministic Turing machine such that (1) given any $w \in I$ as input at least one computation leads to an accepting state in polynomial time and (2) no computation leads to an accepting state whenever $w \notin I$. Case (2) leaves the possibility that the machine runs forever, but computations that take more than polynomial time may be discarded without affecting the functionality of the machine, so that we can always assume that the computation takes polynomial time in both cases (1) and (2). However, the two cases are highly asymmetric since a computation leading to a rejecting state does not say anything about the input. There is another useful definition of the class **NP** in terms of efficiently verifiable certificate. A problem I is in **NP** if there is a deterministic Turing machine with polynomial time complexity, the *verifier*, such that (a) for every $w \in I$ of size n there exists a word $c \in \mathscr{B}^*$ of size polynomial in n, where $\mathscr{B}$ is the verifier's alphabet, so that the verifier accepts the pair (w, c) in time polynomial in n and (b) if $w \notin I$ the verifier rejects (w, c) whichever c we choose. Hence, c acts as a **certificate**, or efficiently verifiable proof for being a YES instance.

Theorem 3.8. *The two definitions of the class* **NP** *by means of nondeterministic machines or in terms of certificates and deterministic verifiers are equivalent.*

PROOF. Suppose that a language I is recognized by a nondeterministic machine M in polynomial time. By assumption, for every $w \in I$ there is a computation on M with input w that leads to an accepting state in polynomial time. As a certificate, we choose the list c of transitions applied during this computation. We define a verifier V that takes (w, c) as input and essentially simulates the computation of M on w guided by c. The successive transitions choices in c allow V to maintain the current configuration of M determined by those choices. The main task of the verifier is thus to check that each transition choice corresponds to an actual transition of M that applies to the current configuration. Clearly, V operates in polynomial time and $w \in I$ if and only if we can choose c so that the simulation leads to an accepting state of M. We have thus proved that I is in **NP** according to the second definition.

Conversely, suppose that every word in I has a certificate verifiable by a polynomial time Turing machine V. We define a nondeterministic machine M operating in two stages. In the first stage, M *guesses* a certificate with polynomial length $p(n)$, where n is the length of the input. In practice, this means that M includes $p(n)$ states, say $q_1, \ldots, q_{p(n)}$, together with all the transitions $a\,q_j\,b\,q_{j+1}\,R$ for all a, b in the alphabet of

V, and $1 \leq j \leq p(n)-1$. In the second stage, M simulates V deterministically on the input word concatenated with the guessed certificate. The nondeterminism of M is thus concentrated in the first stage. It is easily seen that M recognizes I as a member of **NP** in the sense of the first definition. □

Exercise 3.9. Show that **NP** ⊆ **PSPACE**.

3.3.1 Reduction and completeness

The notion of reduction allows to compare the difficulty of different problems. Given two problems $I, J \subseteq \mathscr{A}^*$, we say that I **reduces** (in polynomial time) to J, written $I \leq J$, if there is a map $r : \mathscr{A}^* \to \mathscr{A}^*$, computable by a Turing machine with polynomial time complexity, such that $I = r^{-1}(J)$. In other words, the reduction function r transforms YES and NO instances of the first problem to, respectively, YES and NO instances of the second problem[7]. Hence, $I \leq J$ and $J \in \mathbf{P}$ implies $I \in \mathbf{P}$. This is obviously true replacing **P** by any other larger complexity class. If I reduces to J and J to K, the composition of the reduction functions provides a reduction from I to K. The reduction relation is thus a preorder (*i.e.*, a reflexive and transitive relation). Any problem which is an upper bound for a complexity class C is said **C-hard**. It is said **C-complete** if it furthermore belongs to **C**. A **C**-complete problem is thus a hardest representative in **C**. It is a priori not clear whether a complexity class has complete problems.

It follows from the definition that every non-trivial problem (proper subset of $\mathscr{A}^*$) in **P** is **P**-complete. It turns out that the class **NP** has complete problems, among which the satisfiability problem. A Boolean formula is a logical expression over Boolean variables connected by the usual $\wedge, \vee, \neg$ operators. A formula is *satisfiable* if there is an assignment of its variables that makes the formula evaluate to true. The problem **SAT** is the set of satisfiable formulas encoded, say, over the alphabet $\{0, 1, \wedge, \vee, \neg, (,)\}$.

Theorem 3.10 (Cook'71 - Levin'73)**.** *SAT is* **NP***-complete.*

PROOF. Any truth assignment of a formula in SAT provides a certificate that is easily checkable in polynomial time. It follows that SAT ∈ **NP**. It remains to show that every problem $I \in \mathbf{NP}$ reduces to SAT. Let $M = (\mathscr{A}, \mathscr{Q}, \mathscr{T})$ be a nondeterministic machine solving I in polynomial time. For every instance w, we need to construct a formula Φ_w so that $w \in I$ if and only if Φ_w is satisfiable.

Number the cells of the tape once for all from left to right so that at the initial step the tape contains $w = w_0 w_1 \dots w_{n-1}$ with cell 0 containing w_0. By assumption on M, the number of computation steps given w as input is bounded by $p(n)$ for some polynomial p, where $n := |w|$ is the length of w. By convention, we consider that M stays in the same configuration once in a halting state. This way we can assume that the number of computation steps is exactly $p(n)$. It follows that the head of M can only point to a cell with index in the range $J := [-p(n), p(n)]$. In particular, cells with

[7] This type of reduction is called *many-one*, or *Karp reduction*. *Polynomial-time Turing reduction*, also known as *Cook reduction*, is another common notion of reduction, where I reduces to J if I can be solved in polynomial time by a Turing machine with an oracle for J, meaning that the machine is allowed to call a subroutine for problem J at anytime during the computation, in constant time for each call.

index outside this range must contain the empty symbol. The whole computation is thus entirely described by the content of the jth cell at the ith step (configuration) of the computation, with $1 \leq i \leq p(n)$ and $j \in J$, and the sequence of $p(n)$ states and head positions during the computation. In accordance with this description, we introduce Boolean variables $C_{i,j,s}, Q_{i,q}, H_{i,j}$ with $1 \leq i \leq p(n)$, $j \in J$, $s \in \mathscr{A}$ and $q \in \mathscr{Q}$. The variable $C_{i,j,s}$ is intended to be true whenever the jth cell at the ith step contains s and false otherwise. Similarly, $Q_{i,q}$ and $H_{i,j}$ are intended to be true exactly when M is in state q at step i with the head pointing to the jth cell.

We next consider the following Boolean formulas. We recall that $A \implies B$ is a shorthand for $\neg A \vee B$.

- $\phi_{i,j} = \bigvee_{s \in \mathscr{A}} (C_{i,j,s} \wedge (\bigwedge_{t \neq s} \neg C_{i,j,t}))$ expresses that the jth cell at the ith step contains one and only one symbol in $\mathscr{A}$.
- $\phi_i = \big(\bigvee_{q \in \mathscr{Q}} (Q_{i,q} \wedge (\bigwedge_{r \neq q} \neg Q_{i,r}))\big) \wedge \big(\bigvee_{j \in J} (H_{i,j} \wedge (\bigwedge_{k \neq j} \neg H_{i,k}))\big)$ expresses that the state and head position each takes exactly one value at the ith step.
- $\phi_b = \bigwedge_{1 \leq j \leq n} C_{1,j,w_j} \wedge \bigwedge_{j \notin [1,n]} C_{1,j,\emptyset} \wedge Q_{1,q_0} \wedge H_{1,1}$ expresses that the initial tape contains the input w and that M is in the initial state q_0 with the head pointing to the first symbol of w.
- $\phi_e = Q_{p(n),q_a}$, where q_a is the accepting state, expresses that M accepts w.
- $\psi_i = \bigwedge_{\substack{j \in J \\ s \neq t}} \big((C_{i,j,s} \wedge C_{i+1,j,t}) \implies H_{i,j} \big)$ expresses that only the cell pointed by the head may change from step i to $i+1$.
- $\psi_{i,j,q,s} = (Q_{i,q} \wedge H_{i,j} \wedge C_{i,j,s}) \implies \bigvee_{s\,q\,t\,r\,D \in \mathscr{T}} (Q_{i+1,r} \wedge H_{i+1,j+D} \wedge C_{i+1,j,t})$ expresses that when M is in state q at step i with the head pointing to the jth cell containing s, only the relevant transitions may apply. Here, $j+D$ is $j-1$ or $j+1$ depending on whether $D = L$ or $D = R$.

We finally set $\Phi_w = \bigwedge_{i,j} \phi_{i,j} \wedge \phi_b \wedge \phi_e \wedge \bigwedge_i \psi_i \wedge \bigwedge_{i,j,q,s} \psi_{i,j,q,s}$. To conclude, it remains to notice that the description of the formula Φ_w can be computed in polynomial time (with respect to n) and that Φ_w is satisfiable if and only if M recognizes w, *i.e.* $w \in I$. □

4 Some Basic Algorithms

Since graphs are among the main objects of this course, we describe the most basic graph traversal algorithms. We also sketch the Knuth-Morris-Pratt string searching algorithm that is useful for dealing with words in the context of combinatorial group presentations. Here, we use the word RAM model of computation, a very common variant of the RAM model well suited to analyse combinatorial algorithms [Hag98]. This model is parameterized by the word length σ. In this model, a memory cell may

only contain numbers of bit size σ and the size of the input is defined as the number of cells it uses. The rest of the description is otherwise the same as for the standard RAM described in Section 3.2. Implicitly, we are thus assuming that $\sigma = \log m$, where m is larger than the largest integer stored in the input and is also larger than the number of inputs. In practice, such a word RAM allows to add, subtract and multiply numbers in constant time. It can also call a subroutine or perform (conditional) jump in constant time, loop through a piece of code with unit cost per loop in addition to the cost of the piece of code. In the end, one can loosely describe an algorithm with a pseudocode in a way similar to any procedural programming language such as C++. See Algorithm 1 or 2 in Section 4.2 for an example. For and while loops, pointers, variable assignments, arithmetic operation are thus available to simply describe the algorithm and provide a complexity analysis equivalent to the word RAM model.

4.1 Graph traversals

We assume given a finite graph $G = (V, E)$ with vertex set V and edge set E (a more formal definition will be given in the next lecture). This graph is usually given in input by its **adjacency lists**. There is one list per vertex in V given as an array containing the index of the edges incident to that vertex. The size of this array is thus the degree of the vertex in the graph. The whole description of G takes $O(|V|+|E|)$ space.

A **graph traversal** – also known as graph search – from a source vertex $v \in V$ is a list of the vertices $v = v_0, v_1, \ldots, v_k$ of the connected component G_v of v in G so that for $i \in [1,k]$, the vertex v_i is connected to one of $v_0, \ldots, v_{i-1}$ through some edge e_i. The connecting edges e_i thus form a spanning tree (with source v) of G_v. The way v_i is chosen determines the type of the traversal. Two important traversals are breadth-first search and depth-first search.

4.1.1 Breadth-first search (BFS)

In a breadth-first search, v_i is recursively chosen so that e_i is incident to v_j with j minimal in $[0, i-1]$. Hence, if all the neighbours of v_0 have already been *visited*, *i.e.* their indices are smaller than i, then j must be at least 1. A classical implementation of Breadth-first search uses a FIFO (First In First Out) queue. Each vertex in V is associated with three attributes:

- a distance initially set to $+\infty$,
- the connecting edge to the parent vertex in the search, initially set to any value,
- a binary status marked/unmarked, initially set to unmarked.

In a first step, set the distance of the source vertex v to zero and set its connecting edge to some dummy value. Then, mark and enqueue v in an empty queue. As long as the queue is nonempty, the following is repeated. The head vertex of the queue, say u, is dequeud. If u is unmarked, then for every edge e in its adjacency list whose other endpoint w is unmarked set the distance of w to the distance of u plus one and set its connecting edge to e. Finally mark and enqueue w.

Note that nothing is done when u is already marked. For this traversal, vertices are ordered as they are dequeued. During the search, every vertex is enqueued and dequeued exactly once. Morover, each edge is scanned twice, once each time an endpoint is dequeued. It follows that the time complexity of breadth-first search is $O(|V|+|E|)$, hence linear. It is an exercise to show that when the algorithm stops the distance attribute of each vertex is its (graph) distance to the source vertex, *i.e.* the minimal length of a path connecting that vertex to the source vertex. This minimal length path is obtained by following the connecting edges from the vertex to the source. For this reason the BFS tree is also called a **shortest path tree**.

4.1.2 Depth-first search (DFS)

Depth-first search works very similarly to breadth-first search except that v_i is chosen to be connected to the most recently discovered vertex instead of the oldest. With the above notations, v_i is recursively chosen so that e_i is incident to v_j with j maximal in $[0, i-1]$. A possible implementation of depth-first search uses a LIFO (Last In First Out) stack. Each vertex in V is associated with a connecting edge and with a binary status marked/unmarked initialized as unmarked.

In a first step, set the connecting edge of the source vertex v to some dummy value. Then, push v on top of an empty stack. As long as the stack is nonempty, the following is repeated. The top vertex of the stack, say u, is popped. If u is unmarked, then mark u and for every edge e in its adjacency list whose other endpoint w is unmarked set its connecting edge to e and push w onto the stack.

Every pushed vertex can be associated with its connecting edge. Note that the pushed connecting edges are all distinct. It follows that the total number of pushes is bounded by $|E|$, though a vertex can be pushed more than once. We conclude that DFS runs in linear time.

A DFS tree has interesting properties. For instance, every chord of a DFS tree joins a vertex to an ancestor in the tree. For this reason the chords of a DFS tree are called *back edges*. DFS can be used to find the disconnecting vertices (articulation points) or disconnecting edges (bridges) of a graph in linear time [CLRS09]. It can also be used to test the planarity of a graph in linear time [Tar72].

4.2 The Knuth-Morris-Pratt (KMP) algorithm

In a string matching problem we are given two words on some alphabet, respectively called the *pattern* and the *text*. The problem is to determine if the pattern occurs as a subword of the text. We assume that pattern and text are provided as arrays $P = P[1,m]$ and $T = T[1,n]$ of respective length m and n. We say that P occurs with shift s in T if $T[s+1, s+m] = P$.

A straightforward algorithm is to test for every $s \in [1, n-m]$ whether P occurs with shift s in T. This last test takes $O(m)$ time for comparing the two arrays $T[s+1, s+m]$ and P. In total, the complexity of this algorithm is $O((n-m)m)$. The Knuth-Morris-Pratt algorithm from the early 1970's, also discovered by Yuri Matiyasevich around the same time, allows to reduce the complexity from quadratic to linear. Let us write $U \sqsubset V$ whenever U is a prefix of V and write $U \sqsupset V$ if U is a suffix of V. The KMP

algorithm starts computing a prefix function $\pi : [1,m] \to [0,m-1]$ such that $\pi(q)$ is the length of the longest *proper* prefix of $P[1,q]$ which is also a suffix of $P[1,q]$:

$$\pi(q) = \max\{k : k < q \text{ and } P[1,k] \sqsupset P[1,q]\}$$

In particular, $\pi(q) < q$. The function π can be easily computed recursively. We have $\pi(1) = 0$ and knowing $\pi(q)$ we may evaluate $\pi(q+1)$ as follows. Since $P[1,\pi(q)] \sqsupset P[1,q]$, we have two possibilities. Either the symbol just after $P[1,\pi(q)]$ coincides with the symbol just after $P[1,q]$, *i.e.* $P[\pi(q)+1] = P[q+1]$. In this case we may extend the prefix by one symbol. In other words, $\pi(q+1) = \pi(q)+1$. Or, we should try to extend $P[1,\pi(\pi(q))]$ since this is by definition the second longest proper prefix of P that ends at position q. This leads to the following pseudocode.

Algorithm 1 Calculate the prefix function π

```
1:  π(1) ← 0, i ← 0        // When i > 0, we maintain the property P[1,i] ⊐ P[1,q]
2:  for q = 1 to m − 1 do
3:     while i > 0 and P[i+1] ≠ P[q+1] do
4:        i ← π(i)
5:     end while
6:     if P[i+1] = P[q+1] then
7:        i ← i+1
8:     end if
9:     π(q+1) ← i
10: end for
```

Note that the variable i is incremented at line 7 only, so that it can increase by at most $m-1$ in the end. Since i remains nonnegative and π is strictly decreasing, line 4 can be executed at most $m-1$ times. It readily follows that the time complexity of Algorithm 1 is $O(m)$.

Having computed π, we may efficiently decide if P is a substring of T by computing for every $q \in [1,n]$ the longest prefix of P that is also a suffix of $T[1,q]$. In other words we may compute $\omega : [0,n] \to [0,m]$ such that $\omega(0) = 0$ and for $q \in [1,n]$:

$$\omega(q) = \max\{k : P[1,k] \sqsupset T[1,q]\}$$

We have that P occurs with shift s in T if and only if $\omega(s+m) = m$. As for π, ω may be computed recursively: Either $P[\omega(q)+1]$ coincides with $T[q+1]$, in which case $\omega(q+1) = \omega(q)+1$, or we should try to extend $P[1,\pi(\omega(q))]$ since this is the second longest prefix of P that is also a suffix of $T[1,q]$. This leads to the following pseudocode.

Algorithm 2 Calculate the prefix function ω

```
1: ω(0) ← 0, i ← 0        // When i > 0, we maintain the property P[1, i] ⊐ T[1, q]
2: for q = 0 to n − 1 do
3:    while i > 0 and P[i + 1] ≠ T[q + 1] do
4:       i ← π(i)
5:    end while
6:    if P[i + 1] = T[q + 1] then
7:       i ← i + 1
8:    end if
9:    ω(q + 1) ← i
10: end for
```

The same argument on the variable i as for algorithm 1 shows that the time complexity of computing is $O(n)$. We conclude that given P and T, KMP runs in time $O(|P|+|T|)$.

References

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[CLRS09] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[Hag98] Torben Hagerup. Sorting and searching on the word ram. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 366–398. Springer, 1998.

[Pap94] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[Tar72] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms*. Number 44 in CBMS-NFS Regional conference series in applied mathematics. SIAM, 1983.

[Tur36] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(Series 2):230–265, 1936.

[vEB90] Peter van Emde Boas. *Handbook of theoretical computer science*, volume A, chapter Machine Models and Simulations, pages 3–66. Elsevier, 1990.

[Wig06] Avi Wigderson. P, NP and mathematics – a computational complexity perspective. In *Proc. of the 2006 International Congress of Mathematicians*, volume 3, 2006.