2 Brief Introduction to Computational Complexity

Typically, there are many different ways to tackle a computational problem. Different approaches lead to different algorithms that solve the same problem. Consequently, one faces the task of comparing these algorithms with each other. What should be the criteria under which we evaluate an algorithm? Depending on the application, different criteria may be important: Sometimes *simplicity* is crucial to enable a quick and error-free implementation. For complex problems, a clean and easy-to-understand structure is often in the foreground: A mathematical language, mathematical tools, and a mathematical formalism should be used in such a way that complicated aspects are cleanly expressed in a comprehensible way. This avoids errors in implementation and interpretation. When the focus is on dealing with large problem instances, however, the *efficiency* of the algorithm is often decisive: we want to solve the problem with the least amount of computation. This leads us to the runtime analysis of algorithms. When we examine the running time of an algorithm, we want to know how much time it needs to solve a particular problem instance. We are interested in the dependence of this duration on the size of the problem instance. From a formal point of view, we have to be clear on what we mean by the "size" of a problem instance and how we measure the "running time" (or "runtime") of an algorithm. When we talk about a problem class in the context of Complexity Theory, we limit ourselves to instances of a problem for which the algorithm we are examining was designed. For example, for the problem of finding the largest among n numbers, a list of n numbers comprises a problem instance.

The size of a problem is the number of bits needed to store it. For example, an integer $a \in \mathbb{Z}_{\geq 0}$ can be saved in a standard representation with $\lceil \log_2(a+1) \rceil$ bits, where $\lceil q \rceil$ for $q \in \mathbb{R}$ represents the smallest integer $r \in \mathbb{Z}$ that satisfies $r \geq q$. The number 9 can therefore be represented as a binary code in the form `1001`. Because we often want to encode negative numbers, we can use an extra bit to encode the sign. More generally, a (possibly negative) integer $a \in \mathbb{Z}$ can be represented in $\lceil \log_2(|a| + 1) \rceil + 1$ bits.

Unfortunately, it is difficult to give a good definition of the "runtime" of an algorithm. A first approach could be to implement the algorithm in a common programming language, run it on different sized problem instances and analyze the measured running times. However, such measurements are highly dependent on the underlying hardware, the programming language used, and many details linked to the implementation. But all these aspects have no relation to the efficiency of the algorithm that we actually want to measure. For the running time of an algorithm, a much less fine-grained measure is employed in Complexity Theory, with the goal of achieving independence of the above-described factors: The running time of an algorithm is measured by counting *elementary operations*. Elementary operations include additions, subtractions, multiplications, divisions, and comparisons of two numbers. In particular, we want to

measure the number of elementary operations performed depending on the size of the problem instance, which we refer to as $\langle \text{input} \rangle$ and which represents the number of bits used to encode input.

Example 2.1: Maximum element

Consider the problem of finding the largest number in a given list $a_1, \dots, a_n \in \mathbb{Z}_{\geq 0}$ of numbers. One possible algorithm that solves this problem is as follows: Start by comparing a_1 and a_2 and store the larger of the two numbers as x, i.e., $x = \max\{a_1, a_2\}$. Compare x with a_3. If a_3 is the larger of the two numbers, override x with a_3, and so on. This algorithm performs $n-1$ comparisons, so its running time is $n-1$. Because we need at least n bits to store the numbers $a_1, \dots, a_n$, the running time of this algorithm is linearly (upper) bounded by the size of the problem instance.
This example shows a situation typical of running time analyses: Often, one can describe the running time as a function of the number of given elements (in this case n), without having to look at the exact size of the input instance in binary encoding. If a polynomial bound of this type is possible on the number of elementary operations, we talk about a *strongly polynomial algorithm*.

In the example above, it was easy to specify an upper bound on the runtime of the algorithm. Note that the actual running time of an algorithm, i.e., the number of elementary operations, depends not only on the size of the concrete problem instance but also on the problem instance itself. For example, consider an algorithm that calculates the largest common divisor of two numbers $a_1, a_2 \in \mathbb{Z}_{>0}$. Such an algorithm can terminate much faster on some problem instances (e.g., if $a_1 = a_2$) than on others. Classical running time analysis in Complexity Theory focusses on so-called *worst case analysis*. That is, for each possible input size $s = \langle \text{input} \rangle$, we want to know an upper bound on the running time of the algorithm that holds for *any* instance of input size at most s. The runtime of an algorithm is thus captured by a function $f\colon \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$, such that every problem instance of input size at most s is solved in at most $f(s)$ elementary operations.

To further simplify the analysis and avoid dependencies on unimportant details, we are not interested in the exact number of elementary operations, but only in their *order*. More specifically, we want to determine the runtime up to a constant factor. For example, we call an algorithm *linear* if there is a constant $c > 0$ such that, for each problem instance, the algorithm needs at most $c \cdot \langle \text{input} \rangle$ elementary operations, where $\langle \text{input} \rangle$ is the size of the problem instance. Thus, for an algorithm to have linear running time, it does not matter how large the constant c is, as long as its runtime is bounded by $c \cdot \langle \text{input} \rangle$. To succinctly describe the order of a term without including constant factors, we use the Landau O notation. In this notation, the runtime $f(\langle \text{input} \rangle) \leq c \cdot \langle \text{input} \rangle$ of a linear algorithm is written as $f(\langle \text{input} \rangle) = O(\langle \text{input} \rangle)$, that of a quadratic algorithm as $f(\langle \text{input} \rangle) = O(\langle \text{input} \rangle^2)$. Analogously to the Landau O notation, which describes upper bounds up to constant factors, the notations Ω and Θ are commonly used for lower bounding expressions and relating expressions of the same order (up to constant factors), respectively. They are defined as follows.

Definition 2.2: Landau notation

Let f and g be two functions.

(i) We write $f = O(g)$ if and only if

$$\exists M > 0, c > 0 \quad \text{such that} \quad |f(s)| \leq c \cdot |g(s)| \quad \forall s \geq M \ .$$

(ii) We write $f = \Omega(g)$ if and only if $g = O(f)$.

(iii) We write $f = \Theta(g)$ if and only if $f = O(g)$ and $f = \Omega(g)$.

In words, $f = O(g)$ means that, for values of s that are large enough, $f(s)$ does not grow faster than $g(s)$, up to a constant factor. Similarly, $f = \Omega(g)$ means that, for s being large enough, $f(s)$ does not grow more slowly than $g(s)$, up to a constant factor. The combination of both properties, that is $f = \Theta(g)$, means that f and g grow equally fast, up to constant factors, for large values.

It turns out that subdividing algorithms into fast and slow algorithms, according to the criterion of whether their runtime can be bounded by a polynomial in the size of the input instance, is a natural way that proved to be highly influential both in theory and practice. As highlighted below, algorithms that run in polynomial time are called *efficient* algorithms and, equally importantly, this notion can be used to classify problems.

Definition 2.3: Polynomial algorithms and problems

An algorithm is *polynomial* or *efficient* if its running time $f(\langle \text{input} \rangle)$ is bounded by a polynomial in the size of the input, i.e., there is a polynomial g such that

$$f = O(g) \ .$$

A problem is *solvable in polynomial time* if it can be solved by a polynomial algorithm.

Example 2.4: Systems of linear equations

Consider a linear equation system of the form $Ax = b$ with $A \in \mathbb{Z}^{m \times n}$ and $b \in \mathbb{Z}^m$. Note that the special case of integer systems that we consider also covers the more general case of rational systems of equations, since each rational system of equations can be converted into an equivalent integer equation system by multiplying it by a common multiple of all occurring denominators. The size of the problem instance is approximately given by

$$\langle \text{input} \rangle \approx \sum_{i=1}^{m} \sum_{j=1}^{n} \left(\lceil \log_2(|A_{ij}| + 1) \rceil + 1 \right) + \sum_{i=1}^{m} \left(\lceil \log_2(|b_i| + 1) \rceil + 1 \right) \ . \tag{2.1}$$

We do not write an actual equality in the above statement, because we normally cannot just write all encodings consecutively without additional overhead. Indeed, we have to make sure that we are able to separate the encodings of the involved entries. From a technical point of view, this is not hard to achieve, for example when using additional syntax for delimiters to

mark the beginning and end of each encoded entry. However, the number of these extra bits is typically small compared to the expression in (2.1). Because, as explained above, we are only interested in the order of magnitude of the running time as a function of the input size, constant factors do not play a role in the analysis. It is therefore sufficient to estimate the size of the problem instance up to a constant factor. In this case, we see that (2.1) satisfies this requirement. In addition, we emphasize that due to the double sum in (2.1) we have $\langle \text{input} \rangle \geq m \cdot n$.
The system $Ax = b$ can be solved through Gaussian elimination in $O(mn^2)$ elementary operations. In particular, this means that the number of elementary operations is restricted by a polynomial in $\langle \text{input} \rangle$, because $\langle \text{input} \rangle \geq m \cdot n$. It can also be shown that the encoding length of any matrix entry encountered during the calculations is upper bounded by a polynomial in $\langle \text{input} \rangle$. Hence, Gaussian elimination is an efficient algorithm for the solution of linear systems of equations.

We are particularly interested in the complexity of finding optimal solutions to mathematical optimization problems. When talking about the complexity of problems, one typically considers so-called *decision problems*. A decision problem is a problem that asks a yes-or-no question. For example, one could consider a linear program and ask whether its optimal value is at least some number q. This way, optimization problems can naturally be transformed into decision problems.

Complexity Theory divides decision problems into different classes. The following are the arguably two best known complexity classes:

- The class $\mathcal{P}$, which contains all polynomially-solvable problems (or the corresponding decision problems). We have seen that both finding the maximum in a list of given numbers (Example 2.1) and solving linear systems of equations (Example 2.4) are included in $\mathcal{P}$. For problems in $\mathcal{P}$, also large problem instances can very often be solved in a relatively short time.
- The class $\mathcal{NP}$, which contains all the decision problems for which the answer "yes" can be *verified* by a cleverly chosen polynomial-size *certificate* in polynomial time. The *certificate* is often a solution to the problem itself, but may also contain other (polynomially bounded) information. The idea of a certificate is that you can quickly convince yourself that a "yes" instance really is a "yes" instance. For example, consider the decision problem whether a number $a \in \mathbb{Z}_{>0}$ is *not* a prime. If a is not a prime, you could choose as a certificate a divisor $t \in \{2, \dots, a-1\}$ of a. In fact, given such a divisor t of a, we can convince ourselves very quickly (in polynomial time) that a is a "yes" instance, i.e., it is not a prime: it is enough to check that a is actually divisible by t. This algorithmic process of convincing yourself with a certificate is called *verification*. Hence, problems in $\mathcal{NP}$ allow for fast verification of "yes" instances, but that does not imply that a "no" instance can be quickly verified, nor does it imply that a problem class can be solved efficiently, i.e., that it lies in $\mathcal{P}$.
  Another example of a problem in $\mathcal{NP}$ is the so-called *knapsack problem*: Given is a set $[n]$ of n objects, where each object $i \in [n]$ has a weight $w_i \in \mathbb{Z}_{\geq 0}$ and a value $p_i \in \mathbb{Z}_{\geq 0}$, and a global weight bound $W \in \mathbb{Z}_{\geq 0}$. The task is to select a subset $I \subseteq [n]$ of the objects with total weight not exceeding W, i.e., $\sum_{i \in I} w_i \leq W$ and total value as large as possible,

i.e., it should maximize $\sum_{i \in I} p_i$. The natural decision version of this problem adds to the input of the problem a value $P \in \mathbb{Z}_{\geq 0}$ and asks whether there is a knapsack solution of value at least P. A knapsack solution of value at least P clearly is a certificate that the answer is "yes". However, it is not clear how, in case of a "no"-instance, one could provide a quickly checkable proof showing that no knapsack solution with value at least P exists.

One may wonder, whether by moving from an optimization problem to its natural decision version, like in the knapsack example above, one may end up with a significantly easier problem. This is not the case. For example, assume we had an efficient algorithm for the decision version of the knapsack problem. Then we could apply binary search with respect to P to find a knapsack solution of largest possible value, thus solving the optimization problem. The range over which we have to apply binary search can be bounded by $[0, \sum_{i=1}^{n} p_i]$. Binary search needs a number of steps that is logarithmic in the width of this search interval, which leads to a number of steps that is linear in the input size. Hence, in particular, an efficient algorithm to solve the decision version of the knapsack problem can easily be transformed to efficiently solve the optimization version.

Intriguingly, it remains unknown whether the two complexity classes $\mathcal{P}$ and $\mathcal{NP}$ are different, i.e., whether $\mathcal{P} \neq \mathcal{NP}$. The question of whether $\mathcal{P} \neq \mathcal{NP}$ is one of the most famous open problems in Mathematics and Computer Science. It is one of the 7 Millennium Prize Problems, stated by the Clay Mathematics Institute, who awards a prize of 1 000 000 USD for its resolution.[1] The prevailing opinion of experts in the field is that $\mathcal{P} \neq \mathcal{NP}$.

Notice that we have $\mathcal{P} \subseteq \mathcal{NP}$, because a provably correct algorithm for a problem in $\mathcal{P}$ can be used to certify in polynomial time the correctness of a yes answer. The problems in $\mathcal{NP}$ are a very big problem class with simple and extremely complex problems. To better understand how difficult problems can be in $\mathcal{NP}$, a subclass of $\mathcal{NP}$ has been defined, the so-called $\mathcal{NP}$-*complete problems*. $\mathcal{NP}$-complete problems can be interpreted as the most difficult problems in $\mathcal{NP}$. In particular, it can be shown that if you can efficiently solve any single $\mathcal{NP}$-complete problem, you can efficiently solve all the problems in $\mathcal{NP}$, which would imply $\mathcal{P} = \mathcal{NP}$. The existence of $\mathcal{NP}$-complete problems is a very central and profound result in Complexity Theory that led to a significantly better understanding of the problems that the class $\mathcal{NP}$ captures. If $\mathcal{P} \neq \mathcal{NP}$ is true, then no $\mathcal{NP}$-complete problem can be solved efficiently. An obvious proof plan for $\mathcal{P} \neq \mathcal{NP}$ would thus start with an $\mathcal{NP}$-complete problem and show that it is not efficiently solvable.

It is crucial to understand that the number of solutions typically does not reflect the complexity of a problem. Many combinatorial problems have solution spaces that are exponentially large in the size of the input but can nevertheless be solved efficiently. One such example is sorting n different integers in ascending order. To algorithmically deal with a problem, it is crucial to know which complexity class the problem belongs to. In some cases, this is anything but simple—and there remain many interesting problem classes for which this question remains open.

Complexity Theory has strongly influenced research on algorithms. Especially for problems that are known to belong to difficult complexity classes (for example, $\mathcal{NP}$-complete problems),

[1] See https://www.claymath.org/millennium-problems for more information on the Millennium Prize Problems.

Complexity Theory gives clear indications that there are good chances that no polynomial algorithms exist to solve these problems. Hence, different approaches need to be explored. This is a central motivation of so-called *approximation algorithms*. These are efficient algorithms that return solutions of a guaranteed quality (in a well-defined sense) with respect to an optimal solution. Also, when we know from a complexity-theoretic point of view that a problem is very difficult to solve, this also justifies the use of *heuristic* algorithms, which typically do not come with any solution guarantee but are often very fast.