Lecture Notes on Computational Complexity

(part of the course Efficient Algorithms and Problem Complexity)

These lecture notes give a very rough introduction to some of the most basic notions of computational complexity and their connections to each other. Every student is recommended to read a good book on the topic, such as [Pap94], [Sip97] or at least the corresponding sections of the course textbook [JS04].

1 Introduction

Solving a computational problem requires resources of various descriptions. Often, we just want to solve the problem at hand as quickly as possible, i.e., we focus on the resource time. However, we may also want to save memory space, wish to find a parallel solution that does not make use of an excessive number of processors, solve the problem in a distributed manner with little communication overhead, or maybe solve it using a short program.

Whatever type of resource we intend to consider, the first question to answer is how to formalize this resource. Since we want to study problems and their solutions in general, abstract notions are needed. This becomes most obvious if the resource is time, because physical time is something that does not exist in the abstract mathematical world of computational problems and their algorithmic solutions. However, once we fix a model of computation that comes with a natural notion of single atomic computation steps, such as the Turing machine, we can simply count steps. Hence, one computation step is considered to take one unit of time, whatever that unit is, and the length of a computation – the number of steps it consists of – is regarded as the time consumed by that computation. This is certainly an idealization, because the execution time of computation steps on a real computer may vary between different types of instructions. However, it is a reasonable assumption that they do not differ arbitrarily, which means that the idealization only disregards a constant factor that, furthermore, is unknown and varies between different CPUs and computer architectures.

The running time of an algorithm is usually not a constant, but depends on the concrete input. However, measuring it in terms of the input itself would be much too fine grained to be useful. It makes much more sense to measure it in terms of the input size. Thus, we are looking for a function \( \rho : \mathbb{N} \rightarrow \mathbb{N} \) that, for every input size \( n \), returns the number \( \rho(n) \) of steps required for inputs of that size. One of the obvious advantages of such functions is that they can be composed, which yields a means to combine running times of algorithms, e.g., if one algorithm is applied to the output of another. Unfortunately, the abstraction from the concrete input creates a new problem, namely how to define the running time required for inputs of size \( n \). After all, the different inputs of the same size may lead to different running times. Possible answers are: take the maximum, the average, or the minimum over all inputs of that size.

The by far most common answer is to take the maximum, i.e., to consider the worst case behaviour of algorithms. There are several good reasons for this:
• Considering the minimum (thus making a best case analysis) seldom yields interesting insights, because the best cases are often related to trivial inputs for which an algorithm can immediately return the answer. If we want to know how hard a problem is, we should not focus on the trivial problem instances.

• A worst case analysis guarantees that the execution of the algorithm will never take more than $\rho(n)$ steps, which most of the time is the information one wants to have.

• In comparison to an average case analysis, statements about the worst case behaviour of algorithms or the worst case complexity of problems are easy to prove.

• For an average case analysis to be possible, assumptions must be made regarding the input distribution. If the assumptions do not match the concrete application situation, the analysis is useless for that particular situation. In fact, in concrete applications the input distribution is often unknown, which means that no average case analysis can be put to use.

Above, it was argued that running time should be measured as a function of the input size, because looking at the input itself yields an analysis too fine grained to be (possible and) useful. However, it quickly becomes clear that even this is too precise. Since we cannot reasonably deal with constant factors, we should not differentiate between a running time $\rho(n)$ as above and another running time $\rho'(n) = c \cdot \rho(n)$, where $c$ is any positive constant. Not only does this account for the fact that constant factors usually vary (and are, therefore, unknown), it also simplifies many arguments and allows us to consider classes of problems of “the same” complexity. The mathematical formalization of how to disregard constant factors leads to the big-O notation and its relatives, which will briefly be recalled below.

Based on this, the area of computational complexity tries to classify problems with respect to their required resource consumption. The usual pattern is to consider a certain type $A$ of algorithms (e.g., deterministic, nondeterministic, parallel, randomized, . . . ), a resource $R$ (e.g., time, space, number of processors, . . . ), and a class $C$ of functions $\rho: \mathbb{N} \to \mathbb{N}$ (e.g., the class of all polynomials in one variable). Then the complexity class of problems determined by these three parameters consists of all problems that can be solved by an $A$-algorithm whose $R$-consumption is bounded by some $\rho \in C$. For example, choosing $R$ to be time and $C$ to be the set of all polynomials, we obtain the class $P$ if $A$ is the set of deterministic algorithms and $NP$ if it is the set of nondeterministic algorithms. (See below for the definitions of $P$ and $NP$.)

\section*{2 Some Basic Notation}

To avoid confusion, we collect some basic notation here. We will denote the set of natural numbers (including 0) by $\mathbb{N}$. If $A$ is a set, the set of all finite sequences (or strings) over $A$ is denoted by $A^*$. The length of a string $u$ is denoted by $|u|$. The composition of two functions $f: A \to B$ and $g: B \to C$ is denoted by $g \circ f$. Thus, $(g \circ f)(x) = g(f(x))$ for all $x \in A$.

---

1 Unfortunately, speaking in absolute terms rather than relative ones, worst case complexity is nevertheless in many respects an extremely difficult subject to study.

2 It should be emphasized that this discussion refers to deterministic algorithms and their average case analysis. An entirely different picture is obtained by looking at randomized algorithms that have a built-in ability to flip coins. Randomized complexity is a very interesting subject, and as the example of randomized quicksort shows, randomization has highly useful practical applications.
3 Problems and Algorithms

If we want to study the complexity of computational problems, we must define what a problem is, and we also have to agree on a formalism in which algorithms can be formulated. In the following, a problem is a function $A : I \to O$ that maps a set $I$ of problem instances to a set $O$ of outputs. A decision problem is the special case in which $O = \{\text{yes}, \text{no}\}$. Such a problem is often identified with a set, namely the set $\{I \in I \mid A(I) = \text{yes}\}$ of so-called yes-instances of the problem. (In mathematical terms, this is the set of which $A$ is the characteristic function.) Since decision problems are the by far most common problems studied in computational complexity, we may sometimes wish to stress that a given problem is not necessarily a decision problem. In this case, we speak of function problems.

At this point, a remark regarding the input and output domains $I$ and $O$ is in order. Almost all reasonable types of formal machines that can be used to define precisely what an algorithm is, work on a specific type $T$ of data like strings or numbers. This means that we, strictly speaking, can only use it to solve problems for which $I, O \subseteq T$. In order to circumvent this problem we use codings, normally without explicitly mentioning this fact. In other words, we choose a suitable encoding $\kappa : I \to T$ and use $\kappa(I)$ instead of $I$ as the input (and similarly for the output). We usually do not make $\kappa$ explicit, but have to be careful enough to avoid unreasonable encodings. In particular, the size of $\kappa(I)$ should not differ too much from the size of $I$ (however these two are defined).

Let us now make precise which type of algorithms we want to base our reasonings upon. The standard model used in computational complexity is the multitape Turing machine. In this case, the input and output must be encoded as strings, because Turing machines work on strings. However, in these notes we will use a machine model which is more similar to the modern computer, namely the random access machine. This machine model uses natural numbers as its data type and a program similar to a program in assembly language.

**Definition 1 (RAM)** A (deterministic) random access machine (RAM, for short) consists of program counter $pc$, an unbounded number of registers $R_0, R_1, \ldots$, and a finite sequence of instructions $I_1 \cdots I_m$, the program of the RAM. At any given point in time, each register $R_i$ holds a natural number that is denoted by $R_i$. The value of the program counter is a natural number as well. The available instructions and their effects are given as follows, where $i, j, k \in \mathbb{N}$:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_i \leftarrow j$ or $R_i \leftarrow R_j$</td>
<td>store $j$, $R_j$ or $R_{R_j}$ in $R_i$, increment $pc$ by 1</td>
</tr>
<tr>
<td>$RR_i \leftarrow j$ or $RR_i \leftarrow R_j$ or $RR_i \leftarrow RR_j$</td>
<td>store $j$, $R_j$ or $R_{R_j}$, in $R_{R_i}$, increment $pc$ by 1</td>
</tr>
<tr>
<td>$R_i \leftarrow R_j + R_k$</td>
<td>store $R_j + R_k$ in $R_i$, increment $pc$ by 1</td>
</tr>
<tr>
<td>$R_i \leftarrow R_j - R_k$</td>
<td>store $\max(0, R_j - R_k)$ in $R_i$, increment $pc$ by 1</td>
</tr>
<tr>
<td>if $i$ then goto $j$ ($1 \leq j \leq m + 1$)</td>
<td>set $pc$ to $j$ if $R_i &gt; 0$; else increment $pc$ by 1</td>
</tr>
</tbody>
</table>

We can now define how a RAM computes a function.

**Definition 2 (function computed by a RAM)** For $d \in \mathbb{N}$, let $||d||$ denote the length of the binary representation of $d$. A RAM $M$ with the program $I_1 \cdots I_m$ computes a function $f : \mathbb{N}^* \to \mathbb{N}^*$, as follows:

- Initially, the input $x = a_1 \cdots a_d \in \mathbb{N}^*$ is stored in registers $R_1, \ldots, R_n$, the input registers, while $R_0$ contains $2^d - 1$. All other registers contain the value 0 and the program counter is set to 1.
• As long as \( pc \in \{1, \ldots, m\} \), the corresponding instruction \( I_{pc} \) is executed (with the effect specified in Definition 1). The computation terminates if \( pc = m + 1 \).

• If \( M \) terminates, the output is \( M(x) = R_1 \cdots R_{\|R_0\|} \).

The input size is \( \sum_{i=1}^{d} \|a_i\| \), and the length of a computation (or time consumed) is the total number of steps executed.

The initial value of register \( R_0 \) "informs" the RAM about the number \( d \) of registers that contain input values. It should be noted that, intuitively, this value is stored in unary notation, i.e., if the number of input registers is \( d \), then the binary representation of \( R_0 \) consists of \( d \) ones (unless \( d = 0 \), in which case its value is 0). When the RAM terminates, the number of output registers is specified in a similar way. Although, in the latter case, \( R_0 \) is not explicitly required to be of the form \( 2^d - 1 \), we will in the following assume that this is the case, because it is quite simple to modify a given RAM so that it meets this requirement (by adding a small post-processing phase).

It should be rather obvious that RAMs, despite their deliberately simple definition, can compute all functions that can be computed using modern computers and programming languages. One could, e.g., construct a compiler that compiles simple imperative programs into RAMs. In particular, one could implement constructs like loops and procedure calls that have parameters and return a result.

As mentioned above, we will mainly consider decision problems in these notes. Therefore, the following definition is useful.

**Definition 3** Let \( A \) be a decision problem. A RAM \( M \) decides \( A \) if, for all inputs \( x \),

\[
M(x) = \begin{cases} 
1 & \text{if } x \in A \\
0 & \text{otherwise.} 
\end{cases}
\]

We say that \( M \) accepts \( x \) if \( M(x) = 1 \) and that it rejects \( x \) if \( M(x) = 0 \).

### 4 Efficiently Solvable Problems – the Class P

Let us now discuss polynomial time solvability.

**Definition 4 (polynomial-time decidability)** A RAM \( M \) runs in **polynomial time** if there is some \( k \in \mathbb{N} \) such that the length of computations of \( M \) is \( O(n^k) \), where \( n \) is the input size. The set of all decision problems that can be decided in polynomial time is denoted by \( P \).

Strictly speaking, the preceding definition covers only decision problems in which the instances are sequences of numbers. However, as mentioned above, we can use encodings in order to circumvent this limitation. For example, we can use strings over an alphabet \( \Sigma = \{\sigma_1, \ldots, \sigma_k\} \) as input by applying the encoding given by \( \kappa(\sigma_i) = i \). Thus, a string \( u = \sigma_{i_1} \cdots \sigma_{i_m} \in \Sigma^* \) is turned into the number sequence \( \kappa(u) = i_1 \cdots i_m \). Of course, we can use any other injective mapping of symbols to natural numbers as well. If we use the ASCII or UTF encoding as \( \kappa \), this is even more similar to what is done in a real computer.

The encodings used are usually not explicitly mentioned. Thus, we may for example talk about RAMs that solve problems on graphs, silently assuming that a reasonable encoding is chosen. Fortunately, the class \( P \) is extremely robust against changing the encoding of a problem. In other words, if a problem is in \( P \) under one encoding, it is also in \( P \) under any
other encoding, as long as we consider reasonable encodings. Unreasonable encodings are, e.g., encodings that are extremely difficult to decode and encodings that blow up the size of the input exponentially. If we have to deal with problems involving numbers, there is a potential pitfall that we must carefully avoid: a RAM that computes a function \( f(n) \) in time \( O(n^k) \) for some \( k > 0 \) is not polynomial. The reason is that the input \( n \) has the length \( \log n \), which means that \( O(n^k) \), though polynomial in \( n \), is exponential in the input length. Algorithms of this type are usually said to be pseudopolynomial.

When discussing concrete problems and their complexity, we usually define what the problem instances are (strings over a certain alphabet, graphs, matrices, triples of natural numbers, etc.) and devise algorithms that solve the problem under the assumption that the input is valid. However, what if this is not the case? According to the definitions above, such inputs must be handled within the given time bounds as well. Therefore we should, strictly speaking, always consider this case as well. Fortunately, it turns out that this is not necessary.

**Theorem 5 (disregarding invalid inputs)** Let \( A \) be a decision problem and assume that there is a RAM \( M \) and a \( k \in \mathbb{N} \) such that the following hold for all inputs \( x \):

1. \( M \) accepts \( x \) if and only if \( x \in A \), and
2. if \( x \in A \), then \( M \) runs in time \( O(n^k) \).

Then \( A \) is in \( P \).

**Proof** We have to modify \( M \) in such a way that it rejects \( x \) in polynomial time if \( x \notin A \). Suppose that \( M \) runs in time \( cn^k \) on all \( \text{yes} \)-instances of size \( n \geq n_0 \). Since there are only finitely many \( \text{yes} \)-instances smaller than \( n_0 \), there is a \( c_0 \) such that \( M \) runs in time less than \( cn^k + c_0 \) on all \( \text{yes} \)-instances. (Just choose \( c_0 \) to be the length of the longest accepting computation on inputs of size less than \( n_0 \).)

Now, modify \( M \) as follows. With input \( x \), the modified RAM \( M' \) first computes \( c\|x\|^k + c_0 \) and stores the result in a register \( R_z \). This can easily be done in polynomial time. After this preparatory step, \( M' \) continues precisely like \( M \), but uses \( R_z \) as a yardstick. After each step of \( M \), it decreases \( R_z \) by one. If \( M \) accepts the input before \( R_z \) becomes 0, \( M' \) accepts. Otherwise, \( M' \) rejects the input as soon as \( R_z \) becomes 0.

By construction, \( M' \) runs in polynomial time. To see that it decides \( A \), consider the two cases. If \( x \in A \), then \( M \) accepts \( x \) in time less than \( c\|x\|^k + c_0 \) (by assumptions (1) and (2)). Hence, \( M' \) accepts it before the counter \( R_z \) reaches 0. Conversely, \( M' \) accepts \( x \) only if \( M \) accepts \( x \). By assumption (1), this means that \( x \in A \).

It was mentioned above that \( P \) is a very robust class. This also becomes clear when looking at the composition of polynomial-time RAMs. Composition is a very useful operation since it allows us to combine algorithms to solve complex problems (or break up complex proofs into simpler parts).

**Theorem 6 (composition of polynomial-time RAMs)** If \( M_1 \) and \( M_2 \) are RAMs that run in polynomial time, then \( M_2 \circ M_1 \) can be computed by a RAM in polynomial time.

**Proof** The RAM \( M \) that computes \( M_2 \circ M_1 \) works in the obvious way. If the program of \( M_1 \) consists of \( m \) instructions, the program of \( M \) is obtained by simply concatenating the programs of \( M_1 \) and \( M_2 \), but replacing every instruction if \( i \) then goto \( j \) in \( M_2 \) by if \( i \) then goto \( m+j \). Thus, with an input \( x \), the computation of \( M \) consists of a first phase in which it acts like \( M_1 \) and computes \( M_1(x) \). When \( M_1 \) terminates, the result of its computation is, by definition,
found in the input registers of $M_2$, and the program counter points has the value $m + 1$, meaning that it points to the first instruction of the program of $M_2$. Thus, when $M_2$ terminates, the result will be $M_2(M_1(x))$, as required.

It remains to be shown that $M$ runs in polynomial time. This is, in fact, the slightly more interesting part of the proof. Suppose that $M_1$ and $M_2$ run in time $O(n^k)$ and $O(n^l)$, resp., for some $k, l \geq 1$. Clearly, the running time of $M$ is the sum of the running times of $M_1$ and $M_2$. It is also clear that $M_1$ runs in time $O(n^k)$. However, the running time of $M_2$ in the composition is not necessarily $O(n^l)$, because its input is $M_1(x)$, which may be larger than $x$. How much larger can it be? In each step of the computation of $M_1$, at most one new register can be occupied. Moreover, the maximum number of bits stored in any of the registers can increase by at most 1 in each step (namely by adding the largest number to itself). Thus, after $O(n^k)$ steps, we have $\|R_0\| = O(n^k)$, and the total number of bits stored in the $\|R_0\|$ registers that constitute the output of $M_1$ is at most $\|R_0\| \cdot O(n^k) = O(n^{2k})$. It follows that the second phase of the computation of $M$ runs in time $O((n^{2k})^l) = O(n^{2kl})$. Consequently, the entire computation runs in time $O(n^k + n^{2kl}) = O(n^{2kl})$, which shows that $M$ runs in polynomial time.

\section{Nondeterministic Algorithms and NP}

Nondeterminism is a powerful theoretical concept. Even though it does not seem have a counterpart in the real world, and is thus a purely formal creation, it is very useful in order to understand the phenomenon of problem complexity. In order to discuss nondeterminism, we need a nondeterministic model of computation. We obtain this model by extending the instruction set of random access machines by an instruction that implements a nondeterministic choice.

\begin{definition}[RAM] The \textit{nondeterministic random access machine} (nRAM, for short) is defined in the same way as the RAM (see Definition 1) except for the fact that a program $I_1 \cdots I_m$ may contain nondeterministic instructions of the form $\text{goto } i | j$, where $1 \leq i, j \leq m + 1$. This instruction sets the program counter to either $i$ or $j$.
\end{definition}

Since the nondeterministic instruction has two possible effects, an nRAM with a given input does not only have a single unique computation, but a set of possible computations. Each time a nondeterministic instruction is reached in the execution of the program, there are two ways in which the computation can continue. Thus, for every input the set of valid computations forms a tree in which each deterministic step has exactly one child and each nondeterministic one has two children.

We shall use nRAMs only for solving decision problems. The corresponding definition follows next.

\begin{definition}[deciding problems by an nRAM] An nRAM $N$ \textit{accepts} an input $x$ if at least one of its computations with input $x$ yields the output 1, and it \textit{rejects} $x$ if all computations with input $x$ yield the output 0. $N$ \textit{decides} a decision problem $A$ if it accepts all $x \in A$ and rejects all other inputs.
\end{definition}

Note the asymmetry of the preceding definition. Acceptance requires only one computation to accept the input rather than, e.g., requiring a majority vote.

\begin{definition}[nondeterministic polynomial-time decidability] An nRAM $N$ runs \textit{in time} $f(n)$ for some function $f$ if all computations with inputs of length $n \in \mathbb{N}$ terminate after at most $f(n)$ steps. NP denotes the class of all decision problems that can be decided by an nRAM in polynomial time, i.e., in time $O(n^k)$ for some $k \in \mathbb{N}$.
\end{definition}
The notion of time consumption introduced in the preceding definition is intuitively unrealistic because it seems that one would have to explore the whole computation tree of $N$ (e.g., by depth-first search) in order to find out whether a given input is accepted. This would obviously take time $\Omega(2^{f(n)})$. The whole point in discussing nondeterminism and researching the relation between deterministic and nondeterministic computation is to find out whether this exploration of the entire computation tree can be avoided. Intuitively, the definition nondeterministic acceptance and rejection stipulates that we can somehow “magically” find out whether there is an accepting computation without spending more time than it takes to perform a single computation.

One might ask why such a formalism, that will perhaps never admit a physical realization, is nevertheless considered to be important. Some of the major reasons are:

- Nondeterminism isolates an aspect of computation whose influence on efficiency we do not really understand. If we could understand nondeterminism, this would be a huge step forward in our understanding of computational complexity in general.

- The aspect we do not understand is how to search for a solution in a space of potential solutions. If we can efficiently check the correctness of a solution, but do not know whether there is one, do we really have to explore the whole solution space in a “dumb” way to find out, or can this be avoided?

- Many nondeterministic algorithms are extremely simple. Thus, if we could automatically turn a nondeterministic solution into a deterministic one, this would not only increase efficiency but also save a lot of work.

- In contrast to what many people believe, the interest in NP does not come from the fact that the computational complexity of problems in this class is exceedingly high. The converse is true – NP is the class of problems right beyond the border of what we currently can solve efficiently. Thus, if there is any chance to expand the realm of problems we can solve in an efficient manner, NP is the candidate to bet on.

Indirect addressing causes a slight technical inconvenience in proofs, because a RAM or an nRAM may make use of registers with very large indices even if the total number of registers used is small. The following lemma shows that this can be avoided. Let us say that an nRAM $N$ is memory compact if it has the following property: For every input $a_1 \cdots a_d$ and every computation of $N$, after $j$ computation steps it holds that $R_i = 0$ for all $i > d + j$.

**Lemma 10** For every nRAM $N$, an equivalent memory compact nRAM $N'$ can be constructed. If $N$ runs in polynomial time, $N'$ runs in polynomial time as well.

**Proof** We sketch how $N'$ can be constructed from $N$. Naturally, $N'$ simulates $N$. However, while doing this $N'$ keeps track of the index of the next unused register $R_{\text{next}}$. Whenever a previously unused register $R_i$ is assigned a value by $N$, $N'$ chooses $R_{\text{next}}$ instead, records $\alpha(i) = \text{next}$ in a binary tree, and increments $\text{next}$. The tree representing the mapping $\alpha$ of old to new register addresses is organized in such a way that, if $b_1 \cdots b_l$ is the binary representation of $i$, then $\alpha(i)$ is stored in the node reached by following the path $b_1 \cdots b_l$ from the root of the tree.

Each time a register $R_i$ is accessed by $N$ (for reading or writing), $N'$ uses the binary representation of $i$ to find and access $\alpha(i)$ instead. If $\alpha(i)$ cannot be found in the tree, the register $R_i$ has not been assigned a value before. Thus, if the instruction assigns a value to $R_i$ (i.e., $R_i$}

---

Frank Drewes, Dept. of Computing Science, Umeå University (Sweden)
occurs to the left of $\leftarrow$, a new register is allocated as described above. If the instruction just uses the value $R_i$, $N'$ uses the value 0 instead (since registers that have not been assigned a value before carry the value 0).

As argued in the proof of Theorem 6, a computation of polynomial length cannot produce register contents consisting of more than a polynomial number of bits. Therefore, searching an address $\alpha(i)$ takes only polynomial time, and so does adding a new address. Consequently, $N'$ runs in polynomial time if $N$ does.

Using Lemma 10 we can prove a relatively easy result about the relation between time complexity classes. Here, EXP denotes the set of all problems that can be decided by a RAM in time $2^{O(n^k)}$ for some $k \in \mathbb{N}$.

**Theorem 11** $P \subseteq NP \subseteq EXP$.

**Proof** The first inclusion is correct since, by definition, every RAM is an nRAM. For the second inclusion, consider a memory compact nRAM $N$ that runs in time $n^k + c$. (Memory compactness may without loss of generality be assumed by Lemma 10) A corresponding RAM $M$ may work as follows. Given $d$ input registers and an input of size $n$, $M$ first computes $h = n^k + c + d$, the largest index of registers that $N$ may write to. Now, let us say that a configuration of $N$ (with the given input) is a sequence $pc R_0 \cdots R_h$ of integers, where $pc$ is the current value of the program counter and $R_0, \ldots, R_h$ are the values currently stored in the registers. In every step of the simulation of $N$, the memory contents of $M$ will mainly consist of a stack of configurations of $N$. (In addition, some auxiliary registers are needed in order to implement correct behaviour.) $M$ starts by turning its input into the initial configuration of $N$ by writing $1 R_0 \cdots R_h$ to the stack. Now, $M$ simulates $N$ in a depth-first manner. It always simulates the next step of $N$, applied to the topmost configuration on the stack. There are three cases:

1. If the topmost configuration is terminal (i.e., the program counter is $m + 1$, where $m$ is the length of the program of $N$) then the configuration is either deleted (if the configuration is rejecting) or $M$ accepts its input (if the configuration is accepting).

2. If the program counter of the topmost configuration points to a deterministic instruction of $N$, $M$ simply applies the instruction to the configuration.

3. If the program counter points to an instruction $\text{goto } i \mid j$, $M$ copies the whole configuration and sets the program counters of the copies to $i$ and $j$, respectively.

This is repeated until $M$ accepts (because it has encountered an accepting configuration of $N$) or the stack becomes empty. In the latter case, $M$ rejects its input.

It should be clear that $M$ accepts its input if and only if $N$ does. It remains to be argued that the running time of $M$ is at most exponential. Every simulation step takes $O(n^k)$ computation steps in $M$ (the worst case being a nondeterministic step in which a configuration consisting of $h + 1 = O(n^k)$ registers must be copied). There are $2^{O(n^k)}$ simulation steps to be executed, because this is the size of the computation tree of $N$. Thus, we can conclude that $M$ runs in time $O(n^k) \cdot 2^{O(n^k)} = 2^{O(n^k)}$.

Let us have a look at a few examples of typical problems in NP, starting with the problem Independent Set.

**Independent Set**

**Input:** An undirected graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

**Question:** Does $G$ contain an independent set of size $k$, i.e., a set $V' \subseteq V$ of nodes such that $|V'| = k$ and $(u, v) \notin E$ for all $u, v \in V'$?
An nRAM \( \mathcal{N} \) that decides Independent Set is easily described. Let \((G,k)\) be the input, where \( G = (\{1, \ldots, n\}, E) \). The representation of this input as a sequence of natural numbers may be \( n \), followed by the \( n^2 \) entries of the adjacency matrix of \( G \), followed by \( k \). Now, \( \mathcal{N} \) proceeds as follows to check whether \( G \) has an independent set of size \( k \):

1. The first stage of the computation uses nondeterminism to choose \( k \) nodes. In other words, it generates \( k \) numbers \( v_1, \ldots, v_k \in \{1, \ldots, n\} \) stored in \( k \) registers. This can, for example, be done by first copying \( k \) and then using it as a loop counter. The loop body copies \( n \), using it as a counter for an inner loop. The body of this inner loop initializes a register by 1 and then increments it \( n - 1 \) times. However, it precedes the incrementation instruction in the inner loop by a nondeterministic jump that allows to jump over it. In this way, a number between 1 and \( n \) is generated in each execution of the outer loop.

2. In the second stage, \( \mathcal{N} \) executes another nested loop: For all \( i = 1, \ldots, k \) and \( j = i + 1, \ldots, n \), it checks whether
   - \( v_i = v_j \) or
   - \((v_i, v_j) \in E\).

   If so, it terminates and outputs 0.

3. If the second stage is completed without rejection, \( \mathcal{N} \) accepts the input (i.e., returns 1).

Clearly, this procedure solves Independent Set. If there does not exist any independent set of size \( k \), then all computations will return 0, because every possible choice of nodes in the first stage violates the condition checked in the second stage. Conversely, if there is an independent set, the computation that chooses this set in the first stage will not terminate during the second stage. Hence, this computation accepts the input in the third stage.

Since the first and second step each take time \( \Theta(n^2) \) and the third step takes constant time, this shows that Independent Set belongs to NP.

The next problem to look at is probably the most famous problem in NP, namely Satisfiability (SAT).

**Satisfiability (SAT)**

**Input:** A propositional formula \( \varphi \) in conjunctive normal form (CNF).

**Question:** Is \( \varphi \) satisfiable, i.e., is there an assignment \( \alpha \) of truth values to the boolean variables in \( \varphi \) that makes \( \varphi \) true?

An example is the formula \( \varphi = (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor \neg x_3) \land (x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \), which is satisfiable by exactly two different truth assignments (which ones?). Hence, \( \varphi \in \text{SAT} \).

An nRAM that decides SAT nondeterministically may work as follows: Let the input formula \( \varphi = C_1 \land \cdots \land C_k \) contain \( m \) variables \( x_1, \ldots, x_m \).

1. During the first stage, nondeterminism is used to generate an assignment consisting of \( m \) bits \( b_1, \ldots, b_m \in \{0, 1\} \) stored in \( m \) registers. (This is similar to the nondeterministic generation of nodes above, but simpler because no inner loop is required.) Thus, we interpret \( b_i = 1 \) as \( \alpha(x_i) = \text{true} \).

2. In the second stage, for \( i = 1, \ldots, k \), return 0 if \( \alpha \) makes \( C_i \) false, i.e., if \( C_i \) neither contains
   - a literal \( x_j \) such that \( b_j = 1 \) nor
• a literal \( \neg x_j \) such that \( b_j = 0 \).

3. Finally, accept the input (i.e., return 1).

The correctness argument is similar to the case of Independent Set, i.e., if \( \phi \) is not satisfiable, then every choice made during the first stage will lead to rejection in the second stage. Conversely, if there is a satisfying assignment, choosing the corresponding bits in the first stage makes the nRAM pass the second stage, thus leading to acceptance in the third stage. Clearly, the overall running time is \( O(n) \), which shows that \( \text{SAT} \in \text{NP} \).

The third example is also a very famous one, called Hamiltonian Cycle (HAM)

### Hamiltonian Cycle (HAM)

**Input:** An undirected graph \( G = (V, E) \).

**Question:** Does \( G \) contain a simple cycle\(^4\) of length \(|V|\)?

For instance, we have \( \not\in \text{HAM} \) but \( \in \text{HAM} \).

HAM can be decided nondeterministically in polynomial time as follows, for an input graph \( G = (\{1, \ldots, n\}, E) \).

1. If \( R_i \) is the highest input register, use registers \( R_{i+1}, \ldots, R_{i+n} \) as “markers” that are initially 0 and will be set to 1 when a node is visited.
2. Set \( v = 1 \).
3. Repeat \( n - 1 \) times:
   - If \( v \) is marked, return 0; otherwise, mark \( v \) (by storing 1 in \( R_{i+v} \)).
   - Nondeterministically choose an edge \((v, v') \in E\) and make \( v' \) the new \( v \). (Return 0 if there is no edge that originates from \( v \)).
4. Return 1 if \( v = 1 \), and 0 otherwise.

It should be clear that there is an accepting computation if and only if \( G \in \text{HAM} \). Choosing an edge \((v, v')\) can surely be implemented in time \( O(n^2) \). (Depending on the way in which the input graph is represented, linear time may suffice.) Hence, the overall running time is \( O(n^3) \), which shows that \( \text{HAM} \in \text{NP} \).

6 A Characterization of NP in Terms of Polynomally Bounded Witnesses

We shall now discuss a characterization of NP that describes the class without making use of (explicit) nondeterminism. Recall that a characterization of a concept is a theorem that provides an alternative definition of the concept, i.e., a statement of the form “\( x \) is an \( A \) (the original concept) if and only if it is a \( B \) (the alternative definition)”. In fact, strictly

\(^4\) A simple cycle is a cycle that does not contain a smaller cycle.
A decision problem

Theorem 12 (characterization of NP by witnesses)

We can now state the promised characterization of NP.

and only if there is a polynomially bounded binary relation

\[ R \subseteq \mathbb{N}^* \times \mathbb{N}^* \]

that decides

\[ u \parallel v \]

for a constant \( c \) (due to the finiteness of \( \Sigma \)). Another special case is that of a relation

\[ R \subseteq \Sigma^* \times \{0,1\}^* \]

that turn nondeterministic Turing machines into nRAMs and vice versa in such a way that polynomial time machines are turned into polynomial time machines in both directions.)

The characterization we are aiming at is a formal version of the technique used in the nondeterministic algorithms given above for solving Independent Set and SAT. Both algorithms make use of nondeterminism in a very special way. Their nondeterminism is restricted to an initial phase of the computation in which a sequence of natural numbers is nondeterministically “guessed”. The rest of the computation is entirely deterministic and checks whether the guessed numbers satisfy a certain property closely related to the problem at hand. In the first case, the sequence guessed is interpreted as a set of nodes and it is verified that these nodes form an independent set. In the second case, the sequence is interpreted as an assignment of truth values to the variables of the formula and it is verified that this truth assignment makes the given formula become true. The characterization of NP given below shows that the existence of such special algorithms for problems in NP is not a coincidence.

For the characterization, we need the notion of a polynomially bounded relation. Consider a binary relation \( R \subseteq \mathbb{N}^* \times \mathbb{N}^* \). \( R \) is said to be polynomially bounded if there is a polynomial \( p \) such that \( |v| \leq p(|u|) \) for all \((u, v) \in R\). Note that the relation \( R \) is a set, and as such a decision problem. Given a pair \((u, v) \in \mathbb{N}^* \times \mathbb{N}^*\), the question asked is whether \((u, v) \in R\).

It may be worthwhile to mention the case in which the decision problem is a language over a finite alphabet \( \Sigma \). In this case, it suffices to consider polynomially bounded relations of the form \( R \subseteq \Sigma^* \times \mathbb{N}^* \), where we identify every symbol \( a \in \Sigma \) with an encoding \( \kappa(a) \in \mathbb{N} \). In this case, we may rewrite the boundedness condition as \( |v| \leq p(|u|) \), because \( |\kappa^{-1}(a)| \leq c \cdot |u| \) for a constant \( c \) (due to the finiteness of \( \Sigma \)).

Another special case is that of a relation \( R \subseteq \mathbb{N}^* \times \{0,1\}^* \). In this case, the condition is equivalent to \( |v| \leq p(|u|) \). The two restrictions can then be combined, thus considering relations \( R \subseteq \Sigma^* \times \{0,1\}^* \) that have to satisfy the condition \( |v| \leq p(|u|) \) for some polynomial \( p \).

We can now state the promised characterization of NP.

**Theorem 12 (characterization of NP by witnesses)** A decision problem \( A \) is in NP if and only if there is a polynomially bounded binary relation \( R \) such that

1. \( R \in \mathbb{P} \) and
2. \( A \) is the set of all \( u \in \mathbb{N}^* \) for which there exists a \( v \in \mathbb{N}^* \) (called the witness) such that \((u, v) \in R\).

**Proof** \((\Rightarrow)\) Suppose that \( A \in \mathbb{NP} \). By definition, this means that there is an nRAM \( \mathbb{N} \) that decides \( A \) in time \( p(n) \) for some polynomial \( p \). A computation of \( \mathbb{N} \) is uniquely determined by its input \( u \) and a sequence \( v \in \{0,1\}^* \) of length at most \( p(|u|) \) that encodes the nondeterministic choices made during this computation. Thus the digit 0 denotes that the first branch of a statement \( \text{goto } i \mid j \) is taken (thus continuing with instruction \( i \)) and the digit 1 denotes that the second branch is taken. Let us define \( R \) to be the set of all pairs \((u, v)\) such that \( u \) and \( v \) are as described above and the computation determined by them is accepting. Then \( R \) is polynomially bounded by construction, and it is in \( \mathbb{P} \) because a RAM can decide whether \((u, v) \in R\), as follows:

1. In the first phase it computes \( p(|u|) \) and checks that \( |v| = |u| \). If not, it rejects.

Frank Drewes, Dept. of Computing Science, Umeå University (Sweden)
2. In the second phase it works just like \( N \), but uses the bits in \( u \) to replace the nondeterministic jumps by deterministic ones. 

Since \( p(\|u\|) \) can be computed in polynomial time, this procedure takes polynomial time altogether.

\( \iff \) Suppose that a polynomially bounded binary relation \( R \) with the required properties exists. Let \( p \) be the polynomial such that \( \|v\| \leq p(\|v\|) \) for all \( (u,v) \in R \), and let \( M \) be the RAM that decides \( R \). Then an nRAM that decides \( A \) can work in the following rather obvious manner:

1. The first phase computes \( l = p(\|u\|) \) and generates any sequence \( v \in \mathbb{N}^* \) such that \( \|v\| \leq l \) (by means of nondeterminism).
2. The second phase simply runs \( M \) on the input \( (u, v) \).

Clearly, this procedure decides \( A \), and since it is the composition of two polynomial-time nRAMs, it runs in polynomial time. The latter is true by Theorem 6. (Note that, even though Theorem 6 was stated only for the deterministic case, it is clearly valid for nRAMs as well, because the arguments in its proof do not depend on the assumption that the computation is deterministic.)

It should be noted that the proof of Theorem 12 even shows that the relation \( R \) may be assumed to be a subset of \( \mathbb{N}^* \times \{0,1\}^* \), because the only-if direction was proved by using such a relation.

Theorem 12 shows in a formally precise way what the difference between deterministic and nondeterministic polynomial decidability is. If nondeterminism is available, we only have to be able to check efficiently whether a given sequence \( v \in \mathbb{N}^* \) is indeed a witness for the given input \( u \). For example, in the cases of Independent Set and SAT, we check whether \( v \) is (the encoding of) an independent set or a satisfying assignment, respectively. How \( v \) is obtained does not matter as it “falls from heaven”. In contrast, a deterministic algorithm solving these problems would have to solve this step efficiently as well, which seems to be the complex part of the exercise. Whether the latter is indeed true or just an impression caused by our insufficient insight into the nature of computation is the one million dollar question.

7 Reductions, NP-completeness, and Cook’s Theorem

Intuitively, if \( P \neq \text{NP} \), then there must be problems in \( \text{NP} \) that are significantly harder than each of the problems in \( P \). If we manage to formalize what it means to be significantly harder, we can focus on the hardest problems in \( \text{NP} \), trying to show for one of them that it is not in \( P \). In this way, one could hope to prove the inequality of the two classes. However, how could we define significance or insignificance in a reasonable way in this context?

In view of the motivation above, namely to separate hard problems in \( \text{NP} \) from those in \( P \), the definition should view the difference in time complexity between problems in \( P \) to be insignificant, while the difference between problems in and outside \( P \) should be considered significant. A strong hint is provided by Theorem 6 which states that we can apply any polynomial-time preprocessing to the instances of a problem in \( P \) without leaving \( P \). In other words, if we allow to apply some polynomial-time preprocessing to the instances of a

\[5\] The question whether \( P = \text{NP} \) is one of the six open Millennium Prize Problems for whose solution the Clay Mathematics Institute awards a prize of one million dollars.
A polynomial-time reduction from one problem into another. Such translations are called polynomial-time reductions. The formal definition reads as follows.

**Definition 13 (polynomial-time reduction)** Consider two decision problems \( A \) and \( B \). A polynomial-time reduction from \( A \) to \( B \) is a function \( f \) such that

1. for every input \( x, x \in A \iff f(x) \in B \), and
2. \( f \) can be computed (deterministically!) in polynomial time.

If such a function \( f \) exists, we say that \( A \) is polynomial-time reducible to \( B \), and write \( A \leq_p B \).

Thus, \( f \) is the polynomial-time preprocessing that we apply to the instances of \( A \) in order to turn them into instances of \( B \). The first condition ensures that the translation is correct in the sense that it preserves the answers, i.e., if \( x \) is a yes-instance of \( A \) then \( f(x) \) is a yes-instance of \( B \), and vice versa.

The relation \( \leq_p \) is a so-called preorder (or quasi-order), that is, it is transitive and reflexive. To see that it is transitive, suppose that \( A \leq_p B \leq_p C \). We have to show that \( A \leq_p C \). By assumption, there are polynomial-time reductions \( f_1 \) and \( f_2 \) from \( A \) to \( B \) and from \( B \) to \( C \), respectively. Then the composition \( f = f_2 \circ f_1 \) is computable in polynomial time (by Theorem 6), and we have \( x \in A \iff f_1(x) \in B \iff f_2(f_1(x)) \in C \), which shows that \( x \in A \iff f(x) \in C \). Hence, \( f \) is a polynomial-time reduction from \( A \) to \( C \). Thus, \( \leq_p \) is transitive.

The statement that the difference between problems in \( P \) is insignificant under polynomial-time reductions is confirmed by the following theorem. For this theorem, let us call a problem trivial if it has only positive or only negative instances (i.e., the answer is always yes or no).

**Theorem 14** For all problems \( A, B \in P \), if \( B \) is not trivial then \( A \equiv_p B \).

**Proof** We have to find a polynomial-time reduction \( f \) from \( A \) to \( B \). For this, choose any \( y^+ \in B \) and \( y^- \notin B \), and define

\[
f(x) = \begin{cases} 
y^+ & \text{if } x \in A 
y^- & \text{otherwise.}
\end{cases}
\]
By the definition of \( f \), we have \( x \in A \Rightarrow f(x) = y^+ \Rightarrow f(x) \in B \) and, conversely, \( x \notin A \Rightarrow f(x) = y^- \Rightarrow f(x) \notin B \). Hence the first condition is satisfied. Concerning the second condition, we can compute \( f \) by checking whether \( x \in A \) (which can be done in polynomial time since \( A \in \text{P} \)) and then, depending on the result, output either \( y^+ \) or \( y^- \) (which takes constant time). Thus, we have shown that \( f \) is a polynomial-time reduction from \( A \) to \( B \).

It should be noted that, intuitively, the reduction in the preceding proof does all the work itself, by first deciding whether \( x \in A \). It can do so, because \( A \in \text{P} \), and so the reduction is allowed to use all the resources required to solve \( A \). If \( A \notin \text{P} \) (or we do not know whether it is), we cannot argue in this way, because then the reduction would not be polynomial any more. Thus, those cases are the interesting ones, because we have to find a more intelligent way to translate one problem into another one.

Let us now consider the statement that the difference between problems in and outside \( \text{P} \) under polynomial-time reductions is significant. The corresponding theorem follows.

**Theorem 15 (P is (backwards) closed under reductions)** If \( A \leq_p B \) and \( B \in \text{P} \) then \( A \in \text{P} \). (In other words, if \( A \notin \text{P} \) then \( A \nleq_p B \) for all \( B \in \text{P} \).)

**Proof** Let \( \mathcal{M} \) be a RAM that decides \( B \), and let \( \mathcal{M}_0 \) be another one, that computes a reduction \( f \) from \( A \) to \( B \), both in polynomial time. Then the RAM \( \mathcal{M}' \) that computes the composition \( M \circ M_0 \) runs in polynomial time by Theorem 6, and \( M'(x) = M(M_0(x)) = M(f(x)) \). Therefore,

\[
x \in A \iff f(x) \in B \iff M(f(x)) = 1 \iff M'(x) = 1.
\]

Thus, \( \mathcal{M}' \) decides \( A \) in polynomial time.

Together, the preceding theorems tell us that all problems in \( \text{P} \) (except the two trivial ones) are equally hard with respect to polynomial-time reducibility, whereas problems outside \( \text{P} \) are strictly harder (in the sense that there is no polynomial-time reduction from any of those to a problem in \( \text{P} \)). This is precisely what is needed in order to investigate the suspected difference between \( \text{P} \) and \( \text{NP} \). In particular, it makes sense to look at the hardest problems in \( \text{NP} \). These are the \( \text{NP} \)-complete problems, defined below.

**Definition 16 (NP-complete)** A decision problem \( B \) is \( \text{NP-hard} \) if \( A \leq_p B \) for every problem \( A \in \text{NP} \). An \( \text{NP-hard} \) problem which is in \( \text{NP} \) is \( \text{NP-complete} \).

We note here that the definition of hardness and completeness works for other complexity classes, such as \( \text{EXP} \), as well. However, in this form, it only makes sense for classes that are (suspected to be) strict supersets of \( \text{P} \). For \( \text{P} \) and classes even smaller than \( \text{P} \), we would need another type of reduction (because of Theorem 14).

\( \text{NP-complete} \)ness could be used to prove \( \text{P} = \text{NP} \) (provided that the equality holds). It suffices to solve one \( \text{NP-complete} \) problem efficiently.

**Theorem 17** If there is an \( \text{NP-complete} \) problem \( A \) such that \( A \in \text{P} \) then \( \text{P} = \text{NP} \).

**Proof** Since \( A \) is \( \text{NP-complete} \) by assumption, we have \( B \leq_p A \) for all \( B \in \text{NP} \). By the closedness of \( \text{P} \) under reductions (Theorem 15), \( A \in \text{P} \) thus implies that \( B \in \text{P} \), which shows that \( \text{NP} \subseteq \text{P} \).

Another easy result is a more or less direct consequence of the transitivity of \( \leq_p \), which provides us with a convenienta way to prove that a given problem is \( \text{NP-complete} \). For this, we just have to show that another \( \text{NP-complete} \) problem can be reduced to it in polynomial time. This is what the next lemma states.

**Lemma 18** If \( A \in \text{NP} \) and \( B \leq_p A \) for an \( \text{NP-complete} \) problem \( B \), then \( A \) is \( \text{NP-complete} \).
Proof Assume that $A \in \text{NP}$ and $B \leq_p A$, where $B$ is NP-complete. Then, for every problem $C \in \text{NP}$, we have $C \leq_p A$ since $C \leq_p B \leq_p A$.

Hence, to prove that $A$ is NP-complete, the following proof pattern can be applied:

1. show that $A \in \text{NP}$,
2. choose a suitable NP-complete problem $B$, so it is possible to find a reduction $f$ from $B$ to $A$,
3. argue that $f$ is polynomial-time computable,
4. show that $f(x) \in A$ if and only if $x \in B$.

References

