Algorithms. Lecture Notes 1

- These notes are mainly based on: Kleinberg, Tardos, *Algorithm Design*, and they are also influenced by other books and materials.
- The notes are an additional service. They should be considered only as concise summaries of the lectures and a mnemonic aid. It was not the intention to write another textbook.
- Many details, as well as diagrams and calculation examples, are therefore omitted.
- The contents follow roughly the lectures, but they may differ from what was exactly said in class.

About Algorithms in General

Until recently the word "algorithms" was only known to specialists, but nowadays algorithms are even discussed in the media, in connection with big data analytics and automated decision making, and often without a clear idea what the notion of "algorithm" actually means. Let us clarify the notion first:

An **algorithm** is nothing but an instruction for doing some calculations.

These calculations should "make sense", more precisely, we intend to use algorithms to solve problems. Here, a **problem** is specified by a set of **instances** (inputs) and a desired result (solution, output) for every instance. An algorithm must return a correct solution *for every possible instance.*

We will always use the terms "problem" and "instance" in this sense, so distinguish them carefully. A simple example of a problem is the addition of two integers: Given a pair (x, y) of integers, we wish to compute their sum $x + y$. Here, every specific pair (x, y) is an instance, whereas the *problem* is to compute $x + y$ for *every possible* instance (x, y). Note that, in general, a problem has infinitely many possible instances (at least in principle, ignoring physical limitations). Now we can state in more detail:

An algorithm is a precise and unambiguous description of the calculations to be done in order to solve a given problem for every possible instance.

One may object that this is not really a definition, just an intuitive and informal circumsciption. Actually there exist also formal definitions of the concept of algorithms, such as Turing machines and recursive functions. But they are a bit technical, moreover, one cannot prove that they exactly capture what we would like to call algorithms, because for a proof we would have to formalize first what algorithms are, and just this formalization was the job of the definition, so we have a hen-and-egg problem. Independently of this issue, the intuitive understanding of the concept is sufficient, as long as we are concerned with algorithms for specific problens. For any specific calculation instruction it is easy to confirm that it is indeed an algorithm in the intuitive sense. A formal notion of "all possible algorithms" is needed only if we want to explore the limits of computation and prove that some problem cannot be solved by any algorithm.

One of the formal definitions of algorithms is the Turing machine. We do not fully describe it here, but one can say that it formalizes the idea that *the work of any algorithm can be divided into extremely simple steps*: navigate in a discrete space (e.g., simply a tape) where symbols are written, read a symbol, write a symbol; where the next action can be conditional on what

symbol has been read. The Church-Turing thesis holds that all algorithms can be formalized in this way, that is, other kinds of steps are never needed. A deeper discussion of this thesis would touch on philosophical questions.

The Turing machine is an abstract model of a universal computer. From a more practical angle, since algorithms are composed of extremely simple steps, they can be delegated to real machines: Algorithms can be executed mechanically, without human intervention or appeal to human intelligence. In this sense, an algorithm is "mindless", once it is available. But, of course, creativity is needed to get to that point, i.e., to *discover* an algorithm for a given problem. This creative process will be our focus. But beware:

Myth: Developing algorithms is programming. – Not true!

Of course, computer programs are nowadays the way to execute algorithms. But algorithms existed already long before the advent of computers. Actually, the word is derived from the name al-Khwarizmi, a Persian scholar (780–850) who wrote an early textbook on arithmetic calculations. Various nontrivial algorithms were already known to ancient mathematicians, such as Euclid's algorithm for the greatest common divisor, and Eratosthenes' sieve method for finding the prime numbers.

Some of the simplest examples of algorithms are the well-known "school methods" for adding or multiplying two numbers "on paper". In fact, they are algorithms: They specify which simple operations with digits we have to do in which order, and how these partial results have to be combined to the correct final result.

We emphasize that algorithms and computer programs are different things. A program implements an algorithm, i.e., it realizes an algorithm in a specific programming language. But an algorithm as such is an abstract mathematical entity. This distinction is not just an academic question, but it has at least two major practical consequences:

1. When we want to **solve a new problem**, we should first focus on the problem itself, analyze it, and develop an (abstract) algorithm, without already worrying about implementation details and coding tricks. At this stage this would only distract attention from the actual problem. Algorithm design happens before any line of code is written. This process of algorithm design, but not the programming, is the main subject of this course. Except for very trivial problems, implementing the first quick ideas that come into mind would only lead to bad, slow, or even incorrect programs.

2. How do we **explain** algorithms, especially new algorithm, to other people? Program code is hard to read. Code is perhaps even the worst way to explain an algorithm, even when extensive comments are added. The same remark that we made on algorithm development applies to the understanding of algorithms: Implementation details that depend on a programming language can easily obscure the actual idea and the structure of the algorithm as such. **In this course, never describe algorithms by code!** Instead, use natural language and explain how they work. But still you need to be **precise** and unambiguous, therefore use mathematical notation whereever appropriate. It might be hard to find the right balance, but a criterion for a good algorithm description is: The description is readable, and a skilled programmer would be able to fill in the details and to implement the algorithm, using your description only. Commented **pseudocode** is a compromise between code and purely verbal descriptions. Algorithms written in pseudocode look like programs in usual procedural programming languages, but they are freed from non-essential or straightforward details.

More generally (not only in the algorithms field), it is not an exaggeration to say that clear, structured, and reader-friendly technical writing is a challenge in itself, and it must be practiced.

Time Complexity

Let x, y be two given integers. What is easier: to add them (compute $x + y$) or to multiply them (compute $x \cdot y$)?

Most people would spontaneously say that addition is easier than multiplication. But in what sense is it easier? It is natural to consider a problem "easy" if we can solve it by some *fast* algorithm.

Time complexity, that is, the time needed to solve a problem, is the most important performance measure for an algorithm. The amount of other resources (memory, communication, etc.) can also be relevant, but time has a special role: Because every action needs time, the time complexity limits the use of other resources as well.

But what could be a meaningful definition of time complexity?

Here is an attempt: Time complexity of an algorithm is its running time for every instance. More formally, we define time complexity as a function that assigns a positive real number (the running time) to every instance.

However, this definition would not be practical. The exact time for every isntance can be an extremely complicated, even incomprehensible, function.

On the other hand, it is not really necessary to know the exact running time on every instance.

We can greatly simplify the matter by only specifying the running time on instances of any given **instance size** n. That is, we define time complexity as a function from the positive integers (the instance sizes) into the positive real numbers (the running times). We may take the the **maximum** or the **average** running time for all instances of size n, and speak of **worst-case** and **average-case** time complexity, respectively. Moreover, instead of the exact maximum it is good enough to know some (close) upper bound.

Such worst-case bounds provide guarantees that an algorithm stops after at most that time. For certain algorithms, worst-case bounds can be too pessimistic: An algorithm might run fast on typical instances and need very long times only for some rare malicious instances. Then average-case analysis is more appropriate. However, in this course we will mainly consider worst-case bounds, unless stated otherwise.

But the above draft of a definition is not yet usable either. The next issue is: What should be the meaning of the positive real number that indicates the running time for a certain input size n? Counting seconds does not make sense here, because the physical running time depends on things like the speed of our processor, minor implementation details, and other contingencies that have nothing to do with the algorithm itself. (Recall that an algorithm is an abstract mathematical object, not a particular implementation on a particular machine.) Hence absolute figures for each n do not say much. Still, the *time complexity function in its entirety* is a meaningful object: If machine A is faster than machine B by some factor c, then any algorithm implemented on A will run c times faster than on machine B, but the "shape" of the function remains invariant. Therefore we will usually ignore constant factors the time complexities. Instead of the physical time we only consider the *growth* of the time complexity as a function of n. Of course, this is an abstraction. In practice we can ignore constant factors only if they are not unreasonably large.

But, in order to specify the time, we still need to count something! What if not seconds? – Recall that the work of any algorithm can be split into very simple steps. We call them **elementary operations** or **computational primitives**. We simply count these elementary operations carried out by an algorithm. Note that this number depends only on the algorithm (and the instance) but not on the physical circumstances.

Still a small issue remains: The definitions of what is considered elementary operations may be a bit arbitrary: Among other things, they depend

on the type of data we have to deal with. Moreover, we should not "compare apples and plums", that is, operations declared elementary should have similar execution times in reality, so that we can reasonably assume that the number of operations is proportional to the running time. Similar remarks apply to the definition of input size n. Usually, n is the number of symbols needed to write down an instance, but for each data type we must agree on some concrete definition.

For example, for analyzing arithmetic operations with integers like addition and multiplication, it is sensible to count operations with single digits. Accordingly, our elementary operations are: addition and multiplication of two digits (with carry-over), and reading or writing a digit. The size n of an instance is simply the number of digits.

It is important not to confuse this instance size n with the numerical values of the numbers to be processed! Their difference is tremendous: The size is only logarithmic in the numerical value. Conversely, this means that the numerical value is exponential in the size.

For algorithmic problems involving vectors and matrices of real numbers, it can be appropriate to consider additions and multiplications of entire numbers (rather than digits) as elementary operations. For other types of data, such as sequences or graphs, meaningful definitions of elementary operations and instance sizes must be adopted in an ad-hoc way.

It should also be noticed that we always analyze mathematical **models** of time complexity, rather than computations in specific real computer processors. But, of course, we try to keep our simplifying model assumptions close to reality.

Big-O Notation

The notion of upper bounds suppressing constant factors is formalized by the **O-notation**: For two functions t and f from the positive integers into the positive real numbers, we say that t is $O(f(n))$ (speak: "O of f" or "big-O of f" or "order of f"), if there exists a constant $c > 0$ such that $t(n) \leq cf(n)$ holds for all n, with finitely many exceptions. Informally that means: t will eventually grow no faster than f.

The O notation comes comes from the field of Mathematical Analysis, but here we will use it to express time bounds of algorithms. Typically, t is the exact running time, and f is some (usually simple!) function that bounds the running time.

One should not completely forget that constant factors are ignored. In some cases these hidden constants can be huge, and then expressions like $O(n)$ bogusly suggest practical algorithms. But apart from such exceptions, usually the hidden constants are moderate.

There is also a notational issue: To be mathematically strict, one should define $O(f)$ as the *class* of all function t with the mentioned property, and write $t \in O(f)$. Instead, it is quite common to use the convenient but inaccurate notations $t = O(f(n))$ or $O(t) = O(f)$. They are not meant as equations but as shorthands for "t is $O(f)$". Therefore be careful: From $O(t) = O(f)$ one cannot conclude $O(f) = O(t)$.

First Example: A Comparison of Arithmetic Operations

Now we can precisely say in what sense addition is easier than multiplication. Remember the conventions regarding elementary operations and instance size.

Adding two integers of length n requires obviously $O(n)$ time. This time is optimal, that is, no faster algorithm for addition can exist. The reason is simple: Since the sum depends on every digit, we must at least read the whole input, which costs already $O(n)$ time.

Next, adding m numbers, each with n digits, requires $O(mn)$ time. This is no longer obvious, because of the carry-over! But with some care one can, in fact, prove an $O(mn)$ time bound. This time bound is also optimal, for the same reason as above.

What about multiplication of two integers with n digits? The algorithm one usually learns at school reduces this problem to the addition of n integers, each with $O(n)$ digits. Due to the previous statement, this gives the time bound $O(n^2)$. Is this optimal as well? The trivial lower-bound argument used above says only that we cannot be faster than $O(n)$ time. Still this leaves hope for a multiplication algorithm faster than $O(n^2)$.

An indication that the usual method for multiplication might not be the fastest one is that the n partial results to be added are not "independent" integers: In the decimal system, at most 9 different sequences of nonzero digits can appear as summands, as we multiply every digit of one factor by the entire other factor. But the usual algorithm reads all these partial results repeatedly. Maybe this is not necessary. Maybe the algorithm repeats calculations that have already been done elsewhere. On the other hand, it

is not easy to see how we could take advantage of these special summands. Hopefully this discussion makes you curious about the existence of a faster multiplication algorithm.

Some Useful Properties of Big-O

First and foremost, $O(f(n)+g(n))$ equals $O(\max(f(n), g(n)))$. (The simple proof is omitted here.) It follows that, in an upper bound consisting of a sum of terms, only the worst term is important, i.e., the function with the biggest growth. In particular, if the bound is a polynomial of degree d, then only this highest degree is significant, but neither the coefficients nor the minor terms. Formally:

$$c_0 n^d + c_1 n^{d-1} + c_2 n^{d-2} \ldots = O(n^d).$$

This property makes O-expressions typically very simple. Lengthy sums of terms appear naturally in the time analysis of algorithms, since most algorithms consist of several parts, often with nested loops and other structures. Nevertheless, the overall time bound is usually a simple standard function.

This property has yet another nice aspect: As pointed out earlier, the definitions of instance size and of elementary operations with data are a little arbitrary. But due to the neglect of constant factors and minor summands, the time complexity of an algorithm expressed in O-notation is not sensitive to all these arbitrary choices in the details of the definitions. Any different but "natural" definitions yield the same O-bounds. In this sense, O-bounds are robust, and they are objective performance measures.

As we want to compare algorithms by their speed, we should be able to compare the growth of several standard functions, and also get a feeling for growth rates. A useful general result says: If f is any monotone growing function, and $c > 0$, $a > 1$ are constants, then $(f(n))^c = O(a^{f(n)})$. (Again we omit the proof. It needs some mathematical analysis, however this result is plausible.) We give two important examples of the use of this result: With $f(n) = n$ we get $n^c = O(a^n)$. In words: "polynomial is always smaller than exponential". With $f(n) = \log_a n$ we get $(\log_a n)^c = O(n)$. In words: "polylogarithmic is always smaller than linear."

Logarithms are common in time bounds, especially in certain algorithms that successively halve the input. A frequent constellation is that we have a bad $O(n^2)$ time algorithm and a clever $O(n \log n)$ time algorithm for the

same problem, and then we should appreciate that the latter one is significantly faster: $n \log n = O(n^2)$, but not vice versa.

Note that we may write $\log n$ in O-terms without mentioning the logarithm base, since logarithms at different bases differ by constant factors. (In the case that this is not clear, it is advisable to recapitulate the laws of logarithms ...)

A convenient way to prove O-bounds is to consider limits of ratios. For example, for any fixed c the following implication holds:

$$\lim_{n\to\infty} t(n)/f(n) = c \geq 0 \Longrightarrow t = O(f).$$

Finally we emphasize the special role of **polynomial bounds**. If the size of the instance of a problem is doubled, we would like the time to grow by only a constant factor, too. That is, the time bound f should satisfy $f(2n) \leq cf(n)$ for some constant c. This condition can be rephrased as $f(n) \leq n^d$ for some constant exponent d. Thus, in general we consider an algorithm "efficient" only if it has a polynomial time bound. For example, the known algorithms for addition and multiplication have polynomial time bounds.

Problem: Interval Scheduling

Given: a set of n intervals $[s_i, f_i]$, $i = 1, \ldots, n$, on the real axis. (An interval $[s, f]$ is defined as the set of all real numbers t with $s \leq t \leq f$.)

Goal: Select a subset X of these intervals, as many as possible, which are pairwise disjoint.

Remark: We may suppose that all $2n$ start and end points are distinct. Otherwise we can make them distinct by slightly extending some intervals, without creating new intersections.

Motivations:

Some resource is requested by users for certain periods of time, described by intervals with start time s_i and finishing time f_i. That is, a problem instance is a booking list with n intervals. Unfortunately, the intervals of many requests may overlap, because reservations have been made independently by several users. Our goal is to accept as many as possible of these requests.