# Chapter 1

# Introduction to Discrete Optimization

Roughly speaking, **discrete optimization** deals with finding the best solution out of finite number of possibilities in a computationally efficient way. Typically the number of possible solutions is larger than the number of atoms in the universe, hence instead of mindlessly trying out all of them, we have to come up with insights into the **problem structure** in order to succeed. In this class, we plan to study the following classical and basic problems:

- Minimum spanning trees
- The Shortest Path problem
- Maximum flows
- Matchings
- The Knapsack problem
- Min Cost flows
- Integer Programming

The purpose of this class is to give a proof-based, formal introduction into the theory of discrete optimization.

## 1.1 Algorithms and Complexity

In this section, we want to discuss, what we formally mean with **problems**, **algorithms** and **running time**. This is made best with a simple example. Consider the following problem:

| FIND DUPLICATE |
| --- |
| **Input:** A list of numbers $a_1, \ldots, a_n \in \mathbb{Z}$ |
| **Goal:** Decide whether some number appears at least twice in the list. |

Obviously this is not a very interesting problem, but it will serve us well as introductory example to bring us all one the same page. A straightforward algorithm to solve the problem is as follows:

(1) FOR $i = 1$ TO $n$ DO
    (2) FOR $j = i + 1$ TO $n$ DO
        (3) If $a_i = a_j$ then return "yes"
(4) Return "no"

The algorithm is stated in what is called **pseudo code**, that means it is not actually in one of the common programming languages like Java, C, C++, Pascal or BASIC. On the other hand, it takes only small modifications to translate the algorithm in one of those languages. There are no consistent rules what is allowed in pseudo code and what not; the point of pseudo code is that it does not need to be machine-readable, but it should be human-readable. A good rule of thumb is that everything is allowed that also one of the mentioned programming languages can do.

In particular, we allow any of the following operations: addition, subtraction, multiplication, division, comparisons, etc. Moreover, we allow the algorithm an infinite amount of memory (though our little algorithm above only needed the two variables $i$ and $j$).

The next question that we should discuss in the analysis of the algorithm is its **running time**, which we define as the **number of elementary operations** (such as adding, subtracting, comparing, etc) that the algorithm makes. Since the variable $i$ runs from 1 to $n$ and $j$ runs from $j = i + 1$ to $n$, step (3) is executed $\binom{n}{2} = \frac{n(n-1)}{2}$ many times. On the other hand, should we count only step (3) or shall we also count the FOR loops? And in the 2nd FOR loop, shall we only count one operation for the comparison or shall we also count the addition in $i + 1$? We see that it might be very tedious to determine the exact number of operations. On the other hand we probably agree that the running time is of the form $Cn^2$ where $C$ is some constant that might be, say 3 or 4 or 8 depending on what exactly we count as an elementary operation. Let us agree from now on, that we only want to determine the running time up to constant factors.

As a side remark, there is a precisely defined formal computational model, which is called a **Turing machine** (interestingly, it was defined by Alan M. Turing in 1936 before the first computer was actually build). In particular, for any algorithm in the Turing machine model one can reduce the running time by a constant factor while increasing the state space. An implication of this fact is that running times in the Turing machine model are actually only well defined up to constant factors. We take this as one more reason to be content with our decision of only determining running times up to constant factors.

So, the outcome of our runtime analysis for the FIND DUPLICATE algorithm is the following:

> There is some constant $C > 0$ so that the FIND DUPLICATE algorithm finishes after at most $Cn^2$ many operations. (1.1)

Observe that it was actually possible that the algorithm finishes much faster, namely if it finds a match in step (3), so we are only interested in an upper bound. Note that the simple algorithm that we found is not the most efficient one for deciding whether $n$ numbers contain a duplicate. It is actually possible to answer that question in time $C'n\log(n)$ using a sorting algorithm. If we want to compare the running times $Cn^2$ and $C'n\log(n)$, then we do not know which of the constants $C$ and $C'$ is larger. So for small values of $n$, we don't know which algorithm would be faster. But $\lim_{n\to\infty} \frac{Cn^2}{C'n\log(n)} = \infty$, hence if $n$ is large enough the $C'n\log(n)$ algorithm would outperform the $Cn^2$ algorithm. Thus, we do consider the $C'n\log(n)$ algorithm as the more efficient one[1].

It is standard in computer science and operations research to abbreviate the claim from (1.1) using the ***O*-notation** and replace it by the equivalent statement:

> The FIND DUPLICATE algorithm takes time $O(n^2)$. (1.2)

[1]Most constants that appear in algorithms are reasonable small anyway. However, there are fairly complicated algorithm for example for matrix multiplication which theoretically are faster than the naive algorithms, but only if $n$ exceeds the number of atoms in the universe.

The formal definition of the $O$-notation is a little bit technical:

**Definition 1.** If $f(s)$ and $g(s)$ are two positive real valued functions on $\mathbb{N}$, the set of non-negative integers, we say that $f(n) = O(g(n))$ if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n$ greater than some finite $n_0$.

It might suffice to just note that the statements (1.1) and (1.2) are equivalent and we would use the latter for the sake of convenience.

Going once more back to the FIND DUPLICATE algorithm, recall that the **input** are the numbers $a_1, \ldots, a_n$. We say that the **input length** is $n$, which is the number of numbers in the input. For the performance of an algorithm, we always compare the running time with respect to the input length. In particular, the running time of $O(n^2)$ is bounded by a polynomial in the input length $n$, so we say that our algorithm has **polynomial running time**. Such polynomial time algorithms are considered **efficient** from the theoretical perspective[2]. Formally, we say that any running time of the form $n^C$ is polynomial, where $C > 0$ is a constant and $n$ is the length of the input. For example, below we list a couple of possible running times and sort them according to their asymptotic behavior:

$$\underbrace{100n \ll n\ln(n) \ll n^2 \ll n^{10}}_{\text{efficient}} \ll \underbrace{2^{\sqrt{n}} \ll 2^n \ll n!}_{\text{inefficient}}$$

### 1.1.1 Complexity theory and NP-hardness

We want to conclude with a brief and informal discussion on **complexity theory**. The **complexity class P** is the class of problems that admit a **p**olynomial time algorithm. For example, our problem of deciding whether a list of numbers has duplicates is in **P**. However, not all problems seem to admit polynomial time algorithms. For example, there is no polynomial time algorithm known for the following classical problem:

PARTITION
**Input:** A list of numbers $a_1, \ldots, a_n \in \mathbb{N}$
**Goal:** Decide whether one can partition $\{1, \ldots, n\}$ into $I_1 \dot\cup I_2 = \{1, \ldots, n\}$ so that

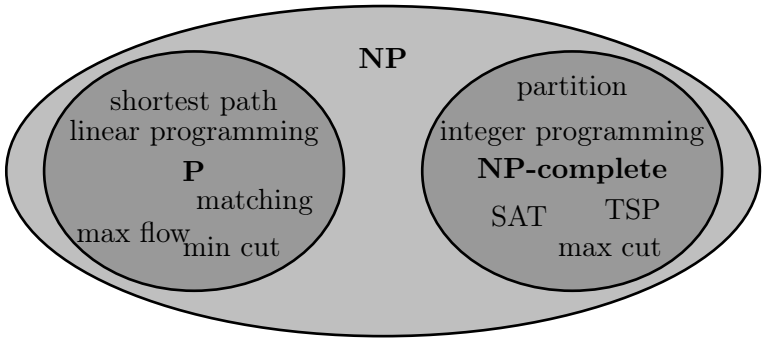
$$\sum_{i \in I_1} a_i = \sum_{i \in I_2} a_i.$$

To capture problems of this type, one defines a more general class: **NP** is the class of problems that admit a **n**on-deterministic **p**olynomial time algorithm. Intuitively, it means that a problem lies in **NP** if given a solution one is able to verify in polynomial time that this is indeed a solution. For example for PARTITION, if somebody claims to us that for a given input $a_1, \ldots, a_n$ the answer is "yes", then (s)he could simply give us the sets $I_1, I_2$. We could then check that they are indeed a partition of $\{1, \ldots, n\}$ and compute the sums $\sum_{i \in I_1} a_i$ and $\sum_{i \in I_2} a_i$ and compare them. In other words, the partition $I_1, I_2$ is a **computational proof** that the answer is "yes" and the proof can be verified in polynomial time. That is exactly what problems in **NP** defines. Note that trivially, $\mathbf{P} \subseteq \mathbf{NP}$.

We say that a problem $P \in \mathbf{NP}$ is **NP-complete** if with a polynomial time algorithm for $P$, one could solve any other problem in **NP** in polynomial time. Intuitively, the **NP**-complete problems

[2]This is true for theoretical considerations. For many practical applications, researchers actually try to come up with near-linear time algorithms and consider anything of the order $n^2$ as highly impractical.

are the hardest problems in **NP**. One of the 7 Millenium problems (with a $1,000,000 award) is to prove the conjecture that **NP**-complete problems do not have polynomial time algorithms (i.e. $\mathbf{NP} \neq \mathbf{P}$). An incomplete overview over the complexity landscape (assuming that indeed $\mathbf{NP} \neq \mathbf{P}$) is as follows:




From time to time we want to make some advanced remarks that actually exceed the scope of this lecture. Those kind of remarks will be in gray box labeled **advanced remark**. Those comments are not relevant for the exam, but they give some background information for the interested student.

**Advanced remark:**

Now with the notation of **P** and **NP**, we want to go back to how we determine the running time. We used what is called the **arithmetic model / RAM model** where any arithmetic operation like addition, multiplication etc, counts only one unit. On the other hand, if we implement an algorithm using a Turing machine, then we need to encode all numbers using bits (or with a constant number of symbols, which has the same effect up to constant factors). If the numbers are large, we might need a lot of bits per number and we might dramatically underestimate the running time on a Turing machine if we only count the number of arithmetic operations. To be more concrete, consider the following (useless) algorithm

(1) Set $a := 2$
(2) FOR $i = 1$ TO $n$ DO
    (3) Update $a := a^2$.

The algorithm only performs $O(n)$ arithmetic operations. On the other hand, the variable at the end is $a = 2^{2^n}$. In other words, we need $2^n$ bits to represent the result, which leaves an exponential gap between the number of operations in the arithmetic model and the **bit model** where we count each bit operation. It is even worse: One can solve **NP**-complete problems using a polynomial number of arithmetic operations by creating numbers with exponentially many bits and using them to do exponential work.
For a more formal accounting of running time, it is hence necessary to make sure that the gap between the arithmetic model and the bit model is bounded by a polynomial in the input length. For example it suffices to argue that all numbers are not more than single-exponentially large in the the input. All algorithms that we consider in this lecture notes will have that property, so we will not insist on doing that (sometimes tedious) part of the analysis.