Texts in Computational Complexity: The preliminaries – computational tasks and models

Oded Goldreich
Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Rehovot, Israel.

December 15, 2005

Preface

This text provides the necessary preliminaries for a course on computational complexity. It includes a discussion of computational tasks and computational models, as well as natural complexity measures associated with the latter. More specifically, this text recalls the basic notions and results of computability theory (including the definition of Turing machines, some undecidability results, the notion of universal machines, and the definition of oracle machines). In addition, this text presents the basic notions underlying non-uniform models of computation (like Boolean circuits).

We start by introducing the general framework for a discussion of computational tasks (or problems), which refers to the representation of instances and to two types of tasks (i.e., searching for solutions and making decisions). Once the stage is set, we consider two types of models of computation: uniform models that correspond to the intuitive notion of an algorithm, and non-uniform models (e.g., Boolean circuits) that allow for a closer look at the way computation progresses.

The contents of Sections 1–3 corresponds to a traditional *Computability course*, and most of this material is taken for granted in the rest of the current course. In contrast, Section 4 presents basic preliminaries regarding non-uniform models of computation (i.e., various types of Boolean circuits), and these are only used lightly in the rest of the current course. Thus, whereas Sections 1–3 are absolute prerequisites for the rest of this course, Section 4 is not.

Teaching note: I believe that there is no real need for a semester-long course in Computability (i.e., a course that focuses on what can be computed rather than on what can be computed efficiently). Instead, undergraduates should take a course in computational complexity, where the computability aspects will serve as a basis for the rest of the course. Specifically, the former aspects should occupy at most 25% of the course, and the focus should be on basic complexity (i.e., P, NP and NP-completeness) and on some more advanced material.

1 Representation

In Mathematics and related sciences, it is customary to discuss objects without specifying their representation. This is not possible in the theory of computation, where the representation of objects plays a central role. In a sense, a computation merely transforms one representation of an object to another representation of the same object. In particular, a computation designed to solve some problem merely transforms the problem instance to its solution, where the latter can be though of as a (possibly partial) representation of the instance. Indeed, the answer to any fully specified question is implicit in the question itself.

Computation refers to objects that are represented in some canonical way, where such canonical representation provides an "explicit" and "full" description of the corresponding object. We will consider only *finite* objects like sets, graphs, numbers, and functions (and keep distinguishing these types of objects although, actually, they are all equivalent).

Strings. We consider finite objects, each represented by a finite binary sequence, called a string. For a natural number n, we denote by $\{0,1\}^n$ the set of all strings of length n, hereafter referred to as n-bit strings. The set of all strings is denoted $\{0,1\}^*$; that is, $\{0,1\}^* = \cup_{n\in\mathbb{N}}\{0,1\}^n$. For $x\in\{0,1\}^*$, we denote by $|x|$ the length of x (i.e., $x\in\{0,1\}^{|x|}$), and often denote by x_i the $i^{\rm th}$ bit of x (i.e., $x = x_1x_2\cdots x_{|x|}$). For $x,y\in\{0,1\}^*$, we denote by xy the string resulting from concatenation of the strings x and y.

At times, we associate $\{0,1\}^*\times\{0,1\}^*$ with $\{0,1\}^*$; the reader should merely consider an adequate encoding (e.g., the pair $(x_1\cdots x_m, y_1\cdots y_n)\in\{0,1\}^*\times\{0,1\}^*$ may be encoded by the string $x_1x_1\cdots x_mx_m01y_1\cdots y_n \in \{0,1\}^*$). Likewise, we may represent sequences of strings (of fixed or varying length) as single strings. When we wish to emphasize that such a sequence (or some other object) is to be considered as a single object we use the notation $\langle\cdot\rangle$ (e.g., "the pair (x,y) is encoded as the string $\langle x,y\rangle$").

Numbers. Unless stated differently, natural numbers will be encoded by their binary expansion; that is, the string $b_{n-1}\cdots b_1b_0 \in \{0,1\}^n$ encodes the number $\sum_{i=0}^{n-1} b_i\cdot 2^i$. Rational numbers will be represented as pairs of natural numbers. In the rare cases in which one considers real numbers as part of the input to a computational problem, one actually mean rational approximations of these real numbers.

Special symbols. We denote the empty string by λ (i.e., $\lambda\in\{0,1\}^*$ and $|\lambda| = 0$), and the empty set by $\emptyset$. It will be convenient to use some special symbols that are not in $\{0,1\}^*$. One such symbol is $\perp$, which typically denotes an indication by some algorithm that something is wrong.

2 Computational Tasks

Two fundamental types of computational tasks are so-called search problems and decision problems. In both cases, the key notions are the problem's *instances* and the problem's specification.

Search problems. A search problem consists of a specification of a set of valid solutions (possibly an empty one) for each possible instance. That is, given an instance, one is required to find a corresponding solution (or to determine that no such solution exists). For example, consider the problem in which one is given a system of equations and is asked to find a valid solution. Needless to say, much of computer science is concerned with solving various search problems. Furthermore, search problems correspond to the daily notion of "solving a problem" and thus a discussion of the possibility and complexity of solving search problems corresponds to the natural concerns of most people. In the following definition of solving search problems, the potential solver is a function (which may be thought of as a solving strategy), and the sets of possible solutions associated with each of the various instances are "packed" into a single binary relation.

Definition 1 (solving a search problem) *Let* $R \subseteq \{0,1\}^* \times \{0,1\}^*$. *A function* $f : \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$ solves the search problem of R *if for every* x *it holds that* $(x, f(x)) \in R$ *if and only if* $R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\}$ *is not empty.*

Indeed, $R(x)$ denotes the set of valid solutions for the problem instance x, and it is required that whenever there exist valid solutions (i.e., $R(x)$ is not empty) the solver finds one. It is also required that the solver f never outputs a wrong solution (i.e., if $R(x) \neq \emptyset$ then $f(x) \in R(x)$), and it follows that if $R(x) = \emptyset$ then $f(x) = \perp$, which in turn means that f indicates that x has no solution. A special case of interest is the case that $|R(x)| = 1$ for every x, where R is essentially a (total) function, and solving the search problem of R means computing (or evaluating) the function R (or rather the function R' defined defined by $R'(x) \stackrel{\text{def}}{=} y$ where $R(x) = \{y\}$).

Decision problems. A decision problem consists of a specification of a subset of the possible instances. Given an instance, one is required to determine whether the instance is in the specified set. For example, consider the problem where one is given a natural number, and is asked to determine whether or not the number is a prime. One important case, which corresponds to the aforementioned search problems, is the case of the set of instances having a solution; indeed, being able to determine whether or not a solution exists is a prerequisite to being able to solve the corresponding search problem (as per Definition 1). In general, decision problems refer to the natural task of making binary decision, a task that is not uncommon in daily life. In any case, in the following definition of solving search problems, the potential solver is again a function (i.e., in this case it is a Boolean function that is supposed to indicate membership in the said set).

Definition 2 (solving a decision problem) *Let* $S \subseteq \{0,1\}^*$. *A function* $f : \{0,1\}^* \to \{0,1\}$ solves the decision problem of S (or decides membership in S) *if for every* x *it holds that* $f(x) = 1$ *if and only if* $x \in S$.

Indeed, if f solves the search problem of R then the Boolean function $f' : \{0,1\}^* \to \{0,1\}$ defined by $f'(x) \stackrel{\text{def}}{=} 1$ if and only if $f(x) \neq \perp$ solves the decision problem of $S \stackrel{\text{def}}{=} \{x : R(x) \neq \emptyset\}$. We often identify the decision problem of S with S itself, and identify S with its characteristic function (i.e., with $\chi_S : \{0,1\}^* \to \{0,1\}$ defined such that $\chi_(x) = 1$ if and only if $x \in S$).

Most people would consider search problems to be more natural than decision problems: typically, people seeks solutions more than they stop to wonder whether or not solutions exist. Definitely, search problems are not less important than decision problems, it is merely that their study tends to require more cumbersome formulations. This is the main reason that most expositions choose to focus on decision problems. The current course attempts to devote at least a significant amount of attention also to search problems.

Promise problems (an advanced comment). Many natural search and decision problems are captured more naturally by the terminology of promise problems, where the domain of possible instances is a subset of $\{0,1\}^*$ rather than $\{0,1\}^*$ itself. In particular, note that the natural formulation of many search and decision problems refers to instances of a certain types (e.g., a system of equations, a pair of numbers, a graph), whereas the natural representation of these objects uses only a strict subset of $\{0,1\}^*$. A nasty convention is to postulate that every string represents some legitimate object (i.e., each string that is not used in the natural representation of these objects is postulated as a representation of some fixed object). In the current text, we will ignore this issue, but refer the interested reader to [1].

3 Uniform Models (Algorithms)

We are all familiar with computers, and the ability of computer programs to manipulate data. But how does one capture *all* computational processes? Before being formal, we offer a loose description, capturing many artificial as well as natural processes, whereas the former are associated with computers and the latter are used to model (aspects of) the natural reality (be it physical, biological, or even social).

A *computation* is a process that modifies an environment via repeated applications of a predetermined rule. The key restriction is that this rule is *simple*: in each application it depends and affects only a (small) portion of the environment, called the active zone. We contrast the *a-priori bounded* size of the active zone (and of the modification rule) with the *a-priori unbounded* size of the entire environment. We note that, although each application of the rule has a very limited effect, the effect of many applications of the rule may be very complex. Put in other words, a computation may modify the relevant environment in a very complex way, although it is merely a process of repeatedly applying a simple rule.

As hinted, the notion of computation can be used to model some aspects of the natural reality. In this case, the process that takes place in the natural reality is the starting point of the study, and the goal of the study is to learn the (computation) rule that underlies this natural process. In a sense, the goal of Science at large can be phrased as learning (simple) rules that govern various aspects of reality (or rather one's abstraction of these aspects of reality).

Our focus, however, is on artificial computation rules designed by humans in order to achieve specific desired effects on the corresponding artificial environment. Thus, our starting point is a desired functionality, and our aim is to design computation rules that effect it. Such a computation rule is referred to as an algorithm. Loosely speaking, an algorithm corresponds to a computer program written in a high-level (abstract) programming language. Let us elaborate.

We are interested in the transformation of the environment affected by the computational process (or the algorithm). Throughout (most of) this course, we will assume that, when invoked on any finite initial environment, the computation halts after a finite number of steps. Typically, the initial environment to which the computation is applied encodes an input string, and the end environment (i.e., at termination of the computation) encodes an output string. We consider the mapping from inputs to outputs induced by the computation; that is, for each possible input x, we consider the output y obtained at the end of a computation initiated with input x, and say that the computation maps input x to output y. Thus, a computation rule (or an algorithm) determines a function (computed by it): this function is exactly the aforementioned mapping of inputs to outputs.

In the rest of this course, we will also consider the number of steps (i.e., applications of the rule) taken by the computation for each possible input. The latter function is called the time complexity of the computational process (or algorithm). While time complexity is defined per input, we will often considers it per input length, taking the maximum over all inputs of the same length.

In order to define computation (and computation time) rigorously, one needs to specify some model of computation; that is, provide a concrete definition of environments and a class of rules that may be applied to them. Such a model corresponds to an abstraction of a real computer (be it a PC, mainframe or network of computers). One simple abstract model that is commonly used is that of *Turing machines* (see, Section 3.1 below). Thus, specific algorithms are typically formalized by corresponding Turing machines (and their time complexity is represented by the time complexity of the corresponding Turing machines). We stress, however, that most results in the Theory of Computation hold regardless of the specific computational model used, as long as it is "reasonable" (i.e., satisfies the aforementioned simplicity condition and can perform some obviously simple computations).

What is being Computed? The above discussion has implicitly referred to computations and algorithms as means of computing functions. Specifically, an algorithm A computes the function $f_A : \{0,1\}^* \to \{0,1\}^*$ defined by $f_A(x) = y$ if, when invoked on input x, algorithm A halts with output y. However, computations can also be viewed as a means of "solving search problems" or "making decisions" (as in Definitions 1 and 2). Specifically, we will say that algorithm A solves the search problem of R (resp., decides membership in S) if f_A solves the search problem of R (resp., decides membership in S). In the rest of this exposition we associate the algorithm A with the function f_A computed by it; that is, we write $A(x)$ instead of $f_A(x)$. For sake of future reference, we summarize the foregoing discussion.

Definition 3 (solution by an algorithm) *We denote by $A(x)$ the output of algorithm A on input x. Algorithm A* solves the search problem *R* (resp., the decision problem *S*) *if A, viewed as a function, solves R* (resp., *S*)*.*

Organization of the rest of this section: In Section 3.1 we provide a sketchy description of the model of Turing machines. This is done merely for sake of providing a concrete model that supports the study of computation and its complexity, whereas most of the material in this course will not depend on the specifics of this model. In Section 3.2 and Section 3.2 we discuss two fundamental properties of any reasonable model of computation: the existence of uncomputable functions and the existence of universal computations. The time (and space) complexity of computation is defined in Section 3.4. We also discuss oracle machines and restricted models of computation (in Section 3.5 and Section 3.6, respectively).

3.1 Turing machines

The model of Turing machines offer a relatively simple formulation of the notion of an algorithm. The fact that the model is very simple complicates the design of machines that solve desired problems, but it makes the analysis of such machines simpler. Since the focus of complexity theory is on the analysis of machines and not on their design, the choice of this model and the trade-off that it offers is a good one. We stress again that the model is merely used as a concrete formulation of the intuitive notion of an algorithm, whereas we actually care about the intuitive notion and not its formulation. In particular, all results mentioned in this course hold for any other "reasonable" formulation of the notion of an algorithm.

The model of Turing machines is not supposed to provide a good (or "tight") model of real-life computers (although a task can be solved by a real-life computer if and only if it can be solved by a Turing machine). Historically, the model of Turing machines was invented before modern computers were even built, and was meant to provide a concrete model of computation (as opposed to the abstract definition of "recursive functions" that defines a class of "computable" functions in terms of composition of such functions). Indeed, this concrete model clarified fundamental properties of computable functions and plays a key role in defining the complexity of computable functions.

The model of Turing machines was envisioned as an abstraction of the process of an algebraic computation carried out by a human using a sheet of paper. In such a process, at each time, the human looks at some location on the paper, and depending on what he/she sees and what he/she has in mind (which is little...), he/she modifies the contents of this location and moves its look to an adjacent location.

The actual model. Following is a high-level description of the model of Turing machines; the interested reader is referred to standard textbooks (e.g., [9]) for further details. Recall that we need to specify the set of possible environments, the set of machines (or computation rules), and the effect of applying such a rule on an environment.

- The main component in the environment of a Turing machine is an infinite sequence of cells, each capable of holding a single symbol (i.e., member of a finite set $\Sigma \supset \{0,1\}$). In addition, he environment contains the current location of the machine on this sequence, and the internal state of the machine (which is a member of a finite set Q). The aforementioned sequence of cells is called the tape, and its contents combined with the machine's location and internal state is called the instantaneous configuration of the machine.
- The Turing machine itself consists of a finite rule (i.e., a finite function), called the transition function, which is defined over the set of all possible symbol-state pairs. Specifically, the transition function is a mapping from $\Sigma \times Q$ to $\Sigma \times Q \times \{-1,0,+1\}$, where $\{-1,+1,0\}$ correspond to a movement instruction (which is either "left" or "right" or "stay", respectively). In addition, the machine's description specifies an initial state and a halting state, and the computation of the machine halts when the machine enters its halting state.[1]

  In contrast to the finite description of the machine, the tape has an a priori unbounded length (and is considered, for simplicity, as being infinite).
- A single computation step of such a Turing machine depends on its current location on the tape, on the contents of the corresponding cell and on the internal state of the machine. Based on the latter two elements, the transition function determines a new symbol-state pair as well as a movement instruction (i.e., "left" or "right" or "stay"). The machine modifies the contents of the said cell and its internal state accordingly, and moves as directed. That is, suppose that the machine is in state q and resides in a cell containing the symbol σ, and suppose that the transition function maps (σ, q) to (σ', q', D). Then, the machine modifies the contents of the said cell to σ', modifies its internal state to q', and moves one cell in direction D. Figure 1 shows a single step of a Turing machine that, when in state 'b' and seeing a binary symbol σ, replaces σ with the symbol $\sigma + 2$, maintains its internal state, and moves one position to the right.[2]

  Formally, we define the successive configuration function that maps each instantaneous configuration to the one resulting by letting the machine take a single step. This function modifies only the contents of one cell (i.e. at which the machine resides), the internal state of the machine and its location, as described above.

[1] Envisioning the tape as extending from left to right, we also use the convention by which if the machine tries to move left of the end of the tape then it is considered to have halted.

[2] Figure 1 corresponds to a machine that, when in the initial state (i.e., 'a'), replaces the symbol σ by $\sigma + 4$, modifies its internal state to 'b', and moves one position to the right. Indeed, "marking" the leftmost cell (in order to allow for recognizing it in the future), is a common practice in the design of Turing machines.

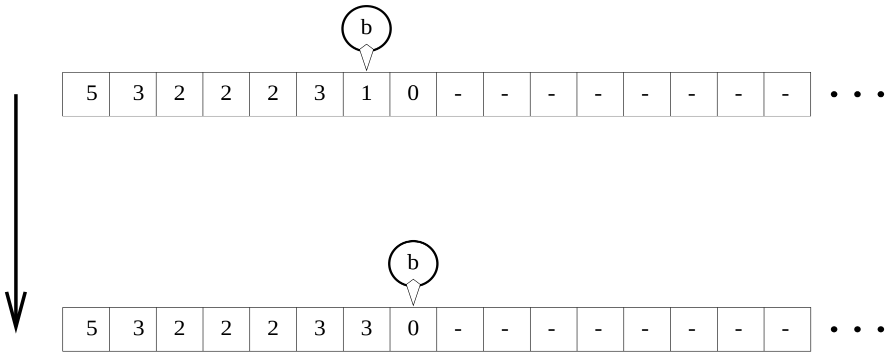

Figure 1: A single step by a Turing machine.

The initial environment (or configuration) of a Turing machine consists of the machine residing in the first (i.e., left-most) cell and being in its initial state. Typically, one also mandates that, in the initial configuration, a prefix of the tape's cells hold bit values, which concatenated together are considered the input, and the rest of the tape's cells hold a special symbol (which in Figure 1 is denoted by '-'). Once the machine halts, the output is defined as the contents of the cells that are to the left of its location (at termination time).[3] Thus, each machine defines a function mapping inputs to outputs, called the function computed by the machine.

Multi-tape Turing machines. We comment that in most expositions, one refers to the location of the "head of the machine" on the tape (rather than to the "location of the machine on the tape"). The standard terminology is more intuitive when extending the basic model, which refers to a single tape, to a model that supports a constant number of tapes. In the model of multi-tape machines, each step of the machine depends and effects the cells that are at the head location of the machine on each tape. As we shall see in Section 3.4, the extension of the model to multi-tape Turing machines is crucial to the definition of space complexity. A less fundamental advantage of the model of multi-tape Turing machines is that it allows for an easier design of machines that compute desired functions.

> **Teaching note:** I strongly recommend avoiding the standard practice of teaching the student to program with Turing machines. These exercises seem very painful and pointless. Instead, one should prove that a function can be computed by a Turing machine if and only if it is computable by a model that is closer to a real-life computer (see "sanity check" below). For starters, one should prove that a function can be computed by a single-tape Turing machine if and only if it is computable by a multi-tape (e.g., two-tape) Turing machine.

The Church-Turing Thesis: The entire point of the model of Turing machines is its simplicity. That is, in comparison to more "realistic" models of computation, it is simpler to formulate the model of Turing machines and to analyze machines in this model. The Church-Turing Thesis asserts that nothing is lost by considering the Turing machine model: *A function can be computed by some Turing machine if and only if it can be computed by some machine of any other* "reasonable and general" *model of computation.*

[3]By an alternative convention, the machine halts while residing in the left-most cell, and the output is defined as the maximal prefix of the tape contents that contains only bit values.

This is a thesis, rather than a theorem, because it refers to an intuitive notion that is left undefined on purpose (i.e., the notion of a *reasonable and general model of computation*). The model should be reasonable in the sense that it should refer to computation rules that are "simple" in some intuitive sense. On the other hand, the model should allow to compute functions that intuitively seem computable. At the very least the model should allow to emulate Turing machines (i.e., compute the function that given a description of a Turing machine and an instantaneous configuration returns the successive configuration).

A philosophical comment. The fact that a thesis is used to link an intuitive concept to a formal definition is common practice in any science (or, more broadly, in any attempt to reason rigorously about intuitive concepts). The moment an intuition is rigorously defined, it stops being an intuition, and becomes a definition and the question of the correspondence between the original intuition and the derived definition arises. This question can never be rigorously treated, because it relates to two objects, one being undefined. Thus, the question of correspondence between the intuition and the definition always transcends a rigorous treatment (i.e., it is always at the domain of the intuition).

A sanity check: Turing machines can emulate an abstract RAM. To gain confidence in the Church-Turing Thesis, one may attempt to define an abstract Random-Access Machine (RAM), and verify that it can be emulated by a Turing machine. An abstract RAM consists of an infinite number of memory cells, each capable of holding an integer, a finite number of similar registers, one designated as program counter, and a program consisting of instructions selected from a finite set. The set of possible instructions includes the following instructions:

- `reset`(r), where r is an index of a register, results in setting the value of register r to zero.
- `inc`(r), where r is an index of a register, results in incrementing the content of register r. Similarly `dec`(r) causes a decrement.
- `load`(r_1, r_2), where r_1 and r_2 are indices of registers, results in loading to register r_1 the contents of the memory location m, where m is the current contents of register r_2.
- `store`(r_1, r_2), stores the contents of register r_1 in the memory, analogously to `load`.
- `cond-goto`(r, ℓ), where r is an index of a register and ℓ does not exceed the program length, results in setting the program counter to $\ell - 1$ if the content of register r is non-negative.

The program counter is incremented after the execution of each instruction, and the next instruction to be executed by the machine is the one to which the program counter points (and the machine halts if the program counter exceeds the program's length). The input to the machine may be defined as the contents of the first n memory cells, where n is placed in a special input register. We note that the RAM model satisfies the Church-Turing Thesis, but in order to make it closer to a real-life computer we augment the model by instructions like `add`(r_1, r_2) (resp., `mult`(r_1, r_2)), which results in adding (resp., multiplying) the contents of registers r_1 and r_2 and placing the result in register r_1. We suggest proving that this abstract RAM can be emulated by a Turing machine.[4]

[4] We emphasize this direction of the equivalence of the two models, because the RAM model is introduced in order to convince the reader that Turing machines are not too weak (as a model of general computation). The fact that they are not too strong seems self-evident. Thus, it seems pointless to prove that the RAM model can emulate Turing machines. Still, note that this is indeed the case, by using the RAM's memory cells to store the contents of the cells of the Turing machine's tape.

(Hint: note that during the emulation, we only need to hold the input, the contents of all registers, and the contents of the memory cells that were accessed during the computation.)[5]

Observe that the abstract RAM model is more cumbersome than the Turing machine model. Furthermore, the question of which instructions to allow causes a vicious cycle, which we avoided by trusting the reader to consider only the standard instructions common in any real-life computer. (In general, we should only allow instructions that correspond to "simple" operations; i.e., operations that correspond to easily computable functions...)

3.2 Uncomputable functions

Strictly speaking, the current subsection is not necessary for the rest of this course, but we feel that it provides a useful perspective.

In contrast to what every layman would think, we know that not all functions are computable. Indeed, an important message to be communicated to the world is that *not every well-defined task can be solved* by applying a "reasonable" procedure (i.e., a procedure that has a simple description that can be applied to any instance of the problem at hand). Furthermore, not only is it the case that there exist uncomputable functions, but it is rather that "most" functions are uncomputable. In fact, only relatively few functions are computable.

Theorem 4 (on the scarcity of computable functions): *The set of computable functions is countable, whereas the set of all functions* (from strings to string) *has cardinality* $\aleph$.

We stress that the theorem holds for any reasonable model of computation. In fact, it only relies on the postulate that each machine in the model has a finite description (i.e., can be described by a string).

Proof: Since each computable function is computable by a machine that has a finite description, there is a 1-1 correspondence between the set of computable functions and the set of strings (which in turn is in 1-1 correspondence to the natural numbers). On the other hand, there is a 1-1 correspondence between the set of Boolean functions (i.e., functions from strings to a bit) and the set of real number in $[0,1)$. This correspondence associates each real $r \in [0,1)$ to the function $f : \mathbb{N} \to \{0,1\}$ such that $f(i)$ is the i^{th} bit in the binary expansion of r. ■

The Halting Problem: In contrast to the preliminary discussion, at this point we consider also machines that may not halt on some inputs. (The functions computed by such machines are partial functions that are defined only on inputs on which the machine halts.) Again, we rely on the postulate that each machine in the model has a finite description, and denote the description of machine M by $\langle M\rangle \in \{0,1\}^*$. The halting function, $\mathtt{h} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$, is defined such that $\mathtt{h}(\langle M\rangle, x) \stackrel{\text{def}}{=} 1$ if and only if M halts on input x. The following result goes beyond Theorem 4 by pointing to an explicit function (of natural interest) that is not computable.

Theorem 5 (undecidability of the halting problem): *The halting function is not computable.*

The term undecidability means that the corresponding decision problem cannot be solved by an algorithm. That is, Theorem 5 asserts that the decision problem associated with the set $\mathtt{h}^{-1}(1) =$

[5]Thus, at each time, the Turning machine's tape contains a list of the RAM's memory cells that were accessed so far as well as their current contents. When we emulate a RAM instruction, we first check whether the relevant RAM cell appears on this list, and augment the list by a corresponding entry or modify this entry as needed.

$\{(\langle M\rangle, x) : \mathtt{h}(\langle M\rangle, x) = 1\}$ is not solvable by an algorithm (i.e., there exists no algorithm that, given a pair $(\langle M\rangle, x)$, decides whether or not M halts on input x). Actually, the following proof shows that there exists no algorithm that, given $\langle M\rangle$, decides whether or not M halts on input $\langle M\rangle$.

Proof: We will show that even the restriction of $\mathtt{h}$ to its "diagonal" (i.e., the function $\mathtt{d}(\langle M\rangle) \stackrel{\text{def}}{=} \mathtt{h}(\langle M\rangle, \langle M\rangle)$) is not computable. Note that the value of $\mathtt{d}(\langle M\rangle)$ refers to the question of what happens when we feed M with its own description, which is indeed a "nasty" (but legitimate) thing to do. We will actually do worse: towards the contradiction, we will consider the value of $\mathtt{d}$ when evaluated at a (machine that is related to a) machine that supposedly computes $\mathtt{d}$.

We start by considering a related function, $\mathtt{d}'$, and showing that this function is uncomputable. The function $\mathtt{d}' : \{0,1\}^* \to \{0,1\}$ is defined such that $\mathtt{d}'(\langle M\rangle) \stackrel{\text{def}}{=} 1$ if and only if M halts on input $\langle M\rangle$ with output 0. (That is, $\mathtt{d}'(\langle M\rangle) = 1$ if M halts on input $\langle M\rangle$ with a specific output, and $\mathtt{d}'(\langle M\rangle) = 0$ if either M does not halt on input $\langle M\rangle$ or its output does not equal the designated value.) Now, suppose, towards the contradiction, that $\mathtt{d}'$ is computable by some machine, denoted $M_{\mathtt{d}'}$. Note that machine $M_{\mathtt{d}'}$ is supposed to halt on every input, and so $M_{\mathtt{d}'}$ halts on input $\langle M_{\mathtt{d}'}\rangle$. But, by definition of $\mathtt{d}'$, it holds that $\mathtt{d}'(\langle M_{\mathtt{d}'}\rangle) = 1$ if and only if $M_{\mathtt{d}'}$ halts on input $\langle M_{\mathtt{d}'}\rangle$ with output 0 (i.e., if and only if $M_{\mathtt{d}'}(\langle M_{\mathtt{d}'}\rangle) = 0$). Thus, $M_{\mathtt{d}'}(\langle M_{\mathtt{d}'}\rangle) \neq \mathtt{d}'(\langle M_{\mathtt{d}'}\rangle)$ in contradiction to the hypothesis that $M_{\mathtt{d}'}$ computes $\mathtt{d}'$.

We next prove that $\mathtt{d}$ is uncomputable, and thus $\mathtt{h}$ is uncomputable (because $\mathtt{d}(z) = \mathtt{h}(z, z)$ for every z). To prove that $\mathtt{d}$ is uncomputable, we show that if $\mathtt{d}$ is computable then so is $\mathtt{d}'$ (which we already know not to be the case). Let A be an algorithm for computing $\mathtt{d}$ (i.e., $A(\langle M\rangle) = d(\langle M\rangle)$ for every machine M). Then we construct an algorithm for computing $\mathtt{d}'$, which given $\langle M'\rangle$, invokes A on $\langle M''\rangle$, where M'' is defined to operate as follows:

1. On input x, machine M'' emulates M' on input x.
2. If M' halts on input x with output 0 then M'' halts.
3. If M' halts on input x with an output different from 0 then M'' enters an infinite loop (and thus does not halt).
4. Otherwise (i.e., M' does not halt on input x), then machine M'' does not halt (because it just stays stuck in Step 1 forever).

Note that the mapping from $\langle M'\rangle$ to $\langle M''\rangle$ is easily computable (by augmenting M' with instructions to test its output and enter an infinite loop if necessary), and that $\mathtt{d}(\langle M''\rangle) = \mathtt{d}'(\langle M'\rangle)$, because M'' halts on x if and only if M'' halts on x with output 0. We thus derived an algorithm for computing $\mathtt{d}'$ (i.e., transform the input $\langle M'\rangle$ into $\langle M''\rangle$ and output $A(\langle M''\rangle)$), which contradicts the already established fact by which $\mathtt{d}'$ is uncomputable. ■

Turing-reductions. The core of the second part of the proof of Theorem 5 is an algorithm that solves one problem (i.e., computes $\mathtt{d}'$) by using as a subroutine an algorithm that solves another problem (i.e., computes $\mathtt{h}$). In fact, the first algorithm is actually an algorithmic scheme that refers to a "functionally specified" subroutine rather than to an actual (implementation of such a) subroutine, which may not exist. Such an algorithmic scheme is called a Turing-reduction (i.e., we have Turing-reduced the computation of $\mathtt{d}'$ to the computation of $\mathtt{d}$, which in turn Turing-reduces to $\mathtt{h}$). The "natural" ("positive") meaning of a Turing-reduction of f' to f is that when given an algorithm for computing f we obtain an algorithm for computing f'. In contrast, the proof of Theorem 5 uses the "unnatural" ("negative") counter-positive: if (as we know) there exists no algorithm for computing $f' = \mathtt{d}'$ then there exists no algorithm for computing $f = \mathtt{h}$ (which is what we wanted to prove). Jumping ahead, we mention that resource-bounded Turing-reductions (e.g., polynomial-time reductions) play a central role in complexity theory itself, and again they are used mostly in a "negative" way. We will define such reductions and extensively use them in the rest of the course.

Rice's Theorem. The undecidability of the halting problem (or rather the fact that the function $\mathtt{d}$ is uncomputable) is a special case of a more general phenomenon: Every non-trivial decision problem *regarding the function computed by a given Turing machine* has no algorithmic solution. We state this fact next, clarifying what is the aforementioned class of problems. (Again, we refer to Turing machines that may not halt on all inputs.)

Theorem 6 (Rice's Theorem): *Let $\mathcal{F}$ be a non-trivial subset[6] of the set of all computable partial functions, and let $S_{\mathcal{F}}$ be the set of strings that describe machines that compute functions in $\mathcal{F}$. Then deciding membership in $S_{\mathcal{F}}$ cannot be solved by an algorithm.*

Theorem 6 can be proved by a Turing-reduction from $\mathtt{d}$. We do not provide a proof because this is too remote from the main subject matter of the course. We stress that Theorems 5 and 6 hold for any reasonable model of computation (referring both to the potential solvers and to the machines the description of which is given as input to these solvers). Thus, Theorem 6 means that *no algorithm can determine any non-trivial property of the function computed by a given computer program* (written in any programming language). For example, *no algorithm can determine whether or not a given computer program halt on each possible input.* The relevance of this assertion to the project of program verification is obvious.

The Post Correspondence Problem. We mention that undecidability arises also outside of the domain of questions regarding computing devices (given as input). Specifically, we consider the Post Correspondence Problem in which the input consists of two (equal length) sequences of strings, $(\alpha_1, ..., \alpha_k)$ and $(\beta_1, ..., \beta_k)$, and the question is whether or not there exists a sequence of indices $i_1, ..., i_\ell \in \{1, ..., k\}$ such that $\alpha_{i_1} \cdots \alpha_{i_\ell} = \beta_{i_1} \cdots \beta_{i_\ell}$. (We stress that the length of this sequence is not bounded.)[7]

Theorem 7 *The Post Correspondence Problem is undecidable.*

Again, the omitted proof is by a Turing-reduction from $\mathtt{d}$ (or $\mathtt{h}$).

3.3 Universal algorithms

So far we have used the postulate that, in any reasonable model of computation, each machine (or computation rule) has a finite description. Furthermore, we also used the fact that such model should allow for the easy modification of such descriptions such that the resulting machine computes an easily related function (see the proof of Theorem 5). Here we go one step further and postulate that the description of machines (in this model) is "effective" in the following natural sense: there exists an algorithm that, given a description of a machine (resp., computation rule)

[6]The set S is called a non-trivial subset of U if both S and $U \setminus S$ are non-empty. Clearly, if $\mathcal{F}$ is a trivial set of computable functions then the corresponding decision problem can be solved by a "trivial" algorithm that outputs the corresponding constant bit.

[7]In contrast, the existence of an adequate sequence of a specified length can be determined in time that is exponential in this length.

and a corresponding environment, determines the environment that results from performing a single step of this machine on this environment (resp. the effect of a single application of the computation rule). This algorithm can, in turn, be implemented in the said model of computation (assuming this model is general; see the Church-Turing Thesis). Successive applications of this algorithm leads to the notion of a universal machine, which (for concreteness) is formulated below in terms of Turing machines.

Definition 8 (universal machines): *A* universal Turing machine *is a Turing machine that on input a description of a machine M and an input x returns the value of $M(x)$ if M halts on x and otherwise does not halt.*

That is, a universal Turing machine computes the partial function $\mathtt{u}$ that is defined over pairs $(\langle M\rangle, x)$ such that M halts on input x, in which case it holds that $\mathtt{u}(\langle M\rangle, x) = M(x)$. We note that if M halts on all possible inputs then $\mathtt{u}(\langle M\rangle, x)$ is defined for every x. We stress that the mere fact that we have defined something does not mean that it exists. But, as hinted above and obvious to anyone who has written a computer program (and thought about what he/she was doing), universal Turing machines do exist.

Theorem 9 *There exists a universal Turing machine.*

Theorem 9 asserts that the partial function $\mathtt{u}$ is computable. In contrast, it can be shown that any extension of $\mathtt{u}$ to a total function is uncomputable. That is, for any total function $\hat{\mathtt{u}}$ that agrees with the partial function $\mathtt{u}$ on all the inputs on which the latter is defined, it holds that $\hat{\mathtt{u}}$ is uncomputable.[8]

Proof: Given a pair $(\langle M\rangle, x)$, we just emulate the computation of machine M on input x. This emulation is straightforward, because by the effectiveness of the description of M, we can iteratively determine the next instantaneous configuration of the computation of M on input x. If the said computation halts then we will obtain its output and can output it (and so, on input $(\langle M\rangle, x)$, our algorithm returns $M(x)$). Otherwise, we turn out emulating an infinite computation, which means that our algorithm does not halt on input $(\langle M\rangle, x)$. Thus, the foregoing emulation procedure constitutes a universal machine (i.e., yields an algorithm for computing $\mathtt{u}$). ■

As hinted above, the existence of universal machines is the fundamental fact underlying the paradigm of general-purpose computers. Indeed, a specific Turing machine (or algorithm) is a device that solves a specific problem. A priori, solving each problem would have required building a new physical device that allows for this problem to be solved in the physical world (rather than as a thought experiment). The existence of a universal machine asserts that it is enough to build one physical device; that is, a general purpose computer. Any specific problem can then be solved by writing a corresponding program for the general purpose computer. In other words, the existence of universal machines says that software can be viewed as (part of the) input.

In addition to their practical importance, the existence of universal machines (and their variants) has important consequences in the theories of computability and computational complexity. Here we merely note that Theorem 9 implies that many questions about the behavior of a universal

[8]The claim is easy to prove for the total function $\hat{\mathtt{u}}$ that extends $\mathtt{u}$ and assigns the special symbol $\perp$ to inputs on which $\mathtt{u}$ is undefined (i.e., $\hat{\mathtt{u}}(\langle M\rangle, x) \stackrel{\text{def}}{=} \perp$ if $\mathtt{u}$ is not defined on $(\langle M\rangle, x)$ and $\hat{\mathtt{u}}(\langle M\rangle, x) \stackrel{\text{def}}{=} \mathtt{u}(\langle M\rangle, x)$ otherwise). In this case $\mathtt{h}(\langle M\rangle, x) = 1$ if and only if $\hat{\mathtt{u}}(\langle M\rangle, x) \neq \perp$, and so the halting function $\mathtt{h}$ is Turing-reducible to $\hat{\mathtt{u}}$. In the general case, we may adapt the proof of Theorem 5 by observing that, for a machine M that halts on every input, it holds that $\hat{\mathtt{u}}(\langle M\rangle, x) = \mathtt{u}(\langle M\rangle, x)$ for every x (and in particular for $x = \langle M\rangle$).

machine on certain input types are undecidable. In particular, there is no algorithm that, given $X \stackrel{\text{def}}{=} (\langle M \rangle, x)$, can tell whether or not a (fixed) universal machine halts on input X. Revisiting the proof of Theorem 7, it follows that the Post Correspondence Problem remains undecidable even if the sequences are restricted to have a specific length (i.e., k is fixed).

A detour: Kolmogorov Complexity. The existence of universal machines, which may be viewed as universal languages for writing effective and succinct descriptions of objects, plays a central role in Kolmogorov Complexity. Loosely speaking, the latter theory is concerned with the length of (effective) descriptions of objects, and views the minimum such length as the inherent "complexity" of the object; that is, "simple" objects (or phenomena) are those having short description (resp., short explanation), whereas "complex" objects have no short description. Needless to say, these (effective) descriptions have to refer to some fixed "language" (i.e., to a fixed machine that, given a succinct description of an object, produces its explicit description). Fixing any machine M, a string x is called a description of s with respect to M if $M(x) = s$. The complexity of s with respect to M, denoted $K_M(s)$, is the length of the shortest description of s with respect to M. Certainly, we want to fix M such that every string has a description with respect to M, and furthermore that this description is not "significantly" longer than the description with respect to a different machine M'. The following theorem make it natural to use a universal machine as the "point of reference" (i.e., the aforementioned M).

Theorem 10 (complexity w.r.t a universal machine): *Let U be a universal machine. Then, for every machine M', there exists a constant c such that $K_U(s) \leq K_{M'}(s) + c$ for every string s.*

The theorem follows by (setting $c = O(|\langle M' \rangle|)$ and) observing that if x is a description of s with respect to M' then $(\langle M' \rangle, x)$ is a description of s with respect to U. Here it is important to use an adequate encoding of pairs of strings (e.g., the pair $(\sigma_1 \cdots \sigma_k, \tau_1 \cdots \tau_\ell)$ is encoded by the string $\sigma_1\sigma_1 \cdots \sigma_k\sigma_k 01\tau_1 \cdots \tau_\ell)$. Fixing any universal machine U, we define the Kolmogorov Complexity of a string s as $K(s) \stackrel{\text{def}}{=} K_U(s)$. The reader may easily verify the following facts:

1. $K(s) \leq |s| + O(1)$, for every s.

   (Hint: apply Theorem 10 to the machine that computes the identity mapping.)

2. There exist infinitely many strings s such that $K(s) \ll |s|$.

   (Hint: consider $s = 1^n$. Alternatively, consider any machine M such that $|M(x)| \gg |x|$.)

3. Some strings of length n have complexity at least n. Furthermore, for every n and i,

   $$|\{s \in \{0,1\}^n : K(s) \leq n - i\}| < 2^{n-i+1}$$

   (Hint: different strings must have different descriptions with respect to U.)

It can be shown that *the function K is uncomputable.* The proof is related to the paradox captured by the following "description" of a natural number: `the largest natural number that can be described by an English sentence of up-to a thousand letters`. (The paradox amounts to observing that if the above number is well-defined then so is `the integer-successor of the largest natural number that can be described by an English sentence of up-to a thousand letters`.) Needless to say, the above sentence presupposes that any sentence is a legitimate description in some adequate sense (e.g., in the sense defined above). Specifically, the above sentence presupposes that we can determine the Kolmogorov Complexity of each natural number, and furthermore effectively produce the largest number that has Kolmogorov Complexity not exceeding some threshold. Indeed, the paradox provides a proof to the fact that the latter task cannot be performed (i.e., there exists no algorithm that given t produces the lexicographically last string s such that $K(s) \leq t$, because if such an algorithm A would have existed then $K(s) \leq O(|\langle A \rangle|) + \log t$ and $K(s0) < K(s) + O(1) < t$ in contradiction to the definition of s).

3.4 Time and space complexity

Fixing a model of computation (e.g., Turing machines) and focusing on algorithms that halt on each input, we consider the number of steps (i.e., applications of the computation rule) taken by the algorithm on each possible input. The latter function is called the time complexity of the algorithm (or machine); that is, $t_A : \{0,1\}^* \to \mathbb{N}$ is called the time complexity of algorithm A if, for every x, on input x algorithm A halts after exactly $t_A(x)$ steps.

We will be mostly interested in the dependence of the time complexity on the input length, when taking the maximum over all inputs of the relevant length. That is, for t_A as above, we will consider $T_A : \mathbb{N} \to \mathbb{N}$ defined by $T_A(n) \stackrel{\rm def}{=} \max_{x \in \{0,1\}^n} \{t_A(x)\}$. Abusing terminology, we sometimes refer to T_A as the time complexity of A.

The time complexity of a problem. As stated in the preface and in the introduction, complexity theory is typically unconcerned with the (time) complexity of a specific algorithm. It is rather concerned with the (time) complexity of a problem, assuming that this problem is solvable by an algorithm. Intuitively, the time complexity of such a problem is defined as the time complexity of the fastest algorithm that solves this problem (assuming that the latter term is well-defined).[9] More generally, we will be interested in upper and lower bounds on the (time) complexity of algorithms that solve the problem. However, the complexity of a problem may depend on the specific model of computation in which algorithms that solve it are implemented. The following Cobham-Edmonds Thesis asserts that the variation (in the time complexity) is not too big, and in particular is irrelevant for much of the current focus of complexity theory (e.g., for the P-vs-NP Question).

The Cobham-Edmonds Thesis. As stated above, the time complexity of a problem may depend on the model of computation. For example, deciding membership in the set $\{xx : x \in \{0,1\}^*\}$ can be done in linear-time on a two-tape Turing machine, but requires quadratic-time on a single-tape Turing machine.[10] On the other hand, any problem that has time complexity t in the model of multi-tape Turing machines, has complexity $O(t^2)$ in the model of single-tape Turing machines. The Cobham-Edmonds Thesis asserts that the time complexities in any two "reasonable and general" models of computation are polynomially related. That is, *a problem has time complexity t in*

[9] **Advanced comment:** Actually (see "Borodin's Gap Theorem" and "Blum's Speed-up Theorem" in [2, Sec. 12.6]), the naive assumption that a "fastest algorithm" for solving a problem exists is not always justified. On the other hand, the assumption is justified in some important cases (see, e.g., "Levin's optimal search algorithm" [5]).

[10] Proving the latter fact is quite non-trivial. One proof is by a "reduction" from a communication complexity problem [4, Sec. 12.2]. Intuitively, a single-tape Turing machine that decides membership in the aforementioned set can be viewed as a channel of communication between the two parts of the input. Focusing our attention on inputs of the form $y0^n z0^n$, for $y, z \in \{0,1\}^n$, each time the machine passes from the first part to the second part it carries $O(1)$ bits of information (in its internal state) while making at least n steps. The proof is completed by invoking the linear lower bound on the communication complexity of the (two-argument) identity function (i.e, $\mathtt{id}(y,z) = 1$ if $y = z$ and $\mathtt{id}(y,z) = 0$ otherwise, cf. [4, Chap. 1]).

some "reasonable and general" *model of computation if and only if it has time complexity* poly(t) *in the model of* (single-tape) *Turing machines.*

Indeed, the Cobham-Edmonds Thesis strengthens the Church-Turing Thesis. It asserts not only that the class of solvable problems is invariant as far as "reasonable and general" models of computation are concerned but also that the time complexity (of the solvable problems) in such models be polynomially related.

Efficient algorithms. As hinted above, much of complexity theory is concerned with efficient algorithms. The latter are defined as polynomial-time algorithms (i.e., algorithms that have a time complexity that is bounded by a polynomial in the length of the input). By the Cobham-Edmonds Thesis, the choice of a "reasonable and general" model of computation is irrelevant to the definition of this class. The association of efficient algorithms with polynomial-time computation is grounded in the following two considerations:

- *Philosophical consideration*: Intuitively, efficient algorithms are those that can be implemented within a number of steps that is a moderately growing function of the input length. To allow for reading the entire input, at least linear time complexity should be allowed, whereas exponential time (as in "exhaustive search") must be avoided. Furthermore, a good definition of the class of efficient algorithms should be closed under natural composition of algorithms (as well as be robust with respect to reasonable models of computation and with respect to simple changes in the encoding of problems' instances).

  Selecting polynomials as the set of time-bounds for efficient algorithms satisfy all the above requirements: polynomials constitute a "closed" set of moderately growing functions, where "closure" means closure under addition, multiplication and functional composition. These closure properties guarantee the closure of the class of efficient algorithm under natural composition of algorithms (as well as its robustness with respect to any reasonable and general model of computation). Furthermore, polynomial-time algorithms can conduct computations that are definitely simple (although not totally trivial), and on the other hand they do not include naturally inefficient algorithms like exhaustive search.

- *Empirical consideration*: It is clear that algorithms that are considered efficient in practice have running-time that is bounded by a small polynomial (at least on the inputs that occur in practice). The question is whether any polynomial-time algorithm can be considered efficient in an intuitive sense. The belief, which is supported by past experience, is that every natural problem that can be solved in polynomial-time also has "reasonably efficient" algorithms.

We stress that the association of efficient algorithms with polynomial-time computation is not essential to most of the notions, results and questions of complexity theory. Any other class of algorithms that supports the aforementioned closure properties and allows to conduct some simple computations but not overly complex ones gives rise to a similar theory, albeit the formulation of such a theory may be much more complicated. Specifically, all results and questions treated in this course relate the complexity of different computational tasks (rather than provide absolute assertions about the complexity of some computational tasks). These relations can be stated explicitly, by stating how any upper-bound on the time complexity of one task gets translated to an upper-bound on the time complexity of another task. Such cumbersome statements will maintain the contents of the standard statements; they will merely be much more complicated. Thus, we follow the tradition of focusing on polynomial-time computations, stressing that this focus is both natural and provides the simplest way of addressing the fundamental issues underlying the nature of efficient computation.

Universal machines, revisited. Time complexity yields an important variant of the universal function $\mathtt{u}$ (presented in Section 3.3). Define $\mathtt{u}'(\langle M\rangle, x, t) \stackrel{\text{def}}{=} y$ if on input x machine M halts within t steps and outputs the string y, and $\mathtt{u}'(\langle M\rangle, x, t) \stackrel{\text{def}}{=} \perp$ if on input x machine M makes more than t steps. Unlike $\mathtt{u}$, the function $\mathtt{u}'$ is computable. Furthermore, $\mathtt{u}'$ is computable by a machine U' that on input $X = (\langle M\rangle, x, t)$ halts after $\text{poly}(t)$ steps. Indeed, machine U' is a variant of a universal machine (i.e., on input X, machine U' merely emulates M for t steps rather than forever). Note that the number of steps taken by U' depends on the specific model of computation (and that some overhead is unavoidable because emulating each step of M requires reading the relevant portion of the description of M).

Space complexity. Another natural measure of the "complexity" of an algorithm (or a task) is the amount of memory consumed by the computation. We refer to the memory used for storing some intermediate results of the computation. Since much of our focus will be on using memory that is sub-linear in the input length, it is important to use a model in which one can differentiate memory used for computation from memory used for storing the initial input or the final output. In the context of Turing machines, this is done by considering multi-tape Turing machines such that the input is presented on a special read-only tape (called the input tape), the output is written on a special write-only tape (called the output tape), and intermediate results are stored on a work-tape. Thus, the input and output tapes cannot be used for storing intermediate results. The space complexity of such a machine M is defined as a function s_M such that $s_M(x)$ is the number of cells of the work-tape scanned by M on input x.

3.5 Oracle machines

The notion of Turing-reductions, which was discussed in Section 3.2, is captured by the definition of oracle machines. Loosely speaking, an oracle machine is a machine that is augmented such that it may pose questions to the outside. (A rigorous formulation of this notion is provided below.) We consider the case in which these questions, called queries, are answered consistently by some function $f : \{0,1\}^* \to \{0,1\}^*$, called the oracle. That is, if the machine makes a query q then the answer it obtains is $f(q)$. In such a case, we say that the oracle machine is given access to the oracle f. For an oracle machine M, a string x and a function f, we denote by $M^f(x)$ the output of M on input x when given access to the oracle f. (Re-examining the second part of the proof of Theorem 5, observe that we have actually described an oracle machine that computes $\mathtt{h}$ when given access to the oracle d'.)

The notion of an oracle machine extends the notion of a standard computing device (machine), and thus a rigorous formulation of the former extends a formal model of the latter. Specifically, extending the model of Turing machines, we derive the following model of oracle Turing machines.

Definition 11 (using an oracle): *An* oracle machine *is a Turing machine with an additional tape, called the* oracle tape*, and two special states, called* oracle invocation *and* oracle spoke*. The* computation of the oracle machine M on input x and access to the oracle $f : \{0,1\}^* \to \{0,1\}^*$ *is defined based on the successive configuration function. For configurations with state different from* oracle invocation *the next configuration is defined as usual. Let γ be a configuration in which the machine's state is* oracle invocation *and suppose that the actual contents of the oracle tape is q (i.e., q is the contents of the maximal prefix of the tape that holds bit values).*[11] *Then, the configuration*

[11] A common convention is that the oracle can be invoked only when the machine's head resides at the left-most cell of the oracle tape. We comment that, in the context of space complexity, one uses two oracle tapes: a write-only

following γ is identical to γ, except that the state is oracle spoke*, and the actual contents of the oracle tape is $f(q)$. The string q is called M's* query *and $f(q)$ is called the* oracle's reply*.*

We stress that the running time of an oracle machine is the number of steps made during its computation, and that the oracle's reply on each query is obtained in a single step.

3.6 Restricted models

We mention that restricted models of computation are often mentioned in the context of a course on computability, but they will play no role in the current course. One such model is the model of finite automata, which in some variant coincides with Turing machines that have space complexity zero.

In our opinion, the most important motivation for the study of these restricted models of computation is that they provide simple models for some natural (or artificial) phenomena. This motivation, however, seems only remotely related to the study of the complexity of various computational tasks. Thus, in our opinion, the study of these restricted models (e.g., any of the lower levels of Chomsky's Hierarchy [2, Chap. 9]) should be decoupled from the study of computability theory (let alone the study of complexity theory).

4 Non-uniform Models (Circuits and Advice)

By a non-uniform model of computation we mean a model in which for each possible input length one considers a different computing device. That is, there is no "uniformity" requirement relating devices that correspond to different input lengths. Furthermore, this collection of devices is infinite by nature, and (in absence of a uniformity requirement) this collection may not even have a finite description. Nevertheless, each device in the collection has a finite description. In fact, the relationship between the size of the device (resp., the length of its description) and the length of the input that it handles will be of major concern. The hope is that the finiteness of all parameters (which refer to a single device in such a collection) will allow for the application of combinatorial techniques to analyze the limitations of certain settings of parameters.

In complexity theory, non-uniform models of computation are studied either towards the development of lower-bound techniques or as simplified upper-bounds on the ability of efficient algorithms. In both cases, the uniformity condition is eliminated in the interest of simplicity and with the hope (and belief) that nothing substantial is lost as far as the questions in focus are concerned.

We will focus on two related models of non-uniform computing devices: Boolean circuits (Section 4.1) and "machines that take advice" (Section 4.2). The former model is more adequate for the study of the evolution of computation (i.e., development of lower-bound techniques), whereas the latter is more adequate for modeling purposes (e.g., upper-bounding the ability of efficient algorithms).

4.1 Boolean Circuits

The most popular model of non-uniform computation is the one of Boolean circuits. Historically, this model was introduced for the purpose of describing the "logic operation" of real-life electronic circuits. Ironically, nowadays this model provides the stage for some of the most practically removed

tape for the query and a read-only tape for the answer.

studies in complexity theory (which aim at developing methods that may eventually lead to an understanding of the inherent limitations of efficient algorithms).

A Boolean circuit is a directed acyclic graph *with labels on the vertices*, to be discussed shortly. For sake of simplicity, we disallow isolated vertices (i.e., vertices with no in-going or out-going edges), and thus the graph vertices are of three types: *sources*, *sinks*, and *internal vertices*.

1. Internal vertices are vertices having in-coming and out-going edges (i.e., they have in-degree and out-degree at least 1). In the context of Boolean circuits, internal vertices are called gates. Each gate is labeled by a Boolean operation, where the operations typically considered are $\wedge$, $\vee$ and $\neg$ (corresponding to `and`, `or` and `neg`). In addition, we require that gates labeled $\neg$ have in-degree 1. (The in-coming degree of $\wedge$-gates and $\vee$-gates may be any number greater than zero, and the same holds for the out-degree of any gate.)

2. The graph sources (i.e., vertices with no in-going edges) are called input terminals. Each input terminal is labeled by a natural number (which is to be thought of the index of an input variable). (For sake of defining formulae, we allow different input terminals to be labeled by the same number.)[12]

3. The graph sinks (i.e., vertices with no out-going edges) are called output terminals, and we require that they have in-degree 1. Each output terminal is labeled by a natural number such that if the circuit has m output terminals then they are labeled $1, 2, ..., m$. That is, we disallow different output terminals to be labeled by the same number, and insist that the labels of the output terminals are consecutive numbers. (Indeed, the labels of the output terminals will correspond to the indices of locations in the circuit's output.)

For sake of simplicity, we also mandate that the labels of the input terminals are consecutive numbers.[13]

A Boolean circuit with n different input labels and m output terminals induces (and indeed computes) a function from $\{0,1\}^n$ to $\{0,1\}^m$ defined as follows. For any fixed string $x \in \{0,1\}^n$, we iteratively define the value of vertices in the circuit such that the input terminals are assigned the corresponding bits in $x = x_1 \cdots x_n$ and the values of other vertices are determined in the natural manner. That is:

- An input terminal with label $i \in \{1, ..., n\}$ is assigned the i^{th} bit of x (i.e., the value x_i).

- If the children of a gate (of in-degree d) labeled $\wedge$ have values $v_1, v_2, ..., v_d$ then the gate is assigned the value $\wedge_{i=1}^{d} v_i$. The value of a gate labeled $\vee$ (or $\neg$) is determined analogously.

  Indeed, the hypothesis that the circuit is acyclic implies that the process of determining values for the circuit's vertices is well-defined: As long as the value of some vertex is undetermined, there exists a vertex such that its value is undetermined but the values of all its children are determined. Thus, the process can make progress, and terminates when the values of all vertices (including the output terminals) are determined.

[12] This is not needed in case of general circuits, because we can just feed out-going edges of the same input terminal to many gates. Note, however, that this is not allowed in case of formulae, where all non-sinks are required to have out-degree 1.

[13] This convention slightly complicates the construction of circuits that ignore some of the input values. Specifically, we use artificial gadgets that have in-coming edges from the corresponding input terminals, and compute an adequate constant. To avoid having this constant as an output terminal, we feed it into an auxiliary gate such that the value of the latter is determined by the other in-going edge (e.g., a constant 1 fed into an $\vee$-gate). See example of dealing with x_3 in Figure 2.

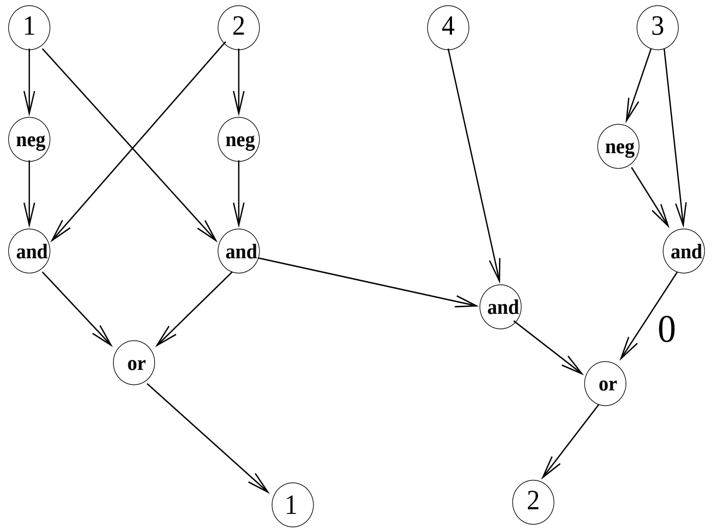

Figure 2: A circuit computing $f(x_1, x_2, x_3, x_4) = (x_1 \oplus x_2, x_1 \wedge \neg x_2 \wedge x_4)$.

The value of the circuit on input x (i.e., the output computed by the circuit on input x) is $y = y_1 \cdots y_m$, where y_i is the value assigned by the above process to the output terminal labeled i. We note that *there exists a polynomial-time algorithm that, given a circuit C and a corresponding input x, outputs the value of C on input x.* This algorithm determines the values of the circuit's vertices, going from the circuit's input terminals to its output terminals.

We say that a family of circuits $(C_n)_{n\in\mathbb{N}}$ computes a function $f : \{0,1\}^* \to \{0,1\}^*$ if for every n the circuit C_n computes the restriction of f to strings of length n. In other words, for every $x \in \{0,1\}^*$, it must hold that $C_{|x|}(x) = f(x)$.

Bounded and unbounded fan-in. We will be most interested in circuits in which each gate has at most two in-coming edges. In this case, the types of (two-argument) Boolean operations that we allow is immaterial (as long as we consider a "full basis" of such operations; i.e., a set of operations that can implement any other two-argument Boolean operation). Such circuits are called circuits of bounded fan-in. In contrast, other studies are concerned with circuits of unbounded fan-in, where each gate may have an arbitrary number of in-going edges. Needless to say, in the case of circuits of unbounded fan-in, the choice of allowed Boolean operations is important and one focuses on operations that are "uniform" (across the number of operants; e.g., $\wedge$ and $\vee$).

Circuit size as a complexity measure. The size of a circuit is the number of its edges. When considering a family of circuits $(C_n)_{n\in\mathbb{N}}$ that computes a function $f : \{0,1\}^* \to \{0,1\}^*$, we are interested in the size of C_n as a function of n. Specifically, we say that this family has size complexity $s : \mathbb{N} \to \mathbb{N}$ if for every n the size of C_n is $s(n)$. The circuit complexity of a function f, denoted s_f, is the infimum of the size complexity of all families of circuits that compute f. Alternatively, for each n we may consider the size of the smallest circuit that computes the restriction of f to n-bit strings (denoted f_n), and set $s_f(n)$ accordingly. We stress that non-uniformity is implicit in this definition, because no conditions are made regarding the relation between the various circuits used to compute the function on different input lengths.

The circuit complexity of functions. We highlight some simple facts about the circuit complexity of functions. (These facts are in clear correspondence to facts regarding Kolmogorov Complexity mentioned in Section 3.3.)

1. Most importantly, any Boolean function can be computed by some family of circuits, and thus the circuit complexity of any function is well-defined. Furthermore, each function has at most exponential circuit complexity.

   (Hint: $f_n : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit of size $O(n2^n)$ that implements a look-up table.)

2. Some functions have polynomial circuit complexity. In particular, any function that has time complexity t (i.e., is computed by an algorithm of time complexity t) has circuit complexity $\text{poly}(t)$. Furthermore, the corresponding circuit family is uniform (in a natural sense to be discussion below).

   (Hint: consider a Turing machine that computes the function, and consider its computation on a generic n-bit long input. The corresponding computation can be emulated by a circuit that consists of $t(n)$ layers such that each layer represents an instantaneous configuration of the machine, and the relation between consecutive configurations is captured by ("uniform") local gadgets in the circuit.)

3. Almost all Boolean functions have exponential circuit complexity. Specifically, the number of functions mapping $\{0,1\}^n$ to $\{0,1\}$ that can be computed by a circuit of size s is at most s^{2s}.

   (Hint: the number of circuits having v vertices and s edges is at most $\binom{v}{2}^s$.)

Note that the first fact implies that families of circuits can compute functions that are uncomputable by algorithms. Furthermore, this phenomenon occurs also when restricting attention to families of polynomial-size circuits. See further discussion in Section 4.2.

Uniform families. A family of polynomial-size circuits $(C_n)_{n\in\mathbb{N}}$ is called uniform if given n one can construct the circuit C_n in $\text{poly}(n)$-time. Note that *if a function is computable by a uniform family of polynomial-size circuits then it is computable by a polynomial-time algorithm.* The algorithm first constructs the adequate circuit (which can be done in polynomial-time by the uniformity hypothesis), and then evaluate this circuit on the given input (which can be done in time that is polynomial in the size of the circuit).

Note that limitations on the computing power of arbitrary families of polynomial-size circuits certainly hold for uniform families (of polynomial-size), which in turn yield limitations on the computing power of polynomial-time algorithms. Thus, lower bounds on the circuit complexity of functions yield analogous lower bounds on their time complexity. Furthermore, as is often the case in mathematics and Science, disposing of an auxiliary condition that is not well-understood (i.e., uniformity) may turn out fruitful. Indeed, this has occured in the study of limited classes of circuits, which is reviewed in Section 4.3.

4.2 Machines that take advice

General (non-uniform) circuit families and uniform circuit families are two extremes with respect to the "amounts of non-uniformity" in the computing device. Intuitively, in the former, non-uniformity is only bounded by the size of the device, whereas in the latter the amounts of non-uniformity is zero. Here we consider a model that allows to decouple the size of the computing device from the amount of non-uniformity, which may range from zero to the device's size. Specifically, we consider algorithms that "take a non-uniform advice" that depends only on the input length. The amount of non-uniformity will be defined to equal the length of the corresponding advice (as a function of the input length).

Definition 12 (taking advice): *We say that* algorithm A computes the function f using advice of length $\ell : \mathbb{N} \to \mathbb{N}$ *if there exists an infinite sequence* $(a_n)_{n\in\mathbb{N}}$ *such that*

1. *For every* $x \in \{0,1\}^*$*, it holds that* $A(a_{|x|}, x) = f(x)$*.*
2. *For every* $n \in \mathbb{N}$*, it holds that* $|a_n| = \ell(n)$*.*

The sequence $(a_n)_{n\in\mathbb{N}}$ *is called the* advice sequence*.*

Note that any function having circuit complexity s can be computed using advice of length $O(s \log s)$, where the log factor is due to the fact that a graph with v vertices and e edges can be described by a string of length $2e \log_2 v$. Note that the model of machines that use advice allows for some sharper bounds than the ones stated in Section 4.1: every function can be computed using advice of length ℓ such that $\ell(n) = 2^n$, and some uncomputable functions can be computed using advice of length 1.

Theorem 13 (the power of advice): *There exist functions that can be computed using one-bit advice but cannot be computed without advice.*

Proof: Taking any uncomputable Boolean function $f : \mathbb{N} \to \{0,1\}$, consider the function f' defined as $f'(x) = f(|x|)$. Note that f is Turing-reducible to f' (e.g., on input n make any n-bit query to f', and return the answer).[14] Thus, f' cannot be computed without advice. On the other hand, f' can be easily computed by using the advice sequence $(a_n)_{n\in\mathbb{N}}$ such that $a_n = f(n)$; that is, the algorithm merely outputs the advice bit (and indeed $a_{|x|} = f(|x|) = f'(x)$, for every $x \in \{0,1\}^*$). ■

4.3 Restricted models

As noted in Section 4.1, the model of Boolean circuits allows for the introduction of many natural subclasses of computing devices. Following is a laconic review of a few of these subclasses. We will refer to various types of Boolean formulae in the rest of this course, and thus suggest not to skip the following two paragraphs.

Boolean formulae. In general Boolean circuits the non-sink vertices are allowed arbitrary out-degree. This means that the same intermediate value can be re-used (without being re-computed (while increasing the size complexity by only one unit)). Such "free" re-usage of intermediate values is disallowed in Boolean formulae, which corresponds to a Boolean expression over Boolean variables. Formally, a Boolean formula is a circuit in which all non-sink vertices have out-degree 1, which means that the underlying graph is a tree (and the formula as an expression can be read by traversing the tree scanning the leaves in order). Indeed, we have allowed different input terminals to be assigned the same label in order to allow formulae in which the same variable occurs multiple times. As in case of general circuits, one is interested in the size of these restricted circuits (i.e., the size complexity of families of formulae computing various functions). We mention that quadratic lower bounds are known for the formula size of simple functions (e.g., `parity`), whereas these functions have linear circuit complexity.

[14] Indeed, this Turing-reduction is not efficient (i.e., it runs in exponential time in $|n| = \log_2 n$), but this is immaterial in the current context.

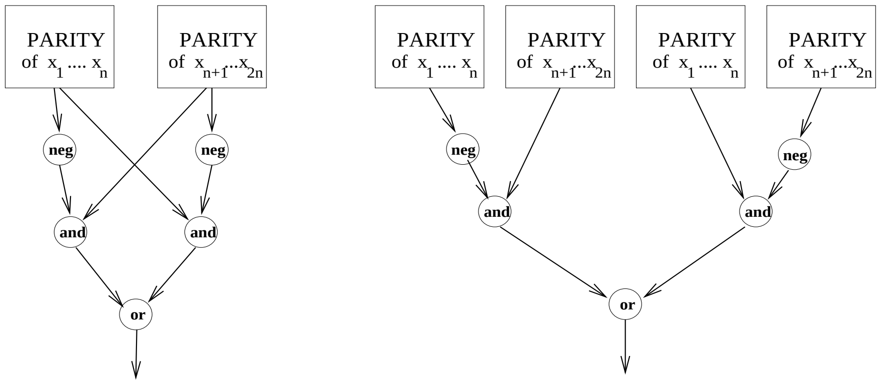

Figure 3: Recursive construction of parity circuits and formulae.

Formulae in CNF and DNF. A restricted type of Boolean formulae consists of formulae that are in conjunctive normal form (CNF). Such a formula consists of a conjunction of clauses, where each clause is a disjunction of literals each being either a variable or its negation. That is, such formulae are represented by layered circuits of unbounded fan-in in which the first layer consists of `neg`-gates that compute the negation of input variables, the second layer consist of `or`-gates that compute the logical-or of subsets of inputs and negated inputs, and the third layer consists of a single `and`-gate that computes the logical-and of the values computed in the second layer. Note that each Boolean function can be computed by a family of CNF formulae of exponential size, and that the size of CNF formulae may be exponentially larger than the size of ordinary formulae computing the same function (e.g., `parity`). For a constant k, a formula is said to be in k-CNF if its CNF has disjunctions of size at most k. An analogous restricted type of Boolean formulae refers to formulae that are in disjunctive normal form (DNF). Such a formula consists of a disjunction of a conjunctions of literals, and when each conjunction has at most k literals we say that the formula is in k-DNF.

Constant-depth circuits. Circuits have a "natural structure" (i.e., their structure as graphs). One natural parameter regarding this structure is the depth of a circuit, which is defined as the longest directed path from any source to any sink. Of special interest are constant-depth circuits of unbounded fan-in. We mention that sub-exponential lower bounds are known for the size of such circuits that compute a simple function (e.g., `parity`).

Monotone circuits. The circuit model also allows for the consideration of monotone computing devices: a monotone circuit is one having only monotone gates (e.g., gates computing $\wedge$ and $\vee$, but no negation gates (i.e., $\neg$-gates)). Needless to say, monotone circuits can only compute monotone functions, where a function f is called monotone if for any $x < y$ it holds that $f(x) \leq f(y)$ (where we refer to the lexicographic order on strings). One natural question is whether, as far as monotone functions are concerned, there is a substantial loss in using only monotone circuits. The answer is *yes*: there exist monotone functions that have polynomial circuit complexity but require sub-exponential size monotone circuits.

Notes

It is quite remarkable that the theories of uniform and non-uniform computational devices have emerged in two single papers. We refer to Turing's paper [10], which introduced the model of Turing machines, and to Shannon's paper [8], which introduced Boolean circuits.

In addition to introducing the Turing machine model and arguing that it corresponds to the intuitive notion of computability, Turing's paper [10] introduces universal machines and contains proofs of undecidability (e.g., of the Halting Problem). Rice's Theorem is proven in [7], and the undecidability of the Post Correspondence Problem is proven in [6].

The formulation of machines that take advice (as well as the equivalence to the circuit model) originates in [3].

References

[1] O. Goldreich. On Promise Problems (a survey in memory of Shimon Even [1935-2004]). *ECCC*, TR05-018, 2005.

[2] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[3] R.M. Karp and R.J. Lipton. Some connections between nonuniform and uniform complexity classes. In *12th ACM Symposium on the Theory of Computing*, pages 302-309, 1980.

[4] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1996.

[5] L.A. Levin. Universal Search Problems. *Problemy Peredaci Informacii 9*, pages 115–116, 1973. Translated in *problems of Information Transmission 9*, pages 265–266.

[6] E. Post. A Variant of a Recursively Unsolvable Problem. *Bull. AMS*, Vol. 52, pages 264–268, 1946.

[7] H.G. Rice. Classes of Recursively Enumerable Sets and their Decision Problems. *Trans. AMS*, Vol. 89, pages 25–59, 1953.

[8] C.E. Shannon. A Symbolic Analysis of Relay and Switching Circuits. *Trans. American Institute of Electrical Engineers*, Vol. 57, pages 713–723, 1938.

[9] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[10] C.E. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. Londom Mathematical Soceity*, Ser. 2, Vol. 42, pages 230–265, 1936. A Correction, *ibid.*, Vol. 43, pages 544–546.