Algorithms. Lecture Notes 1

- These notes are based on: Kleinberg, Tardos, *Algorithm Design* and 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 omitted.
- The contents follow the lectures, but they may differ from what was exactly said in class.

About Algorithms in General

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? For a well motivated answer we need some conceptual framework.

First of all, addition and multiplication are examples of **problems**: Some input is given, and we expect a solution. Here the term "problem" refers to the general task, whereas a concrete input is called an **instance** of the problem.

In this terminology, adding or multiplying *any* two numbers is a problem, and the given numbers x, y form an instance of this problem. In general, a problem has infinitely many possible instances, at least in principle.

Problems are often solvable by **algorithms**. Intuitively, an algorithm is an instruction saying how to do the calculations that solve the problem *for every possible instance*. An algorithm must be precise and unambiguous, such that it can be delegated to a machine. The steps of an algorithm consist of simple operations which can be done mechanically, without human intervention or appeal to human intelligence. They must not leave any room for subjective interpretations.

There also exist precise formal definitions of what an algorithm is. It is believed that these definitions capture the intuitive notion of algorithm (Church-Turing thesis), but this touches on philisophical questions and is quite outside the scope of this course.

Myth: Developing algorithms is programming.

Computer programs are nowadays the way to execute algorithms, but algorithms existed already long before the advent of computers. The word is derived from the name al-Khoarizmi, a Persian scholar (780–850) who wrote an early textbook on arithmetic calculations. Much earlier, various nontrivial algorithms were already known to ancient mathematicians, such as Euclid's algorithm for the greatest common divisor, and Eratosthenes' sieve methods for determinining the prime numbers.

Among the simplest examples of algorithms are the well-known "school methods" for adding or multipling two numbers "on paper". These methods fulfill all the criteria given above. 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 stress that an algorithm and a computer program is not the same thing! 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 more than just academic hair-splitting, rather, it has two major practical consequences:

"Would Descartes have written programs in Pascal?" (U. Eco)

"Computer science is no more about computers than astronomy is about telescopes." (E.W. Dijkstra)

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 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. (Recall: the algorithms for arithmetic calculations came much earlier than any computing devices.) 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 ones, to other people? Program code is hard to read. Perhaps it is even the worst way to explain an algorithm, even when extensive comments are added. The same remark as for algorithm development applies to the understanding of algorithmns: Implementation details depending on a programming language can easily obscure the actual idea and 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 symbols and mathematical notation whereever appropriate. A criterion for a good algorithm description is that it is readable, and that 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 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.

Time Complexity

Back to the first question: In what sense is addition easier than multiplication? 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.

"What is time? If nobody asks me for it, I know it; if I want to explain it to somebody asking, I do not know it." (Augustinus)

Here we do not start a philosophical discussion about time, we only clarify how the time complexity of algorithms should be defined in a meaningful way.

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

However, this function can be extremely complicated, even incomprehensible. On the other hand, we need not know the exact running time on every instance. A more practical approach is to state the running time on instances of any given **input length** n. That is, we define time complexity as a function from the positive integers (input size) into the positive real numbers (running time). 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. Instead of the exact maximum it is 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 time only for some rare malicious instances. Then average-case analysis is more appropriate. In this course we will mainly consider worst-case bounds, unless stated otherwise.

But the above draft of a definition is not yet satisfactory. 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, 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.) Hence absolute figures for each n do not say much. Still the time complexity *function as a whole* 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 in the time complexity. Of course, in practice we can do so only if the factors are not too large.

But still we must count something. What else, if not seconds? Since constant factors are ignored, we may simply count **elementary operations**, also called **computational primitives**. Recall that the work of an algorithm can be split into simple steps. We just assume that every step requires one time unit. A slight problem is that definitions of what is considered elementary are a bit arbitrary: They depend on the type of data we have to deal with. Moreover, we should not "add apples and plums", that is, operations declared elementary should have similar execution times in reality, so that the unit-time assumption is a reasonable simplification. Similar remarks apply to the definition of input

size n. Usually, n is the number of symbols needed to write down an instance, but in each case we must agree on some concrete definition.

For example, for reasoning about 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, and reading or writing a digit. The size n of an instance is simply the number of digits. It is important not to confuse the input length n with the numerical size of the numbers x, y to be processed! The input length is logarithmic in the value, and conversely, the value is exponential in the input length.

For algorithmic problems with other types of data, such as graphs, appropriate definitions of elementary operations and input length must be adopted. It should also be noticed that we analyze mathematical **models** of time complexity rather than real computations in computer processors. This is a conceptual difference, albeit, of course, we hope that our simplifying model assumptions are close to reality.

Big-O Notation and a Comparison of Arithmetic Operations

The notion of upper bounds ignoring 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 not faster than f. When we express time bounds for algorithms, t is the exact running time, and f is some (usually simple!) function that we use as our bound.

> *"Researchers in algorithmics do not care about their salaries, because they ignore constant factors."*

Od course, this quote is not seriously meant. We are talking about growth of functions, not about single numbers. Nevertheless, one should not completely forget that we ignored constant factors. In some cases the hidden constant is huge, but expressions like $O(n)$ bogusly suggest practical algorithms. But apart from such exceptions, usually the hidden constants are moderate.

Authors often 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 $O(f)$". Thus, be careful: From "$O(t) = O(f)$" one cannot conclude that "$O(f) = O(t)$".

Now we can precisely say in what sense addition is easier than multiplication. (Remember the agreements on elementary operations and input 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: Since the sum depends on every digit, we must at least read the 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 carryover. 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 in the summands, as we multiply every digit of one factor with the other factor. But the usual algorithm reads all these partial results again. 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. Interesting?

Some Useful Properties of Big-O

First and foremost, $O(f(n) + g(n))$ equals $O(\max(f(n), g(n)))$. The proof is very simple, just go back to the definition of O. It follows that in a sum of upper bounds only the worst term is important, i.e., the function that grows most. In particular, if the bound is a polynomial of degree d, then only this highest degree is significant, but neither the coefficients nor minor terms:

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

This makes O-expressions typically very simple. Complicated sums 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 of O-expressions to be simple has another nice aspect: As pointed out earlier, the definitions of elementary operations on data are a bit arbitrary. But due to the neglect of constant factors and minor summands, the complexity of an algorithm expressed in O-notation is not affected by all these arbitrary details of definitions. In this sense, O-bounds for the time complexity of algorithms are robust and objective performance measures.

As we want to compare algorithms by 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 $(f(n))^c = O(a^{f(n)})$. We omit the proof which needs some mathematical analysis. We just give wo 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 inputs. 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 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. (It is advisable to recall the laws of logarithms ...)

A convenient way to prove O-bounds is to consider limits of ratios. For example, we have for any fixed c: If $\lim_{n\to\infty} t(n)/f(n) = c \geq 0$ then $t = O(f)$.

Finally we emphasize the special role of **polynomial bounds**. If the input size for 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 s_i and finish 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.

Problem: Interval Partitioning

Given: a set of n intervals $[s_i, f_i]$, $i = 1, \ldots, n$, on the real axis.

Goal: Partition the set of intervals into the smallest possible number d of subsets $X_1, X_2, X_3, \ldots, X_d$, each constisting of pairwise disjoint intervals.

Motivations:

Similar to Interval Scheduling. The difference is that several "copies" of the resource are available, and *all* requests shall be served, using the smallest number of copies.