Algorithms. Lecture Notes 1

These Lecture Notes are mainly based on: Kleinberg, Tardos, *Algorithm Design*, and they are also influenced by other books and materials.

Several sections have "Problem" in the headline. They specify the examples of computational problems that we discuss next in the course. It is recommended that you:

- First, familiarize thoroughly with the problem specification, maybe draw some small examples.
- Then, think a while for yourself: How would I approach this problem, with my current knowledge?
- Finally, continue and learn about the proposed solutions.

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. This is true! 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 first have to formalize 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, an intuitive understanding of the concept is sufficient as long as we are concerned with specific algorithms for specific problems. 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 (such as the halting problem).

Nevertheless, in order to emphasize one typical feature of algorithms we mention one of these formal definitions – the Turing machine. 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, or write a symbol. Moreover, the next action can be conditional on what symbol has been read, and on an internal state. The whole control unit consists of a table that says what action shall be performed next, depending on the current state and on the symbol read. The *Church-Turing thesis* holds that all algorithms can be formalized in this way, that is, other kinds of steps are never needed. However, 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 we can say: 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, as opposed to this, creativity is needed to get to that point, i.e., to *discover* or *develop* 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 deserve the name algorithm: They specify which simple operations with digits we have to do in which order, and how these partial results have to be combined to get 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 matter, but it has at least two major practical consequences:

1. Whenever 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? Long chunks of program code are 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 the 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, however, as a guideline you may use these criteria for a good algorithm description: It is readable like a good manual, 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 **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 the indicated 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 basic 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 in 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 unreasonably small.

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 certain doubt 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 rare 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 functions 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 very 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.

Regarding 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 yields 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": 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. All "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 a bad $O(n^2)$ time algorithm against a clever $O(n \log n)$ time algorithm for the same problem. Then we should appreciate that the latter one is significantly faster: we have $n \log n = O(n^2)$, 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.

Appendix

Here are some optional self-test questions about O-notation. Try to answer them, just for yourself. You may be wondering: "How do I know that my answers are correct?" But that's the point! If you have understood the subject very well, you will be sure about your answers. This also implies: If you feel uncertain, you should discuss them with us or with your classmates.

- $O(3n^2 + n) = O(n^2)$ – is this correct?
- $O(n \log n) = O(n)$, because $\log n$ is very small compared to n and can be neglected – is this correct?
- $n \log n$ is in $O(n \log \log n)$ – is this correct? And in the other direction?
- Is $O(n + \ldots + n) = O(n)$ correct? Does the answer depend on the number of summands?
- $O(n + n/2 + n/4 + n/8 + \ldots) = O(n)$ – is this correct?
- $O(n^{\sqrt{n}}) = O(2^n)$ – is this correct?
- Integers can be represented in the decimal system (base 10), the binary system (base 2), or using any other fixed base. We stated that two integers with n digits can be added in $O(n)$ time. Is this true for every fixed base?
- Given an integer z with n digits, we can check whether z is a prime number by generating all products $x \cdot y$ with $x, y < z$ and compare them to z. (This is not a very smart algorithm, but let us ignore this fact.) What do you think: Is the time of this algorithm polynomial in n?
- The definition of $t = O(f)$ allows the violation of $t(n) \leq cf(n)$ for finitely many n. Would we obtain an equivalent definition if we forbid such exceptions? Furthermore, do you have an idea why such exceptions are useful when we show time bounds for algorithms?

Algorithms. Lecture Notes 2

An Algorithm for Interval Scheduling

According to the spirit of the course, this section illustrates a possible *process* of algorithm *development* for a problem, rather than giving a good algorithm right away.

A naive algorithm would examine all subsets of the given set of intervals and therefore run in $O(n^2 2^n)$ time, where the n^2 factor accounts for checking the validity of a chosen subset. (Do not worry about the details of a poor algorithm. The inportant fact here is that more than 2^n steps are used.) Let us try and develop a much, much faster algorithm.

Here is a good heuristic question for algorithm development in general: *Suppose that we are already able to solve the problem for smaller instances, how can we use the partial solutions to solve the overall instance?* Since instances of most computational problems can be split into smaller instances of the same problem in some natural ways, this is a very fruitful approach.

In the case of Interval Scheduling we may ask more specifically: Suppose that we know already how to find the best solution for less than n intervals. Can we perhaps make a decision for one interval x (that is, to put it in the solution X) and then solve the remaining instance?

We can express our wish more explicitly: We would like to find some general rule that determines one interval x that we can safely put in X. Then we could remove all intervals that intersect x (this is enforced by the problem), and continue applying our rule until the instance is empty.

However, we have many options to choose this interval x.

A tempting idea is to serve the first request, i.e., let x be the interval with smallest s_i. But the drawback is obvious: This first interval could be very long. It could even intersect all others, and then it is a bad choice.

Perhaps we should take the lengths $f_i - s_i$ into account? Let us make another attempt: Let x be the shortest interval. The intuition is that, typically, short intervals should not intersect many others. – But unfortunately,

"typical" is not enough. The shortest interval does not necessarily belong to an optimal solution either. The smallest counterexample has only three intervals and is easy to see!

It is time to analyze the reasons for these failures and to learn from them. Our selection rules were bad because the selected intervals may overlap too many other intervals. This suggests yet another idea: Let x be some interval that intersects the smallest number of other intervals. – This sounds very plausible, but sadly this rule fails, too, although this time it is a little harder to find a counterexample. The idea of a counterexample is to work with identical copies of some intervals. Since intervals in X must be pairwise disjoint, this does not affect the optimal solution. But by chosing suitable numbers of copies, one can give some interval $x \notin X$ in the middle the smallest number of overlaps, such that the algorithm would wrongly choose x. This way one can construct a counterexample with 11 intervals.

So the third attempt failed, too. We may try further rules until we are lucky, or we may decide to give up at some point. One last attempt: What else could be a good candidate for the interval x?

> *"Ever tried. Ever failed. No matter.*
> *Try again. Fail again. Fail better."*
> (Samuel Beckett)

Counterexamples are not unfavorable. While they prove incorrectness of some specific algorithm, they can also give valuable hints on what exactly goes wrong. Reviewing the above cases again, we may notice that, in the last two attempts, the selected intervals were somewhere in the middle of the schedule. What if we come back to the original idea and look at the beginning of the schedule? But taking the interval with earliest starting point was bad. What if we instead take the interval with the earliest endpoint!? The rationale is that this interval is in conflict with the smallest number of other intervals in the remaining instance to the right of its endpoint. For clarity, let us write down the proposed algorithm explicitly:

Earliest End First (EEF): Sort the intervals according to their right endpoints. That is, re-index them such that $f_1 < f_2 < \ldots < f_n$. Put the interval $[s_1, f_1]$ (the one with the smallest f_i) in X, and delete all intervals that intersect this first interval. Repeat this step until every interval is either in X or deleted.

This time we will not detect counterexamples. But after the bad experiences where we saw plausible rules breaking down, it should be clear that we need a **correctness proof**. It is not enough to say that no counterexamples are known. There might exist some, but they might be relatively large and non-obvious (as it happened with the last wrong algorithm above). Now, here is a proof of optimality:

Assume that there exists a counterexample, that is, an instance where EEF fails to return an optimal solution Y. That is, the solution X from EEF has size $|X| < |Y|$. Let x be the interval with the earliest end. If $x \notin Y$, then we take the leftmost interval $y \in Y$ and exchange it: Let $Y' = Y \setminus \{y\} \cup \{x\}$. Since x has the earliest end, Y' is also a set of disjoint intervals. Moreover, $|Y'| = |Y|$. Hence Y' is another optimal solution. This shows that some optimal solution Y' with $x \in Y'$ exists.

Informally, we have shown that "it is not a mistake" to choose interval x in the first step, as EEF does. It is also worth noticing that the defining property of x is essentially used in the proof. Of course, the rule of the algorithm must play some role.

But so far we have exchanged only one interval. How does this imply correctness of EEF? We can do a proof by contradiction:

Remember that we assumed a counterexample, where EEF returns some set X being smaller than an optimal solution Y. After deletion of x and of all intersecting intervals, there remains a smaller instance where EEF returns $X \setminus \{x\}$. But $Y' \setminus \{x\}$ is a valid solution, too, and $|X \setminus \{x\}| < |Y' \setminus \{x\}|$. Hence the EEF solution is not optimal on this smaller instance either. That is, we have found a smaller counterexample!

Hence we have shown: For every counterexample to EEF there exists another counterexample with fewer intervals. On the other hand, some counterexample must have the minimum size. This contradiction proves that the initial assumption (existence of some counterexample) was wrong. Thus EEF is correct.

We remark that the same proof can also be formulated as induction on the number of intervals, which is logically equivalent. But the present formulation via a smaller counterexample might appear more intuitive and elegant.

The next step is to think about the implementation details that make the algorithm efficient. We had already sorted the intervals in such a way that $f_1 < f_2 < \ldots < f_n$. Now we may scan this sorted list from left to right, and record the currently last interval $[s_j, f_j] \in X$. When we read the next f_i, we simply check whether $s_i < f_j$. If so, then we skip $[s_i, f_i]$, since this

interval cannot be in X. If not, then we add $[s_i, f_i]$ to X, according to the rule of EEF. Hence this is our new $[s_j, f_j]$. Since we spend only $O(1)$ time on each interval, we need $O(n)$ time in total, plus the time for sorting.

Note that, in this implementation, intervals not chosen in X are merely skipped rather than "physically" deleted. But this does not make a difference, because skipped intervals are never considered again. The formulation with deletions was better suited for understanding the algorithm itself and its correctness, but for the sake of speed, the proposed implementation handles this detail differently. This illustrates again that one should first care about solving the problem, and only later about programming details and fine-tuning.

To What End Do We Study Algorithm Theory?

Myth: Fast algorithms are not needed (because the hardware is so fast). – Not true!

From comparing the growth of different functions it should be clear that the O-complexity of an algorithm has a larger impact on its practicality than the speed of processors.

Next, since an algorithm for a problem must be created only once but will be applied many, many times, good algorithm design ultimately will pay off.

Therefore it is important to solve various computational problems by *fast* algorithms. Does this mean that we have to learn a suite of fast algorithms for the most frequent problems? Yes, but this is not enough. Practical problems rarely arise in nice textbook form, and usually we cannot simply take an algorithm from the shelves. Often we must adjust or combine algorithms that are known for similar problems or for parts of a given problem. In order to be able to do all this, we need a profound understanding of **how and why** these algorithms work. We have to understand the underlying ideas, not only the particular steps.

Moreover, new computational problems will in general require new algorithms. There is no universal recipe for designing good algorithms, except some general guidelines and techniques. The main part of this course provides some of these general design techniques, a basic toolkit so to speak. We illustrate and practice them on various problem examples. But still the actual algorithm design for a given problem remains a trial-and-error

process. (Compare it to craft: Even if one knows one's trade, every application is a bit different, and one must extemporize.) The selected problems are, hopefully, also of some relevance by their own, but the emphasis is on the design process, rather than on the ready-to-use algorithms for specifc problems.

We also have to practice **proving correctness** of new algorithms.

Myth: Correctness proofs are not needed, it suffices to test algorithms on some instances. – Not true!

Recall the Interval Scheduling example. Various algorithms appeared to be plausibe, but they failed. By not caring about correctness proofs we may happily accept erroneous algorithms. Proofs are not a luxury, just made to intellectually please a few researchers, and testing alone cannot guarantee correctness. We may pick, by good luck, some test instances where our algorithm yields the correct results, but it may have a hidden error that shows up in other instances. This can have fatal consequences, especially in sensible technical systems controlled by algorithms, and so this matter touches even questions of ethics in engineering.

Now we can summarize our main goal more precisely: *Develop correct algorithms with low worst-case time bounds which are, preferably, moderate polynomials.*

About Greedy Algorithms

Earliest End First is an example of a **greedy algorithm**. These are algorithms which, in every step, make the currently best choice, according to some simple optimality criterion. (Once more this is not a formal definition, just a circumscription.) In this sense, greedy algorithms are "myopic".

Myth: Take the best, ignore the rest. If we take an optimal decision in every step, then the overall result will be optimal, too. – Not true!

"Greed is good. Greed is right. Greed works." claimed the broker Gordon Gekko, a character in the 1987 film "Wall Street".

Actually *most greedy algorithms are plainly wrong*, and counterexamples are often amazingly small. As we have seen, we do need correctness proofs. The key step of such a correctness proof is often an **exchange argument**:

One item of an optimal solution Y is exchanged, in such a way that Y gets closer to the solution produced by the algorithm, without making it worse. For example, in the correctness proof of EEF we took the leftmost interval of Y and replaced it with the interval with the earliest end in the whole instance (which the algorithm would have chosen). Then, the exchange argument is embedded in an inductive proof or in a proof by contradiction.

Problem: Weighted Interval Scheduling

Given: a set of n intervals $[s_i, f_i]$, $i = 1, \ldots, n$, on the real axis. Every interval has also a positive weight v_i.

Goal: Select a subset X of these intervals which are pairwise disjoint and have maximum total weight.

Motivations:

Similar to Interval Scheduling, but here the requests have different importance. The weights might be profits, e.g., fees obtained from the customers.

Algorithms. Lecture Notes 3

An Algorithm for Weighted Interval Scheduling

After the Interval Scheduling success we dare to attack a more general problem –Weighted Interval Scheduling. The natural first thought is to try and generalize the EEF algorithm directly. Again we sort the intervals such that $f_1 < f_2 < \ldots < f_n$. However, because of the different weights v_i it is no longer true that we can always put the first interval in an optimal solution X. This interval could have a small weight and intersect some other, more profitable intervals. This makes the problem essentially more difficult than Interval Scheduling. The weights "add another dimension" to the problem. But can we perhaps extend solutions of smaller instances to larger instances in some more sophisticated way?

We may decide for each interval in the sorted sequence to add it to X or not. This sounds like exhaustive search. However, a striking observation regarding the "interval structure" of the problem limits this combinatorial explosion: Suppose that we have decided the status of the first j intervals (to be in X or not). Consider all partial solutions where $[s_j, f_j] \in X$. Among all these (exponentially many!) partial solutions that end in f_j, it suffices to keep only one(!) partial solution X_j with maximum total weight. Why is this correct? Let X be any overall solution that contains $[s_j, f_j]$. If X_j is not a subset of X, then we can replace the subset of intervals in X that end before f_j with X_j, to obtain a solution that is no worse. Hence it is safe to assume $X_j \subset X$.

In the following we state the resulting algorithm, along with the correctness arguments.

(1) Defining the objective value:

For $j = 1, \ldots, n$, we define $OPT(j)$ to be the maximum weight that can be achieved by selecting a subset of disjoint intervals from the first j intervals, i.e., from those with endpoints $f_1 < f_2 < \ldots < f_j$. (Note carefully that this definition does not require to put $[s_j, f_j]$ in the solution; this interval may be chosen or not.) Our final goal is to evaluate $OPT(n)$.

(2) Computing the objective value:

"To add or not to add, that is the question."

We can inductively compute every $OPT(j)$ from the previously computed values $OPT(i)$, $i < j$: By definition we have $OPT(1) = v_1$. Next, suppose that all $OPT(i)$, $i < j$, are already computed. For the interval $[s_j, f_j]$ we have two options: to add it to the solution or not. If we don't, then the best total value is clearly $OPT(j-1)$. If we decide to put $[s_j, f_j]$ in the solution, then it contributes v_j to the total value, but we have to make sure that the new interval does not intersect any earlier one. For this step we need some **auxiliary function**: Let $p(j)$ be the largest index i such that $f_i < s_j$. We can take the known solution with value $OPT(p(j))$ and add the new interval. By the observation above, only this best solution is needed in this case. Altogether we have shown that the following formula is correct:

$$OPT(j) = \max\{OPT(j-1), OPT(p(j)) + v_j\}.$$

This part of the algorithm amounts to a simple for-loop, with all $OPT(j)$ stored in an array. Of course, prior to this calculation we must compute and store all the $p(j)$ in another array. (The v_j, s_j, f_j are already given in arrays.) It is easy to compute the $p(j)$ in a single scan: We also sort the s_j in ascending order. Then we determine, for every j, the largest $f_i < s_j$. Since we have sorted the s_j, it suffices to move a pointer in the sorted array of the f_i. Hence we can compute all $p(j)$ in $O(n)$ time, plus the time for sorting. The for-loop that computes the $OPT(j)$ values needs $O(n)$ time, which should be obvious: In every iteration we do one addition and one comparison. (Here we assume that addition and comparison of two numbers are elementary operations.)

Note that the formula above is recursive: $OPT(j)$ is computed by recurring to function values for smaller arguments. But beware: It would be a big practical mistake to implement this formula in a recursive fashion, i.e., as a subroutine with recursive calls to itself! What would happen? Every call spawns two new calls, so that the process splits up into a tree of independent calculations, where the same $OPT(j)$ are computed over and over again in many different branches. (This does not happen if our *compiler*

recognizes repeated calls with the same input parameter and just returns the function value. But the algorithm itself should not rely on that.) The time would be exponential, thus destroying the whole idea that made the algorithm efficient, namely, to compute every $OPT(j)$ only once. This illustrates again the importance of *understanding the structure* of an algorithm. It is not enough to hack formulas in the computer.

Now, have we solved our problem? No. We have computed the *value* $OPT(n)$, but where is the actual solution, that is, a subset of disjoint intervals that realizes this profit? An obvious idea to get the solution would be: Whenever we compute and store a new value $OPT(j)$, we also store a corresponding set of intervals. (After all, we know whether the jth interval has been added to the solutuon for $OPT(j-1)$ or not.) However, this would require many copy operations and result in $O(n^2)$ time. Compared to exponential time this is very good, but unnecessarily slow nevertheless. Surprisingly, we can construct a solution much faster, using only the stored values $OPT(j)$:

Remember how we obtained $OPT(n)$. We compared two values, and depending on which one was larger, we took the nth interval or not. Just by reviewing the OPT values we see what the optimal choice was. Next we review either $OPT(j-1)$ or $OPT(p(j))$ in the same way, and we find out whether the considered interval was taken or not, and so on. In other words, we **trace back** the sequence of optimal decisions. By this procedure we can reconstruct some optimal solution in another $O(n)$ steps.

Dynamic Programming versus Greedy

The scheme used in the algorithm above is called **dynamic programming**, mainly for historical reasons. It can be characterized as follows.

For a given instance of a problem, we consider certain sub-instances that grow incrementally. For each of these sub-instances it suffices to keep one optimal solution (because, by an exchange argument, no other solution can lead to a better final solution for the entire instance). The optimal values are then computed step by step on the growing sub-instances. A "recursive" formula specifies how to compute the optimal value from the previously computed optimal values for smaller sub-instances. However, it is not applied recursively, rather, the values are stored in an array, and calculations happen in a for-loop.

This approach is efficient if we can limit the number of sub-instances to be considered, ideally by a polynomial bound. (This distinguishes dynamic programming from exhaustive search.) These sub-instances are often defined

by some natural restrictions, like the number of items, or some size bound.

The time bound is simply the size of the array of optimal values, multiplied by the time needed to compute each value.

Although this array displays only the optimal *values*, an actual solution is easy to reconstruct in a **backtracing** procedure where we examine, in reverse chronological order, on which way the optimum has been reached. The time for backtracing is no larger than the time for computing the optimal values, as we have to trace back only a path of calculations that were already done.

This outline may still appear a bit nebulous. The best way to fully understand dynamic programming is to study a number of problem examples of different nature, as we will do now. At some point one should notice that the basic scheme is always the same, and only the recursive formula and other specific details depend on the problem.

Dynamic programming can be viewed as restricted exhaustive search, but also as an extension of the greedy paradigm. Instead of following only one path of currently optimal decisions, which may or may not lead to an optimal overall solution, we follow all such paths that might bring us to the optimum. Of course, this is feasible only if there are not too many paths to examine.

It is very rewarding to learn this technique. Whereas greedy algorithms work only for relatively few problems, dynamic programming has considerably more applications. Our examples are taken from different domains.

Explaining Dynamic Programming Algorithms

Look again at the algorithm description in the previous section. It consists of twp parts: (1) Defining the objective value. (2) Computing the objective value. Note that these are two different activities!

When you explain an own dynamic programming algorithm, make sure that you write down also part (1), since otherwise it is (in general) not clear *what* you want to compute in part (2), let alone verification of correctness.

For example, assuming that $p(j)$ is defined, would you understand with ease what "$OPT(j) = \max\{OPT(j-1), OPT(p(j)) + v_j\}$" does, without being told before that $OPT(j)$ is supposed to be "the maximum weight that can be achieved by selecting a subset of disjoint intervals from the first j intervals"?

More generally, all newly introduced methematical symbols must be defined before they are used. (Otherwise, how does the poor reader know what they mean?) This should be obvious, but is easy to forget.

Problem: Knapsack

Given: a knapsack of capacity W, and n items, where the ith item has size (or weight) w_i and value v_i.

Goal: Select a subset S of these items that fits in the knapsack (i.e., with $\sum_{i \in S} w_i \leq W$) and has the largest possible sum of values $v = \sum_{i \in S} v_i$.

Motivations:

- Packing goods of high value (or high importance) in a container.
- Allocating bandwith to messages in a network.
- Placing files in fast memory. The values indicate access frequencies.
- In a simplified model of a consumer, the capacity is a budget, the values are utilities, and the consumer asks himself what he could buy to maximize his happiness.

Problem: Subset Sum

Given: n numbers w_i, $(i = 1, \ldots, n)$ and another number W. (All w_i are positive, and not necessarily distinct.)

Goal: Select a subset S of the given numbers, such that $\sum_{i \in S} w_i$ is as large as possible, but no larger than W. In particular, find out whether there is even a solution with $\sum_{i \in S} w_i = W$.

Motivations:

- This is a special case of the Knapsack problem where $v_i = w_i$ for all i. The goal is to make use of the capacity as good as possible.
- Manufacturing: Suppose that we want to cut up n pieces of lengths w_i $(i = 1, \ldots, n)$, and among our raw materials there is a piece of length W. How can we cut off some of the desired lengths, so that as little as possible of this raw material is left over?

Appendix

Calculation Example for Weighted Interval Scheduling

Suppose that we are given six intervals with these start points (s), end points (f), and values (v).

i	s(i)	f(i)	v(i)
1	0	3	2
2	1	5	4
3	4	6	4
4	2	8	7
5	7	9	2
6	7	9	1

They are already sorted by their end points. First we compute the auxiliary values $p(i)$. We just go through the list sorted by their start(!) points and count how many interval are already finished when a new interval begins. This way we get $p(1) = p(2) = p(4) = 0$, as interval 4 still begins earlier than the first interval ends. Next the counter increases to 1, thus we continue with $p(3) = 1$. Until the start of the last intervals, two further intervals end (namely intervals 2 and 3), hence the counter goes up to 3, and $p(5) = p(6) = 3$. The results are inserted in the table:

i	s(i)	f(i)	v(i)	p(i)
1	0	3	2	0
2	1	5	4	0
3	4	6	4	1
4	2	8	7	0
5	7	9	2	3
6	7	9	1	3

Now we can apply the formula to compute the optimal values.
$OPT(0) = 0$
$OPT(1) = 2$
$OPT(2) = \max\{OPT(1), OPT(0) + 4\} = \max\{2, 0 + 4\} = 4$
$OPT(3) = \max\{OPT(2), OPT(1) + 4\} = \max\{4, 2 + 4\} = 6$
$OPT(4) = \max\{OPT(3), OPT(0) + 7\} = \max\{6, 0 + 7\} = 7$
$OPT(5) = \max\{OPT(4), OPT(3) + 2\} = \max\{7, 6 + 2\} = 8$
$OPT(6) = \max\{OPT(5), OPT(3) + 1\} = \max\{8, 6 + 1\} = 8$

Finally the backtracing can start. We have obtained $OPT(6) = 8$, since $8 > 6 + 1$. Thus, interval 6 is not in the final solution. Rather, we took the solution from $OPT(5)$. We have obtained $OPT(5) = 8$, since $7 < 6 + 2$.

Thus, interval 5 is in the final solution, together with the intervals that contributed to $OPT(3)$, We have obtained $OPT(3) = 6$, since $4 < 2 + 4$. Thus, interval 3 is in the final solution, together with the intervals that contributed to $OPT(1)$, Clearly, the latter set consisits of interval 1 only. Altogether, our solution is formed by intervals 1, 3, 5.

Learning styles are different. If it helps you seeing an algorithm in action, in addition to the abstract reasoning, you are advised to implement some dynamic programming algorithm(s), even though implementation is not the focus of the course. As they typically consist of only a few for-loops, this is not a huge amount of work. When you develop own dynamic programming algorithms, you can then also test (but not prove!) that your idea works in exactly the proposed way, or recognize that you forgot some cases, or did other mistakes.

Algorithms. Lecture Notes 4

Dynamic Programming Algorithms for Subset Sum and Knapsack

A new feature of the next examples of dynamic programming is that the "dynamic programming function" has two parameters rather than one. For some reason this is a quite typical case. We will also see that the function is not always numerical; it can also have Boolean values, for example.

As an indication that dynamic programming (and nothing simpler) will be needed for the Knapsack problem, we begin with a natural greedy algorithm and a small but impressive counterexample where it miserably fails. Since we have to pack as much value as possible in a limited space, it is tempting to re-index the items such that $v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n$ and take the items with the best value-to-size ratios until the knapsack is full. However, consider the following instance, amazingly with only two items: $v_1 = 10\epsilon$, $w_1 = \epsilon$, $v_2 = 90$, $w_2 = 10$, $W = 10$. The optimal solution is item 2 with value 90, but the above greedy algorithm would take item 1, and this rules out the profitable item 2. By making $\epsilon > 0$ arbitrarily small, we get arbitrarily bad greedy solutions ... well, Let us turn to dynamic programming instead.

First we consider the Subset Sum problem in the case when an exact sum is required: Given numbers W and w_i, $i = 1, \ldots, n$, find a subset whose sum is exactly W, or confirm that no solution exists. We assume that all these numbers are integers. (Arbitrary rational numbers can be multiplied with their greatest common divisor, without changing the problem.) It is convenient to call W the capacity and to imagine that we pack items of sizes w_i in a knapsack.

An obvious idea for dynamic programming is: Consider the items in the given order and decide whether to choose the jth item or not. But, in contrast to Interval Scheduling, it is not enough to use j as the only argument in our objective function. Our decisions influence the remaining capacity, thus we must keep track of the capacity as well. Therefore we need a second argument, and we define: $P(j, w) = 1$ if some subset from the first

j items has the sum w, and $P(j, w) = 0$ else. Our function has Boolean values 1 (true) and 0 (false). This is appropriate because there is nothing to optimize here. In this problem we only want to know whether some solution *exists* or not.

The only value that we eventually want is $P(n, W)$. Suppose that we have already computed the $P(i, y)$ for all $i < j$ and $y < w$. If we do not choose the jth item, we just copy the solution for $j - 1$. If we do choose the jth item, the capacity used up before this step was by w_i units smaller. Since these are the only possible options, it is correct to compute each $P(j, w)$ by the following Boolean expression:

$$P(j, w) = P(j - 1, w) \vee P(j - 1, w - w_j).$$

As for the initialization, we have $P(0, w) = 0$ for all $w > 0$, and we have $P(j, 0) = 1$ for all j, since the empty set is always a solution with sum 0. We can also assume $P(j - 1, w - w_j) = 0$ for $w < w_j$, because no solution with negative size exists.

The number nW of sub-instances to consider is reasonably small. In every step we need to know which is the current item, and how much capacity is already used, and this information is enough for making the remaining choices.

The "art" of dynamic programming is to recognize such parameters that limit the number of sub-instances to consider for the given problem. This is the creative step which requires some problem analysis. But once we have found suitable parameters, the development of the algorithm is usually pretty straightforward.

Back to our problem: In the case that $P(n, W) = 1$, we can reconstruct a solution by backtracing (in the same way as earlier). The total time complexity is $O(nW)$, since the computation of every $P(j, w)$ needs $O(1)$ operations.

However, be aware that $O(nW)$ is not a polynomial time bound! Number W is exponential in its description length, since we need only $O(\log W)$ digits to write W. Hence nW cannot be polynomially bounded in the instance size. Still, if $W < 2^n$ then the dynamic programming algorithm is faster than exhaustive search. The condition $W < 2^n$ is often satisfied in practical instances.

Next we consider the more general optimization version of Subset Sum: If no subset has exactly the desired sum W, compute a subset with the largest possible sum below W. ("Pack a knapsack as full as possible.") The only new twist is that we must memoize the optimal sum rather than just a Boolean value. Accordingly, we define $OPT(j, w)$ as the largest number

not exceeding w that can be obtained as a sum of values w_i of some subset of the first j items.

Without much further explanation it should be clear that:

$$OPT(j, w) = \max\{OPT(j-1, w), OPT(j-1, w-w_j) + w_j\},$$

with the initialization $OPT(0, w) = 0$ for all w, and $OPT(j, 0) = 0$ for all indices j. To take care of the case $w - w_j < 0$ we can set $OPT(j, y) = -\infty$ for all j and for all $y < 0$.

Now we are ready to solve the general Knapsack problem with sizes w_j and profit values v_j, almost as a byproduct of our previous discussion. Define $OPT(j, w)$ to be the largest possible total *value* of a subset from the first j items with total size at most w. Because only some minor modification is needed, we give the resulting formula straight away:

$$OPT(j, w) = \max\{OPT(j-1, w), OPT(j-1, w-w_j) + v_j\}.$$

Only one symbol has been replaced in the formula – isn't that amazing?

Finally, consider a variant of the Knapsack problem where arbitrarily many copies of every item are available. Surprisingly, yet another slight modification of the recursive formula solves it immediately:

$$OPT(j, w) = \max\{OPT(j-1, w), OPT(j, w-w_j) + v_j\}.$$

Why is this correct? We leave it to you to think about it.

Problem: Segmentation

This is a generic scheme of problems, rather than one specific problem. Let f be some "easily computable" function that assigns a positive real number to every possible sequence of items. These items can be numbers, characters, or other objects.

Given: a sequence $(x_1, \ldots, x_n)$ of items.

Goal: Partition the sequence into segments $(x_i, \ldots, x_j)$ so that the sum of the values $f((x_i, \ldots, x_j))$ of all these segments is maximized/minimized.

Motivations:
f can be interpreted as a quality measure or a penalty for segments. Our segmentation shall maximize the total quality, or minimize the penalty. We mention a few concrete problem examples:

- *Data analysis:* A sequence of real numbers shall be partitioned into segments that ascend or descend almost linearly. The penalty for every segment is measured by the deviation from the closest linear function (regression line) by, e.g., the sum-of-sqares error. (This problem is treated in Section 6.4 of the textbook.)
- *Parsing:* A text without spaces shall be partitioned into words. The penalty for a segment is, e.g., its edit distance to the most similar real word in a dictionary.

Dynamic Programming for Segmentation Problems

We consider the penalty minimization version first. Let e_{ij} denote the penalty for segment $(x_i \ldots, x_j)$ in the given sequence. Being already trained in dynamic programming, we define $OPT(j)$ to be the smallest possible sum of penalties in a segmentation of $(x_1, \ldots, x_j)$. The last segment may start at any position $i \leq j$, therefore we have:

$$OPT(j) = \min_i OPT(i-1) + e_{ij},$$

where the minimum is taken over all i with $1 \leq i \leq j$. A new phenomenon is that we make a **multi-way choice** in every step. The number of cases is no longer constant, and it takes $O(j)$ time to evaluate every $OPT(j)$. Thus, the time complexity is $O(n^2)$, plus the time for computing

all e_{ij}. It depends on the penalty function how difficult these computations are.

In enhanced versions of segmentation problems, only some maximum number of segments may be allowed in a segmentation. Then, our dynamic programming formula needs a second parameter counting the segments we have already used up.

Problem: Sequence Comparison (String Editing)

Given: two strings $A = a_1 \dots a_n$ and $B = b_1 \dots b_m$, where the a_i, b_j are characters from a fixed, finite alphabet.

Goal: Transform A into B by a minimum number of edit steps. An edit step is to insert or delete a character, or to replace a character with another one.

The *edit distance* of A and B is the minimum number of necessary edit steps. The problem can be reformulated as follows. We define a *gap* symbol that does not already appear in the alphabet. An *alignment* of A and B is a pair of strings A' and B' of equal length, obtained from A and B by inserting gaps before, after or between the symbols. A *mismatch* in an alignment is a pair of different symbols (real symbols or gaps) at the same position in A' and B'. Then, our problem is equivalent to computing an alignment of A and B with a minimum number of mismatches.

Generalized versions of the problem assign costs to the different edit steps. The costs may even depend on the characters.

Motivations:

- *Searching and information retrieval:* Finding approximate occurrences of keywords in texts. Keywords are aligned to substrings of the text. Mismatches can stem from misspellings or from grammatical forms of words.
- *Archiving:* If several, slightly different versions of the same document exist, and all of them shall be stored, it would be a waste of space to store the complete documents as they are. It suffices to store one master copy, and the differences of all versions compared to this master copy. The deviations of any document from the master copy are described in a compact way by a minimum sequence of edit steps.
- *Molecular biology:* Comparison of DNA or protein sequences, searching for variants, computing evolutionary distances, etc.

Appendix: Subset Sum Calculation Example

Suppose that you pay small amounts of money by cash. In your pocket you find some coins with (in this ordering) denominations $5, 2, 2, 1, 2$ in the local currency. With these coins you can pay every integer amount up to 12. The important thing is not this small example as such, but the way the Subset Sum algorithm solves it. Using the example, check your understanding of the proposed dynamic programming algorithm for Subset Sum. Do you see how the table was obtained, and what all the entries mean? Can you explain how the algorithm (not you!) would find solutions, for instance, how it figures out that 7 units can be paid by $5 + 2$ or alternatively by $2 + 2 + 1 + 2$?

-	0	1	2	3	4	5	6	7	8	9	10	11	12
-	1	0	0	0	0	0	0	0	0	0	0	0	0
5	1	0	0	0	0	1	0	0	0	0	0	0	0
2	1	0	1	0	0	1	0	1	0	0	0	0	0
2	1	0	1	0	1	1	0	1	0	1	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	0	0
2	1	1	1	1	1	1	1	1	1	1	1	1	1

Algorithms. Lecture Notes 5

Dynamic Programming for Sequence Comparison

The "linear" structure of the Sequence Comparison problem immediately suggests a dynamic programming approach. Naturally, we choose as subinstances the pairs of prefixes of A and B, and we try to align them with a minimum number of mismatches. Accordingly, we define $OPT(i, j)$ to be the minimum number of edit steps that transform $a_1 \dots a_i$ into $b_1 \dots b_j$. What we want is $OPT(n, m)$.

For this problem, already the construction of a recursive formula for $OPT(i, j)$ requires a bit more careful thinking and problem analysis. If we are smart, we observe that one of these three cases must appear in any alignment:

(1) a_n and b_m are aligned. – It follows that $a_1 \dots a_{n-1}$ and $b_1 \dots b_{m-1}$ are aligned somehow.

(2) a_n is not aligned to any symbol of B but comes later than b_m. – It follows that $a_1 \dots a_{n-1}$ and $b_1 \dots b_m$ are aligned somehow.

(3) Symmetrically, b_m is not aligned to any symbol of A but comes later than a_n. – It follows that $a_1 \dots a_n$ and $b_1 \dots b_{m-1}$ are aligned somehow.

As an additional comment (that you may skip), it is "easy to be too complicated" here: In case (2), some more symbols of A may come later than b_m in the alignment, say a_j is aligned to b_m, and $a_{j+1}, \dots, a_n$ come after B and have gap symbols as alignment partners. Now we may want to specify this position j as well. But this would lead to multi-way choices (one subcase for each j), to an unncessarily complicated dynamic programming formula, and a slow algorithm. Instead, it is wise not to guess the position j and to leave this decision open. This might appear paradoxical at first glance.

The above distinction of only three cases will make the formula rather simple. But first we need an auxiliary function, because the edit distance depends on whether the aligned symbols are equal or different. We define $\delta_{ij} = 1$ if $a_i \neq b_j$, and $\delta_{ij} = 0$ if $a_i = b_j$. Now, our case distinction, applied to the prefixes until positions i, j (rather than to n, m), immediately yields:

$$OPT(i,j) = \min\{OPT(i-1,j-1)+\delta_{ij}, OPT(i-1,j)+1, OPT(i,j-1)+1\}.$$

In case (1) we have aligned a_i and b_j, hence we need an edit step (replacement) if and only if these characters are different. The $+1$ term in cases (2) and (3) comes from deletion of a_i and insertion of b_j, respectively. Initialization is done by $OPT(i,0) = i$ and $OPT(0,j) = j$. As usual, the time complexity is the array size, here $O(nm)$, and an optimal edit sequence can be recoverded from the stored edit distances $OPT(i,j)$ by backtracing: Starting from $OPT(n,m)$ we review which case gave the minimum, and we construct the alignment of A and B from behind.

Problem: Searching

Given: a set S of n elements, and another element x.

Goal: Find x in S, or report that x is not in S.

Motivations:
Searching is, of course, a fundamental problem, appearing in database operations or inside other algorithms. Often, S is a set of numbers sorted in increasing order, or a set of strings sorted lexicographically, or any set of elements with an order relation defined on it.

Searching, Sorting, and Divide-and-Conquer

Both the greedy approach and dynamic programming extend solutions from smaller to larger sub-instances *incrementally*. The third main design principle still follows the pattern of reducing a given problem instance to smaller instances, but the instance sizes jump, rather than growing incrementally. The two main actions are:

Divide: A problem instance is split into a few significantly smaller (and usually disjoint) instances. These are solved independently.
Conquer: The obtained partial solutions are combined appropriately to obtain a solution to the entire instance.

Sub-instances are solved in the same way, thus we end up in a **recursive** algorithm. As opposed to dynamic programming where recursion is avoided, divide-and-conquer algorithms are truly recursive,

A certain difficulty is the time analysis. While we can determine the time complexity of greedy and dynamic programming algorithms essentially by counting the operations in loops and adding these numbers, the matter is more tricky for divide-and-conquer algorithms, due to their recursive structure. We will need a special technique for the time analysis: solving **recurrences**. Luckily, a special type of recurrences covers most of the standard divide-and-conquer algorithms. We will solve these recurrences once and for all, and then we can just apply the results. This way, no difficult mathematical analysis will be needed for every new algorithm.

Among the elementary algorithm design techniques, dynamic programming is perhaps the most useful and versatile one. Divide-and-conquer has, in general, much fewer applications, but it is of central importance for searching and sorting problems.

Divide-and-Conquer. First Example: Binary Search

As an introductory example for divide-and-conquer we discuss the simplest algorithm of this type. Consider the Searching problem. Finding a desired element x in a set S of n elements requires $O(n)$ time if S is unstructured. This is optimal, because in general nothing hints to the location of x, thus we have to read the whole of S. But order helps searching. Assume the following: (i) A total order relation is defined in the "universe" the elements of S are taken from, (ii) for any two elements we can decide by a comparison, in $O(1)$ time, which element is smaller, and (iii) S is already sorted in increasing order. In this case we can solve the Searching problem quickly. (How do you look up a word in an old-fashioned dictionary, that is, in a book where words are sorted lexicographically?)

A fast strategy is: Compare x to the central element c of S. (If $|S|$ is even, take one of the two central elements.) If $x < c$ then x must be in the left half of S. If $x > c$ then x must be in the right half of S. Then continue recursively until a few elements remain, where we can search for x directly.

We skip some tedious implementation details, but one point must be highlighted: We assume that the elements of S are stored in an array. This is crucial. In an array we can compute the index of the central element of the current sub-array: If its left and right end is at position i and j, respectively, then the central element is at position $(i+j)/2$, rounded to the next integer. This calculation would not be possible in a linked list.

Every comparison reduces our "search space" by a factor 2, hence we are done after $O(\log n)$ time. Remarkably, the time complexity is far below $O(n)$. We do not have to read the whole input for solving this problem, however, this works only if we can trust the promise that S is sorted. The above algorithm is called the **halving strategy** or **binary search** or **bisection search**.

Binary search is a particularly simple example of a divide-and-conquer algorithm. We have to solve only one of the two sub-instances, and the conquer step just returns the solution from this half, i.e., the position of x or the information that x is not present.

Although it was very easy to see the time bound $O(\log n)$ directly, we also show how this algorithm would be analyzed in the general context of divide-and-conquer algorithms. (Recall that binary search serves here only as an introductory example.) Let us pretend that, in the beginning, we have no clue what the time complexity could be. Then we may define a function $T(n)$ as the time complexity of our algorithm, and try to figure out this function. What do we know about T from the algorithm? We started from an instance of size n. Then we identified one instance of half size, after $O(1)$ operations (computing the index of the central element, and one comparison). Hence our T fulfills this recurrence: $T(n) = T(n/2) + O(1)$. Verify that $T(n) = O(\log n)$ is in fact a solution. We will show later in more generality how to solve such recurrences.

Problem: Skyline

Given: n rectangles, having their bottom lines on a fixed horizontal line.

Goal: Output the area covered by all these rectangles (in other words: their union), or just its upper contour.

Motivations:

This is a very basic example of problems appearing in computer graphics. The rectangles are front views of skyscrapers, seen from a distance. They may partially hide each other, because they stand in different streets. We want to describe the skyline.

Such basic graphics computations should be made as fast as possible, as they may be called many times as part of a larger graphics programme, of an animation, etc.

Solving The Skyline Problem

A more substantial example is the Skyline Problem. Since this problem is formulated in a geometric language, we first have to think about the representation of geometric data in the computer, before we can discuss any algorithmic issues. In which form should the input data be given, and how shall we describe the output?

Each of our rectangles can be specified by three real numbers: coordinate of left and right end, and height. It is natural to represent the skyline as a list of heights, ordered from left to right, and indicating the coordinates where the heights change.

A straightforward algorithm would start with a single rectangle, insert the other rectangles one by one into the picture and update the skyline. Since the jth rectangle may obscure $O(j)$ lines in the skyline formed by the first $j-1$ rectangles, updating the list needs $O(j)$ time in the worst case. This results in $O(n^2)$ time in total.

The weakness of this obvious algorithm is that is uses linearly many update operations to insert only one new rectangle. This is quite wasteful. The key observation towards a faster algorithm is that merging two arbitrary skylines is no more expensive than inserting a single new rectangle (in the worst case). This suggests a divide-and-conquer approach: Divide the instance arbitrarily in two sets of roughly $n/2$ rectangles. Compute the skylines for both subsets independently. Finally combine the two skylines, by scanning them from left to right and always keeping the higher horizontal line. The details of this conquer phase are straightforward, we skip them here. The conquer phase runs in $O(n)$ time. Hence the time complexity of the entire algorithm satisfies the recurrence $T(n) = 2T(n/2) + O(n)$. For the moment, believe that this recurrence has the solution $T(n) = O(n \log n)$. (We may prove this by an ad-hoc argument, but soon we will do it more systematically.) This is significantly faster than $O(n^2)$.

Again, the intuitive reason for the much better time complexity is that we made a better use of the same number $O(n)$ of update operations. We can also see this from the recurrence for the bad algorithm, which becomes $T(n) = T(n-1) + O(n)$, with solution $T(n) = O(n^2)$.

Appendix

It is recommended to do some calculation examples for Sequence Comparison, in order to see how simple the process is. Below is one, but you may consider some other (more or less funny) pairs of similar words. Do you fully understand how the entries are produced? Why is there a blank symbol in front of both words (in the second row and column)? And how do you get the optimal alignments from the table by backtracing? In this example you should find two slightly different optimal alignments with 4 mismatches:

APHORIS—M
ALGORITHM

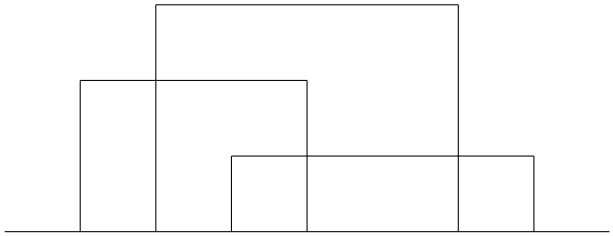
APHORI—SM
ALGORITHM

-	-	A	P	H	O	R	I	S	M
-	0	1	2	3	4	5	6	7	8
A	1	0	1	2	3	4	5	6	7
L	2	1	1	2	3	4	5	6	7
G	3	2	2	2	3	4	5	6	7
O	4	3	3	3	2	3	4	5	6
R	5	4	4	4	3	2	3	4	5
I	6	5	5	5	4	3	2	3	4
T	7	6	6	6	5	4	3	3	4
H	8	7	7	6	6	5	4	4	4
M	9	8	7	7	7	6	5	5	4

Other remarks:

- You might also want to reflect upon the paradoxon mentioned in the derivation of the Sequence Comparison algorithm: How can it be more efficient to defer a decision, and how does the final algorithm take care of that?

- For simplicity we would work with two nested for-loops for the indices i and j, and fill the table completely, row by row. But in the example one can see that the small values appear around the main diagonal, and the backtracing path will not reach the large values. Maybe it is not necessary to compute the entire table? In fact, one can rearrange the computations: first fill in all 0s, then all 1s, then all 2s, and so on, until the right lower corner is reached. This is enough to recover an optimal alignment. This saves time (when the strings are similar),



but it also makes the code more complicated. – Think further in this direction if you like, but this is a bit more advanced (and not expected here).

- Note that the Skyline problem is not in the textbook. The picture shows a very small instance, just for an easier understanding of the problem.
- For those who are already familiar with sorting algorithms: You might have recognized that the proposed Skyline algorithm is just a camouflaged Mergesort. But the point here was to demonstrate the power of divide-and-conquer for certain problems, compared to incremental approaches.
- The other "picture" shows a so-called *proof without words* for the claim that $T(n) = O(n \log n)$ is actually the solution to the recurrence $T(n) = 2T(n/2) + O(n)$. Can you make sense of this rebus? If not: Do not worry. We will also see a serious mathematical proof.

```
o - o - o - o - o - o - o - o
o - o - o - o I o - o - o - o
o - o I o - o I o - o I o - o
o I o I o I o I o I o I o I o
```

Algorithms. Lecture Notes 6

Solving a Special Type of Recurrences

It is time to provide some general tool for the time complexity analysis of divide-and-conquer algorithms. Most of these algorithms divide the given instance of size n in some number a of instances of roughly equal size, say n/b, where a, b are constant integers. (The case $a \neq b$ is quite possible. For example, in binary search we had $b = 2$ but only $a = 1$.) These smaller instances are solved independently and recursively. The conquer phase needs some time, too. It should be bounded by a polynomial, otherwise the whole algorithm cannot be polynomial. Accordingly we assume that the conquer phase needs at most cn^k steps, where c, k are other non-negative constants. We obtain the following type of recurrence:

$$T(n) = aT(n/b) + cn^k.$$

We remark that $a \geq 1$, $b \geq 2$, $c > 0$, and $k \geq 0$. Also assume that $T(1) = c$. This is probably not true for the particular algorithm to be analyzed, but we can raise either $T(1)$ or c to make them equal, which will not affect the O-result but will simplify the calculations.

Since $T(n)$ recurs to $T(n/b)$, it is simpler to restrict attention first to arguments n which are powers of b, say $n = b^m$ for integer m. Starting with $T(b^0) = T(1) = c$ we can explicitly write down the $T(b^m)$ values step by step, simply by applying the recurrence $T(b^m) = aT(b^{m-1}) + cb^{mk}$:

$$T(b^0) = c$$

$$T(b^1) = c(a + b^k)$$

$$T(b^2) = c(a^2 + ab^k + b^{2k})$$

$$T(b^3) = c(a^3 + a^2b^k + ab^{2k} + b^{3k})$$

Maybe these few steps are enough to see the general pattern. The expression for general m is:

$$T(n) = ca^m \sum_{i=0}^{m} (b^k/a)^i$$

Formally this may be proved by induction on m. In this explicit form, $T(n)$ is merely a geometric sum. We will now simplify it, using the O-notation. The ratio b^k/a is decisive for the final result. Three different cases can appear. Remember that $n = b^m$, hence $m = \log_b n$, and recall some laws of logarithms.

If $a > b^k$ then the sum is bounded by a constant, and we simply get

$$T(n) = O(a^m) = O(a^{\log_b n}) = O(n^{\log_b a}).$$

If $a = b^k$ then the sum is $m + 1$, and a few steps of calculation yield

$$T(n) = O(a^m m) = O(n^k \log n).$$

If $a < b^k$ then only the mth term in the sum determines the result in O-notation:

$$T(n) = O(a^m (b^k/a)^m) = O((b^m)^k) = O(n^k).$$

These three formulae are often called the **master theorem** for recurrences.

So far we have only considered very special arguments $n = b^m$. However, the O-results remain valid for general n, for the following reason: The results are polynomially bounded functions. If we multiply the argument by a constant, then the function value is changed by at most a constant factor as well. Every argument n is at most a factor b away from the next larger power b^m. Hence, if we generously bound $T(n)$ by $T(b^m)$, we incur only another constant factor. This demonstrates again the beauty of O-notation.

Most divide-and-conquer algorithms lead to a recurrence settled by the master theorem. In other cases we have to solve other types of recurrences. Approaches are similar, but sometimes the calculations and bounding arguments may be more sophisticated. Therefore it is good that you have seen the template.

Problem: Sorting

Given: a set of n elements, where an order relation is defined, e.g., a set of numbers, equipped with the natural $\leq$ relation. Comparison is assumed to be an elementary operation, that is, any two elements can be compared in $O(1)$ time.

Goal: Output the n given elements in ascending order.

Motivations: obvious

Mergesort

One natural way to solve the Sorting problem is called Bubblesort: The elements are stored in an array, and whenever two neighbored elements are in the wrong order, we swap them. It can be easily shown that Bubblesort needs $O(n^2)$ time, and it does not matter which wrong pairs are swapped first.

Bubblesort is worst if many elements are far from their proper places, because the algorithm moves them only to neighbored places in every step. The Insertion Sort algorithm overcomes this shortage: It uses k rounds to sort the first k elements ($k = 1, \ldots, n$). To insert the $(k+1)$st element we search for the correct position, using binary search. Hence we need $O(n \log n)$ comparisons in all n rounds. This seems to be fine. Unfortunately, this result does not yet imply $O(n \log n)$ time! If we use an array, we may be forced to move $O(k)$ elements in the kth round, giving again an overall time complexity of $O(n^2)$. Using a doubly linked list instead, we can insert an element in $O(1)$ time at a desired position, but now we cannot apply binary search for finding the correct position, because no indices are available. Without further tricks we have to apply linear search, resulting once more in $O(n^2)$ time for all n rounds. One could implement Insertion Sort with a dictionary data structure. This achieves $O(n \log n)$ time, however with an unnecessarily large hidden constant. As opposed to that, the fastest sorting algorithms are genuine divide-and-conquer algorithms.

Mergesort divides the given set arbitrarily in two halves, sorts them separately, and merges the two ordered sequences while preserving the order. This conquer phase runs in $O(n)$ time: We simply scan both ordered sequences simultaneously and always move the currently smallest element to the next position in the output sequence. The time complexity satisfies the recurrence $T(n) = 2T(n/2) + O(n)$, with solution $T(n) = O(n \log n)$.

Mergesort has a particularly simple structure, but it is not the fastest sorting algorithm in practice. The drawback is that it executes too many copy operations besides the comparisons. Moreover, in every merging phase on every recursion level it moves all elements of the merged subsets into a new array. Thus, we also need extra memory, which can be another practical obstacle in the case of large n.

There are several alternative algorithms for sorting which also need $O(n \log n)$ time, but with different hidden constant factors. These factors are hard to analyze theoretically, but some informal reasoning as above gives at least some hints, and runtime experiments can figure out what is really faster in practice.

Quicksort and Random Splitters

One of the favorable sorting algorithms is **Quicksort**. It needs only one array of size n for everything, apart from a few auxiliary memory cells for rearrangements. An algorithm with these properties is said to work *in place*, or (Latin) *in situ*. Bubblesort is an *in place* algorithm as well, however a slow one.

Quicksort divides the given set according to the following idea. Let p be some fixed element from the set, called a **splitter** or **pivot**. Once we have put all elements $< p$ and $> p$, respectively, in two different subsets, it suffices to sort these two subsets independently. Then, just concatenating the sorted sequences, with the splitter in between, gives the final result. Thus the conquer phase is trivial here.

What makes Quicksort quick is the implementation details of the divide phase: Two pointers starting at the first and last element scan the array inwards. As long as the left pointer meets only elements $< p$, it keeps on moving to the right. Similarly, as long as the right pointer meets only elements $> p$, it keeps on moving to the left. When both pointers have stopped, we swap the two current elements. This requires only one additional memory cell where one element is temporarily stored. As soon as both pointers meet, the set is divided as desired.

Clearly, the divide phase needs $O(n)$ time. The hidden constant is really small due to the simple procedure. Moreovoer, we only have to move elements being on the wrong side, and these can be much less than n. What about the overall time complexity? If the splitters exactly halved the sets on every recursion level, we would have our standard recurrence $T(n) = 2T(n/2) + O(n)$, with solution $T(n) = O(n \log n)$. Unfortunately,

this is barely the case if we choose our splitters without care. In the worst case, the splitter may even always be the minimum or maximum element of the set, and then $O(n^2)$ time is needed. What can we do about that?

The ideal splitter for Quicksort would be the median, i.e., the element being larger and smaller than exactly half of the elements in S. But how difficult is it to compute the median? We could sort the set and then read off the element with rank $n/2$. But this would be silly, because sorting was the problem we came from.

Actually, Quicksort goes a different way. A splitter is selected at random! (Indeed, an algorithm can make random decisions. This is not against the general demand that an algorithm must be unambiguous and must not allow for intervention. Random decisions are made by a random number generator rather than by the user.) Now the worst case that the splitter is close to the maximum or minimum is very unlikely. The splitters will mostly be in the middle part of the sorted sequence, and then we get reasonably well balanced partitionings in two sets. In fact, a strict analysis confirms that $O(n \log n)$ time is needed on expectation. We do not give this analysis here, as randomized algorithms are beyond the scope of this course.

The speed in practice is further improved by chosing three random elements and taking their median as the splitter. This needs a few extra operations, but much more operations are saved due to better partitionings.

Problem: Center of a Point Set on the Line

Given: n points $x_1, \ldots, x_n$ on the real line.

Goal: Compute a point x so that the sum of distances to all given points $\sum_{i=1}^{n} |x - x_i|$ is minimized.

Motivations:

Imagine a village consisting of only one long street with n houses, with irregular spaces in between. A building for a new shop shall be erected at a "central" position in this street, that is, the average distance to the houses shall be minimized.

The scenario becomes more interesting in a usual "2-dimensional" village. However, for simplicity let us assume that streets go only in north-south and west-east direction, this road network is complete, and houses stand somewhere in these streets. What would now be the ideal position for the shop, if minimizing the average distance to all houses is the only

criterion? (Here, distances are understood as walking or driving distances along the streets, not as Euclidean distances.)

More complicated (and more realistic) facility location problems with various objectives appear in infrastructure planning.

Problem: Selection and Median Finding

Given: a set of n elements, where an order relation is defined, and an integer k.

Goal: Output the element of rank k, that is, the kth smallest element.

If $k = n/2$ (rounded), we call the kth smallest element the *median*. The term "Selection problem" is a bit unspecific but established. The problem is also called Order Statistics.

Motivations:

In statistical investigations, the median is often better suited as a "typical" value than the average, because it is robust against outliers. For example, the average wealth in a population can raise when only a few people become extremely rich. Then the average gives a biased picture of the wealth of the majority. This does not happen if we look at the median. Changing the median requires substantial changes of the wealth of many people.

More generally speaking, median values, or values with another fixed rank, are often used as thresholds for the discretization of numerical data, because the sets of values above/below these thresholds have known sizes.

We remark that, once we know the *kth smallest element*, we can also find the k *smallest elements* in another $O(n)$ steps, just by $n-1$ comparisons to the rank-k element.

Algorithms for Selection and Median Finding

First we remark that the solution to the "Center of a Point Set on the Line" problem is the median of the given coordinates, and not the average. (Why? Think about it.)

Surprisingly, the Selection problem can be solved without sorting, in $O(n)$ time. The intuitive reason is that Selection needs much less information than Sorting. Here is a fast algorithm. Again we choose a random splitter p and compare all elements to p in $O(n)$ time. Now we know the

rank r of p. If $r > k$ then we throw out p and all elements larger than p. If $r < k$ then we throw out p and all elements smaller than p, and set $k := k - r$. If $r = k$ then we return p. This procedure is repeated recursively.

If the splitters were always in the middle, the time would follow the recursion $T(n) = T(n/2) + O(n)$, with solution $T(n) = O(n)$. Again, a probabilistic analysis confirms an expected time $O(n)$, whereas the worst case is $O(n^2)$. There also exists a deterministic divide-and-conquer algorithm for Selection, but it is non-standard and a bit complicated. More importantly, its hidden constant in $O(n)$ is rather large, making the algorithm more of academic interest, while the random-splitter algorithm is practical.

Information Flow and Optimal Time Bounds

We conclude the discussion of Sorting and Selection with some remarks on *optimal* time bounds. One of our general goals is to make algorithms as fast as possible. So, how good are our time bounds?

In order to find a specific element in an ordered set we needed $\log_2 n$ comparisons of elements, by doing binary search. No other algorithm using only comparisons as elementary operations can have a better worst-case bound. This holds due to an **information-theoretic** argument explained as follows. How much information do we gain from our elementary operations? Every comparison gives a binary answer ("smaller" or "larger"), thus it splits the set of possible results in two subsets for which either of the answers is true. In the worst case we always get the answer which is true for the larger subset, and this reduces the number of candidate solutions by a factor at most 2. Since there were n possible solutions in the beginning, *every* algorithm needs at least $\log_2 n$ comparisons in the worst case.

The same type of argument shows that no sorting algorithm can succeed with less than $O(n \log n)$ comparisons in the worst case: Since a set with n elements can be ordered in $n!$ possible ways, but only one of them is the correct order, any sorting algorithm can be forced, by a nasty instance, to use $\log_2 n!$ comparisons. Some calculation shows that this is $n \log n$, subject to a constant factor. As we ignore such constants anyway, the proof is a one-liner:

$$\log_2 n! = \sum_{k=1}^{n} \log_2 k \geq (n/2) \log_2 (n/2).$$

This argument does not apply to the Selection problem: There we have only n possible results, and $\log n$ is a very poor lower bound. In fact, $O(n)$ is the optimal bound, but for a totally different reason: We have to read all elements, since every change in the instance can also change the result.

It should also be noticed that the information-theoretic lower bounds for Sorting and Searching hold only under the assumptions that (1) nothing is known in advance about the elements, and (2) doing pairwise comparisons is the only way to gather information. Faster algorithms can exist for special cases where we know more about the instance. For example, Bucketsort works in $O(m + n)$ time, if the n elements come from a fixed range of m different numbers. Similarly, a set of words over a fixed alphabet can be sorted in lexicographic order in $O(n)$ time, where n is the total length of the given words. These results do not contradict each other.

Problem: Counting Inversions

Given: a sequence $(a_1, \ldots, a_n)$ of elements where an order relation $<$ is defined.

Goal: Count the inversions in this sequence. An inversion is a pair of elements where $i < j$ but $a_i > a_j$.

Motivations:

This problem is obviously related to sorting, but here the goal is only to measure either the "degree of unsortedness" of a given sequence, or the dissimilarity of two sequences containing the same elements but in different order. One of them can be assumed to be $(1, 2, 3, \ldots, n)$, and the other sequence is a permutation of it. One natural measure of unsortedness or dissimilarity, among several others, is the number of inversions.

The problem appears, e.g., in the comparison of rankings (e.g., of web pages returned by search engines), and in bioinformatics (measuring the dissimilarity of two rearranged genome sequences).

Algorithms. Lecture Notes 7

An Algorithm for Counting Inversions

Next we want to count the number of inversions in a sequence, faster than by the obvious $O(n^2)$ time algorithm. This problem example is instructive as it combines divide-and-conquer with some general issue in algorithm design (see below).

Due to the vague similarity to Sorting, it should be possible to apply divide-and-conquer. We could split the sequence in two halves, say, $A = (a_1, \ldots, a_m)$ and $B = (a_{m+1}, \ldots, a_n)$, where $m \approx n/2$, and count the inversions in A and B separately and recursively. In the conquer phase we would count the inversions between A and B, that means, those involving one element in each of A and B, and sum up. But it is not easy to see how to execute this conquer phase better than in $O(n^2)$ time. At this point we need a creative idea.

Intuitively, it would be much easier to do the conquer phase when the two halves were sorted. What if we also *sort* the sequence while counting the inversions? This idea may appear counterintuitive: Sorting is not what we originally wanted, and one might think that a problem becomes only harder by extra demands. But in fact, sorting serves here as a tool to make the conquer phase of another algorithm efficient! Figuratively speaking, our inversion counting algorithm will be piggybacked by a recursive sorting algorithm. That is, we extend our problem to Sorting *and* Counting Inversions, and solve it recursively.

As the underlying sorting algorithm we take the conceptually simple Mergesort. If we manage to merge two sorted sequences A and B, and simultaneously count the inversions between A and B, still everything in $O(n)$ time, then the recurrence $T(n) = 2T(n/2) + O(n)$ will apply. Remember that its solution is $T(n) = O(n \log n)$.

In fact, this $O(n)$ time merging-and-counting is easily done, using some pointers and counters: We proceed as in Mergesort, and whenever the next

element copied into the merged sequence is from B, this element has inversions with exactly those elements of A which are not visited yet. Hence we only need $O(n)$ additions of integers, on top of the copy operations.

Faster Multiplication

This is one of the most amazing classic results in the field of efficient algorithms. Recall that the "school algorithm" for multiplication of two integers, each with n digits, needs $O(n^2)$ time. For simplicity let n be a power of 2, otherwise we may fill up the decimal representations of the factors with dummy 0s. This "padding" can at most double the input size, hence the (polynomial) time complexity is increased by some constant factor only.

An attempt to multiply through divide-and-conquer is to split the decimal representations of both factors in two halves, and then to multiply them with help of the distributive law:

$$(10^{n/2}w + x)(10^{n/2}y + z) = 10^n wy + 10^{n/2}(wz + xy) + xz$$

.

That is, we reduce the multiplication of n-digit numbers to several multiplications of $n/2$-digit numbers and some additions. Then we apply the same equation recursively to all the $n/2$-digit numbers. This algorithm satisfies the recurrence $T(n) = 4T(n/2) + O(n)$, since additions and other auxiliary operations cost only $O(n)$ time. Factor 4 comes from the four recursive calls. Note that only w, x, y, z are multiplied recursively, whereas multiplications with powers of 10 are trivial: Append the required number of 0s. Since $2^1 < 4$, the master theorem yields $T(n) = O(n^{\log_2 4}) = O(n^2)$. ... Too bad! Unfortunately, this is not an improvement.

Was this a futile approach? No, we have just failed to fully exploit the power of the idea. The key observation suggesting that the usual algorithm might be unnecessary slow was that it executes the same multiplications of digits many times. Simple geometry gives an idea how to save one of the four recursive multiplications: Consider a rectangle with side lengths $w + x$ and $y + z$. We need the area sizes of three parts of this rectangle: $xz, wy, wz + xy$. The last term is not the area of a rectangle, but looking at the the whole rectangle we see that

$$(w + x)(y + z) = wy + (wz + xy) + xz$$

.

Hence we obtain the desired numbers by only three multiplications: $(w+x)(y+z)$, wy, and xz. The term $wz+xy$ is obtained by subtractions, which are cheaper than another multiplication. Altogether we need only $T(n) = 3T(n/2) + O(n)$ time, which yields $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$. This is indeed considerably better than $O(n^2)$.

A minor remark is that this analysis was not completely accurate: The factors $w+x$ and $y+z$ can have $n/2+1$ digits. But then we can split off the first digit, which gives us recursive calls to instances with (now accurately) $n/2$ digits, plus some more $O(n)$ terms in the recurrence which do not affect the time bound in O-notation. Hence this minor technicality can be easily fixed.

But now, why don't we use this clever algorithm in everyday applications? It must be confessed that the acceleration takes effect only for rather large n (more than some 100 digits). The main reason is the administrative overhead for the recursive calls. The simple traditional algorithm does not suffer from such overhead. Multiplication by divide-and-conquer is not suitable for numerical calculations, since the factors have barely more than a handful digits. Still the algorithm is not useless. Some cryptographic methods rely on the fact that integers are easy to multiply but hard to split into their prime factors. These methods use multiplications of large numbers. They have no numerical meaning but encode messages and secret keys instead. In such applications, n is large enough to make the asymptotically fast algorithm really fast also in practice.

The above algorithm is not yet the fastest known multiplication algorithm. An $O(n \log n \log\log n)$ time algorithm is based on convolution via Fast Fourier Transformation, but this is beyound the scope of this course. It is not known whether one can multiply even faster.

Finally we mention that similar divide-and-conquer algorithms exist also for matrix multiplication, with similar provisoes. Very large matrices can appear in calculations and simulations in mechanics or economy.

Problem: Closest Points

Given: a set of n points in the plane, specified by their Cartesian coordinates (x_i, y_i).

Goal: Find a pair of points with minimum Euclidean distance (i.e., the usual distance in the plane, which is the length of the straight line segment connecting the points).

Motivations:

Some approaches to hierarchical clustering of data take the two closest data points and combine them to a cluster by replacing these two points by their midpoint, and this step is repeated until one cluster remains.

Divide-and-Conquer in Geometry: Closest Points

Fast geometric calculations are needed in computer graphics, computer-aided design, robotics, planning (transport optimization, facility location), chemistry (modelling molecules and their dynamics), for extracting information from geographic databases, etc. The amount of data can be huge (e.g., elements of a picture), such that efficient algorithms make a difference.

Divide-and-conquer is suitable for various geometric problems, because instances can be divided in a natural way. However, the conquer phase is usually less trivial. To give at least an impression, we discuss another geometric problem example: finding a pair of closest points among n given points in the plane.

An obvious algorithm would compute all pairwise distances and determine the minimum in $O(n^2)$ time. Instead, we aim at a divide-and-conquer algorithm satisfying the recurrence $T(n) = 2T(n/2)+O(n)$, hence with time complexity $T(n) = O(n \log n)$.

It is natural to divide the set by a straight line. To make the calculation details simple, we first sort the points by their x-coordinates, and then halve the set by a vertical separator line. More formally, we take the median z of all x-values and put all points with coordinate $x < z$ and $x > z$, respectively, in the two sets. Sorting takes $O(n \log n)$ time and needs to be done only once in the beginning, which does not destroy the desired time bound.

Then, of course, we compute the closest pairs in both subsets recursively. Let d be the minimum of the two minimum distances. The more tricky part is to combine the partial solutions. The global solution could be the best of the two closest pairs from the two subsets, but there could also exist a pair of points with distance smaller than d, having one point on each side of the separator line. But now some geometry helps:

The candidates for such pairs of points are in a stripe of breadth d around the separator line. Moreover, each point has only constantly many partners (at distance smaller than d) on the other side, hence only $O(n)$ such pairs of close points must be considered. These pairs can be identified in $O(n)$ time, if all points are already sorted by their y-coordinates as well. With careful

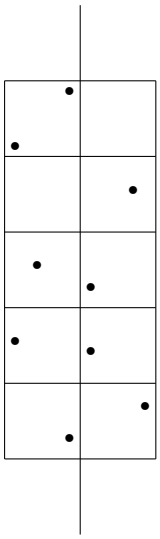


Figure 1: This is why the conquer phase needs only $O(n)$ time. Pairs with points on both sides of the separating line can only be taken from a stripe of breadth $2d$. For clarity, let us partition this stripe into squares with side length d. Since the points on each side must keep a distance at least d (yes, the points must keep some distance, too, not only people ...), there can be only one or two points in every square. Moreover, we need to measure the distances of points only in incident squares, since other points are clearly too far away. These are $O(1)$ candidate partners for each point. Once we have sorted the y-coordinates in the beginning (not during the recursion!), we only have to traverse some sorted list of points.

implementation details (that we omit here), all steps in the conquer phase run in $O(n)$ time as desired.

Algorithms. Lecture Notes 8

Reductions between Problems

In general, any new problem requires a new algorithm. But often we can solve a problem X using a known algorithm for a related problem Y. That is, we can **reduce** X to Y in the following informal sense: A given instance x of X is translated to a suitable instance y of Y, such that we can use the available algorithm for Y. Eventually the result of this computation on y is translated back, such that we get the desired result for x. Does this sound too abstract? Here we illustrate the idea by an example:

Suppose that we want an algorithm for multiplying two integers, and there is already an efficient algorithm available that can compute the square of one integer. It needs $S(n)$ time for an integer of n digits. Can we use it somehow to multiply arbitrary integers a, b efficiently, without developing a multiplication algorithm? These assumptions are a bit made up, but we will see below that the example is meaningful. We do not have to specify the function S, but it is clear that $S(n) \geq n$. (Why?)

Certainly, squaring and multiplication are closely related problems. In fact, we can use the identity $ab = ((a + b)^2 - (a - b)^2)/4$. We only have to add and subtract the factors in $O(n)$ time, apply our squaring algorithm in $S(n)$ time, and divide the result by 4, which can be easily done in $O(n)$ time, since the divisor is a constant.

Thus we have reduced some problem X (multiplication) to some problem Y (squaring). Namely, we have taken an instance of X (the factors a and b), transformed it quickly into two instances of Y (with operand $a+b$ and $a-b$, respectively), solved these instances of Y by the given squaring algorithm, and finally applied another fast computation to the results (an addition, and a division by 4) to obtain the solution ab to the instance of problem X.

It is crucial that not only a fast algorithm for Y is available, but the transformations are fast as well. Note that the total time for our multiplication algorithm is $O(S(n))$. The $O(n)$-time transformations are already

counted in this time bound.

Doing multiplication through an algorithm specialized to squaring may appear somewhat strange. But we get an interesting insight from this reduction: One might conjecture that squares can be computed faster than products of arbitrary numbers, since this problem is only a very special case of multiplication. In fact, in applications with a lot of squarings (simulations of physical systems?) it would be nice to have such a faster algorithm. But due to our reduction, these hopes come to nothing, and we can firmly give a negative answer: Any faster algorithm for squaring would immediately yield a faster algorithm for (general) multiplication as well. We conclude that squaring is not easier than multiplication!

We have identified two different purposes of reductions: (1) solving a problem X with help of an already existing algorithm for a different problem Y, and (2) showing that a problem Y is at least as difficult as another problem X.

Note that (1) is of immediate practical value, and even usual business: Ready-to-use implementations of standard algorithms exist in software packages and algorithm libraries. One can just apply them as black boxes, by using their interfaces, and without caring about their internal details. Formally this is nothing but a reduction! Point (2) might appear less practical, but it gives us a way to compare the difficulty of problems without determining their "absolute" time complexities (which is often impossible to figure out). It can be useful to know such comparisons, If Y is at least as difficult as X, then research on improved algorithms should first concentrate on the easier problem X. Some applications (as in cryptography) even rely on the hardness of certain computational problems, rather than efficient solvability.

Reductions – Now More Formally

After this informal introduction we approach the abstract definitions of reductions that are needed to build up a **complexity theory** of computational problems.

Let X, Y be any two problems. By $|x|$ we denote the length of an instance x of problem X. We say that X *is reducible to* Y *in* $t(n)$ *time*, if we can do the following in $t(n)$ time for any given instance x with $|x| = n$: Transform x into an instance $y = f(x)$ of problem Y, and transform the solution of y back into a solution of x.

Symbol f merely denotes the function describing how an instance is

transformed. It must be computable in $t(n)$ time. Note that the time needed by the algorithm for problem Y is not counted in $t(n)$. Only the transformations of instances and solutions are charged, because these are the extra costs for using the algorithm for Y, so to speak. According to the very idea of reductions, the solution algorithm for Y is not part of the reduction. In other words, a reduction is merely an "affair bwtween two problems".

Assuming that we have an algorithm for problem Y with time bound $u(n)$, we can now solve an instance x of problem X in time $t(|x|)+u(|f(x)|)$. Since $|f(x)| \leq t(n)$ (why?), this is bounded by $t(n) + u(t(n))$.

Loosely speaking we can conclude: If Y is an easy problem and the reduction is fast, then X is an easy problem, too. Conversely, if X is a hard problem, and we have a fast reduction to problem Y, then Y is a hard problem, too. In this sense, a reduction allows a comparison of the difficulty of two problems.

These comparisons become much easier to handle formally when we restrict attention to so-called **decision problems**. A decision probem is simply a computational problem that has to output a Yes or No answer (e.g., the instance has a solution or not). This is not a severe restriction. Every optimization problem can be viewed as a series of decision problems, as follows. Instead of asking "give me a solution where the objective value is minimized" we can ask "does there exist a solution with objective value at most t?", for various thresholds t. Informally, if the optimization problem is easy to solve, then the corresponding decision problem is also easy, for every threshold t. (We just compare an optimal solution to the threshold.) By contraposition, if already the decision problem is hard, then the corresponding optimization problem is also hard.

Finally we define reductions between decision problems X and Y: We say that X is reducible to Y in $t(n)$ time, if we can compute in $t(n)$ time, for any given x with $|x| = n$, an instance $y = f(x)$ of Y such that the answer to x is Yes *if and only if* the answer to y is Yes. (In other words, instances x, y of the decision problems X, Y are equivalent.) If the time $t(n)$ needed for the reduction is bounded by a polynomial in n, we say that X is **polynomial-time reducible** to Y.

Problem: Clique

A **clique** in a graph $G = (V, E)$ is a subset $K \subseteq V$ of nodes such that all possible edges in K exist, i.e., there is an edge between any two nodes in K.

Given: an undirected graph G.

Goal: Find a clique of maximum size in G.

Motivations:

This is a fundamental optimization problem in graphs. Many other problems can be rephrased as a Clique problem. A setting where it appears directly is the following: The graph models an interaction network (persons in a social network, proteins in a living cell, etc.), where an edge means some close relation between two "nodes". We may wish to identify big groups of pairwise interacting "nodes", because such groups may have an important role in the network.

Problem: Independent Set

An **independent set** in a graph $G = (V, E)$ is a subset $I \subseteq V$ of nodes such that no edges in I exist.

Given: an undirected graph G.

Goal: Find an independent set of maximum size in G.

Motivations:

The same general remarks as for the Clique problem apply. A setting where it appears directly is the following: The graph models conflicts between items, and we wish to select as many as possible items conflict-free. For example: Goods shall be packed in a box, but for security reasons certain goods must not be packed together. How many items can we put in the same box?

Problem: Vertex Cover

A **vertex cover** in a graph $G = (V, E)$ is a subset $C \subseteq V$ of nodes such that every edge of G has at least one of its two nodes in C.

Given: an undirected graph G.

Goal: Find a vertex cover of minimum size in G.

Motivations:

Vertex covers are of interest in "facility location" problems. A toy example is the question: How can we place a minimum number of guards in a museum building so that they can watch all corridors?

Another application field is combinatorial inference. As a bioinformatics example, consider some genetic disease that appears if some rare bad variant of a certain gene is present. Geneticists want to figure out what the bad gene variants are. Their number is expected to be small, as a result of a few unfortunate mutations. Every person carries two copies of the gene. Given the genetic data of a group of persons having the disease, we know that each person has at least one bad variant in his/her pair of genes. Now we can try and explain the data by a minimum number of different bad gene variants.

Reductions Between Some Graph Problems

We illustrate the definition of polynomial-time reducibility by some simple reductions between the mentioned graph problems, reformulated as decision problems. Let $G = (V, E)$ be an undirected graph. The Clique problem takes as input a graph G and an integer k and asks whether G contains a clique of at least k nodes. The Independent Set problem takes as input a graph G and an integer k and asks whether G contains an independent set of at least k nodes.

It is rather obvious that Clique and Independent Set are only different formulations of the same problem. To make this precise, we show that Clique and Independent Set are polynomial-time reducible to each other. A reduction function is established by $f(G, k) := (\bar{G}, k)$, where $\bar{G}$ is the complement graph of G, that is, the graph obtained by replacing all edges with non-edges and vice versa. Regarding the formalities, note that f has to transform an instance of a problem into an instance of the other problem, and an instance consists here of a graph G *and* an integer k. The transformation is obviously manageable in polynomial time.

The Vertex Cover problem takes as input a graph G and an integer k and asks whether G contains a vertex cover of at most k nodes. We show that Independent Set and Vertex Cover are polynomial-time reducible to each other. The key observation is that $C \subseteq V$ is a vertex cover if and only if

$V \setminus C$ is an independent set. In other words, vertex covers and independent sets are complement sets in the same graph. Hence, a vertex cover of size at most k exists if and only if an independent set of size at least $n - k$ exists (and similarly in the other direction). This gives us a possible reduction: $f(G, k) := (G, n - k)$. This time we did not change the graph. The only work of the reduction function is to replace the threshold k with $n - k$.

These very simple reductions show that all three problems are essentially the same. In particular, they are equally hard.

If a problem X is merely a special case of problem Y, we immediately have a polynomial-time reduction from X to Y. To give an example: Interval Scheduling is a special case of Independent Set, which is seen as follows. Given a set of intervals, we construct a graph with the given intervals as nodes, where two nodes are adjacent whenever the represented intervals intersect. We call it the **interval graph** of the given set of intervals. The decision version of Interval Scheduling is: Given a set of intervals and an integer k, does there exist a subset of at least k pairwise disjoint intervals? Now it should be clear that the above graph construction is, in fact, a polynomial-time reduction from Interval Scheduling to Independent Set. The function f describing this reduction transforms the set of intervals into its interval graph, while k remains unchanged.

Note that we cannot reduce Independent Set to Interval Scheduling in the same way. The catch is: Starting from an arbitrary graph G, we cannot always find a set of intervals whose interval graph is exactly G. This is because "most" graphs are not interval graphs. But maybe there exists some non-obvious and tricky reduction nevertheless? Later we will be able to rule out this possibility, too.

Complexity Classes and Hardness

Comparisons by polynomial-time reducibility define a partial ordering on the class of decision problems, with respect to their complexities: This relation is **transitive**, that is, if X is polynomial-time reducible to Y, and Y is polynomial-time reducible to Z, then X is polynomial-time reducible to Z. This is not surprising and almost obvious, but we must be a little bit careful with the time bounds.

To prove transitivity, let f and g be the functions transforming the instances from X to Y and from Y to Z, respectively. Let f and g be computable in time p and q, respectively, where p and q are polynomials. In

order to solve an instance x of X (of size n) with the help of an algorithm for Z, we can compute instance $g(f(x))$ of Z, and then run the available algorithm. The time needed for the reduction is $p(n) + q(p(n))$. Note that we can bound the input length $|f(x)|$ in the second term only by $p(n)$, since the transformation algorithm that computes $f(x)$ can use $p(n)$ time, and it may use this time to generate such a long instance. However, since p, q are polynomials, $q(p(n))$ is still a polynomial in n, hence the entire reduction from X to Z is polynomial. Also note that the instances x and $g(f(x))$ are in fact equivalent.

The "bottom" of the mentioned partial ordering of problems compared by their complexities is the class of "easy" problems. We pointed out earlier that efficient algorithms should need polynomial time. Accordingly, we define the **complexity class** $\mathcal{P}$ to be the class of all decision problems that admit an algorithm which solves every instance x correctly and in $O(p(n))$ time, where p is some polynomial, and n denotes the size of x. Note that p may depend on the problem, but not on n.

If a given problem X is quickly reducible to an easy problem, then problem X is easy, too. Formally, if a decision problem X is polynomial-time reducible to a decision problem $Y \in \mathcal{P}$, then $X \in \mathcal{P}$.

The proof is similar to the transitivity proof: Let p be the polynomial time bound for computing the function f which reduces X to Y, and let t be the polynomial time bound of an algorithm for problem Y. (Now we have to count in the time used by this target algorithm.) Given an instance x of X, with size $|x| = n$, we compute $f(x)$ and solve instance $f(x)$ by the algorithm for Y. Now the time bound is $p(n) + t(p(n))$, and this is polynomial in n.

Interestingly, the contraposition says: If X is polynomial-time reducible to Y, and X is *not* in $\mathcal{P}$, we can conclude that Y is not in $\mathcal{P}$ either! Thus, reductions allow us to *prove* hardness of many problems, once we know some hard problem to start with. But can we actually prove that some particular problem is not in $\mathcal{P}$? At least, many natural problems are suspected to be hard in this sense. No polynomial-time algorithms are known for them. Many graph problems are of this type, and also the Knapsack problem. (Remember that our dynamic programming algorithm for Knapsack was not polynomial in the input length!) They seem to resist all our techniques to create fast algorithms. We have no clue how a correct solution to an instance could be built up from solutions to smaller instances in an efficient way. You are welcome to try, but you will always get stuck at some point. Maybe the methods we have learned are too weak for these problems, or too much ingenuity is needed to find the right way of applying the techniques?

The question is: Are we not smart enough, or are the problems intrinsically hard, i.e., outside the class $\mathcal{P}$? In the following we give a cautious "negative" answer.

The Notion of $\mathcal{NP}$-Completeness

Almost all "natural" algorithmic decision problems belong to a certain class of problems that includes $\mathcal{P}$ but is apparently larger. Below we introduce this larger complexity class.

It is common to our problems that we can easily **verify** (confirm, certify) solutions that are already *given*. For example, consider the decision version of Knapsack: Given n items, their weights and values, a capacity W, and a desired total value k, the question is whether some subset of items with total weight no larger than W has a total value of at least k. If somebody supplies us with a solution, we can easily check in polynomial time whether this is in fact a solution: We simply have to add and compare some numbers. Or consider the Independent Set problem: Given a graph and a number k, we can check in polynomial time whether a given subset I of nodes is a valid solution: Count the nodes in I, compare their number to k, and verify for all pairs of nodes $u, v \in I$ that u, v are not joined by an edge. For virtually every natural decision problem we can similarly check an already given solution in a short time.

The complexity class $\mathcal{NP}$ is defined as the class of all decision problems which admit an algorithm that can *verify* every Yes-instance in polynomial time, provided that some "advice" is given, in addition to the input. This "advice" is usually just a solution to the problem instance. In this (typical) case we can say more simply that the mentioned algorithm must verify a given solution in polynomial time.

Some comments on this definition are in order. The verification algorithm is not supposed to *solve* the problem, at least, not in polynomial time. Moreover, the definition does not say *how* the solution is obtained (exhaustive search, a good guess, etc.). It is only concerned with the *verification* of an already available solution. The abbreviatiom $\mathcal{NP}$ stands for **nondeterministic polynomial**, which refers to the interpretation that we may have guessed a solution.

We have $\mathcal{P} \subseteq \mathcal{NP}$. Namely, if we can even *solve* a problem correctly in polynomial time then, trivially, we can also verify in polynomial time that an instance *has* a solution.

As said above, almost every natural, relevant computational problem belongs to $\mathcal{NP}$, and we have that $\mathcal{P} \subseteq \mathcal{NP}$. Is this inclusion strict?! It would be nice to know $\mathcal{P} = \mathcal{NP}$, since this would mean that all these problems are solvable in polynomial time. Unfortunately, the question is open. Moreover, this is perhaps the most famous open question in Computer Science. Nevertheless we can shed some light on this so-called $\mathcal{P}$-$\mathcal{NP}$ question and classify certain problems as "hard". Recall that reductions can be used to compare the difficulty of problems. Now we arrive at the central definition:

A decision problem $Y \in \mathcal{NP}$ is said to be **$\mathcal{NP}$-complete** if every (!) problem $X \in \mathcal{NP}$ is polynomial-time reducible to Y. Informally speaking, $\mathcal{NP}$-complete problems are the hardest problems in $\mathcal{NP}$. We can characterize their hardness as follows:

No $\mathcal{NP}$-complete problem belongs to $\mathcal{P}$, unless $\mathcal{P} = \mathcal{NP}$. Assume for contradiction that some $Y \in \mathcal{P}$ is $\mathcal{NP}$-complete. Then, by definition, all $X \in \mathcal{NP}$ are polynomial-time reducible to Y. But since $Y \in \mathcal{P}$, this implies $X \in \mathcal{P}$ for all $X \in \mathcal{NP}$.

To summarize what we have shown: It is open whether $\mathcal{P} = \mathcal{NP}$ or not, but if not, then no polynomial-time algorithm can exist for $\mathcal{NP}$-complete problems. Since until now nobody could find a fast algorithm for any such problem despite decades of intensive research, it is generally believed that $\mathcal{P} \neq \mathcal{NP}$, and hence all $\mathcal{NP}$-complete problems are really hard.

$\mathcal{NP}$-completeness of any specific problems can be proved via reductions from other such problems, due to the following theorem: If problem Y is $\mathcal{NP}$-complete and polynomial-time reducible to $Z \in \mathcal{NP}$, then Z is also $\mathcal{NP}$-complete. This follows immediately from the definition and from the transitivity of polynomial-time reducibility.

Problem: Satisfiability (SAT)

A **Boolean variable** has two possible values: True (1) or False (0). A **literal** is either a Boolean variable x or its negation $\neg x$. A **Boolean formula** is composed of literals joined by operations AND (conjunction, $\wedge$), OR (disjunction, $\vee$), and perhaps further negations. An **assignment** gives a truth value to every variable in a Boolean formula. An assignment is said to be **satisfying** if the formula evaluates to 1. A **clause** is a set of literals joined by OR. Note that "if-then" conditions can be rewritten as clauses. A Boolean formula is in **conjunctive normal form (CNF)** if it consists of clauses joined by AND. We remark that every Boolean function can be

written equivalently as a CNF formula.

Given: a Boolean formula, either in general form or in CNF.

Goal: Find a satisfying assignment (if there exists some).

Motivations: This is a fundamental problem in logic and related fields like Artificial Intelligence. The following is only an example of a more concrete application scenario.

Certain objects can be described as vectors of Boolean variables, where each variable indicates whether the object has a certain property or not. Suppose that the properties of many objects are stored in a database. Now we want to retrieve an object that satisfies a given set of conditions, expressed as clauses. Before we process an expensive database query, it may be good to check whether the conditions are satisfiable at all, because the given specification may be overconstrained. Furthermore, if the result is positive, we may search the database for occurrences of the satisfying assignments, which is much faster and simpler than testing the conditions for each database entry. (However, the speed depends on the number of satisfying assignments we have to try.)

Appendix

NP Quiz: True or False? (And Why?)

The remarks in the Appendix of Lecture Notes 1 apply also here.

- Consider the following problem. Given three integers x, y, k, decide whether the k-th digit of the product xy, in binary representation, equals 1. Claim: "This problem is in $\mathcal{P}$."
- "The problem in the previous question is in $\mathcal{NP}$."
- "If an optimization problem is in $\mathcal{NP}$ (more precisely: if the corresponding decision problem is in $\mathcal{NP}$), then we can verify in polynomial time that a given optimal solution is, in fact, optimal."

Algorithms. Lecture Notes 9

The "First" $\mathcal{NP}$-Complete Problem

Still we have not seen any single NP-complete problem which could be a starting point for reductions. For one problem, $\mathcal{NP}$-completeness must be proved directly by recurring to the definition. Historically, the first $\mathcal{NP}$-complete problems came from logic: the Satisfiability (SAT) problem for logical formulae (or circuits). The difficulty of SAT, even when restricted to CNF formulae, can be intuitively explained as follows: We may set any variable to 1 in order to satisfy some clause, but the same variable may appear in negated form in other clauses, and then we cannot use it anymore to satisfy these other clauses. Any decisions on the truth value of some variable in a clause restrict the possibilities of satisfying other clauses. This may end up in conflicts where some clause is no longer satisfiable. Then we have to try other combinations of values, etc. In fact, nobody knows how to solve SAT in polynomial time.

Clearly, SAT belongs to $\mathcal{NP}$: If someone gives us a satisfying truth assignment, we can confirm in linear time (in the size of the formula) that it really satisfies the formula. But SAT is probably not in $\mathcal{P}$. A theorem due to S. Cook says that SAT is $\mathcal{NP}$-complete, even for CNF as input.

The proof is long and very technical, but one can give the rough idea in a few sentences: We have to show that any decision problem $X \in \mathcal{NP}$ is polynomial-time reducible to SAT. Since $X \in \mathcal{NP}$, there exists a polynomial-time algorithm that checks the validity of a given solution to an instance x of X. Like every algorithm, it can run on a machine that performs only extremely simple steps, so simple that the internal state of the machine at any time (contents of memory cells etc.) can be described by Boolean variables. A Boolean formula in CNF, of size polynomial in $|x|$, describes the steps of the computation. This CNF is built in such a way that a satisfying truth assignment corresponds to the successful verification of a solution to x. Hence, the whole construction reduces X to SAT in polynomial time. – We emphasize that this was only a brief sketch of the proof.

Don't worry if you find it cryptic. We only need the statement of Cook's theorem, but not its proof.

A side remark: Without Cook's theorem it would not even be clear that $\mathcal{NP}$-complete problems Y exist at all! Remember that the definition requires that *all* problems of the class are reducible to Y, which is a strong demand. This is also the reason for introducing the "strange" class $\mathcal{NP}$: The considered problems must be limited somehow, but to some class that is still large enough to capture most of the relevant computational problems.

3SAT is as Hard as SAT

Surprisingly, some further restriction does not take away the hardness of the SAT problem: A kCNF is a CNF with at most k literals in every clause. kSAT is the SAT problem for kCNF formulae. We show that 3SAT is still $\mathcal{NP}$-complete, by a polynomial-time reduction from the (more general!) SAT problem for arbitrary CNF.

This reduction is best described by an iterative algorithm doing the transformation. Given a CNF, we do the following as long as some clause C with more than 3 literals exists: We split the set of literals of C in two shorter clauses A and B, append a fresh variable u to A and $\bar{u}$ to B. A fresh variable means that u must not occur in any other clause. It is easy to see that $(A \vee u) \wedge (B \vee \bar{u})$ is satisfiable if and only if $C = A \vee B$ is satisfiable. Similarly, the entire formula is satisfiable before the modification if and only if it is satisfiable after the modification. Hence we have got an equivalent instance of the problem. After a polynomial number of steps we are down to a 3CNF.

Stop! It is not completely obvious that the number of steps is bounded by a polynomial. One needs an argument for that. Here is one possibility. We have not epecified exactly how the literals of C are divided in two graoups. We can always put exactly 2 literals in A and the rest in B. Then $(A \vee u)$ has exactly 3 literals, and $(B \vee \bar{u})$ is strictly shorter than $A \vee B$. Hence we have strictly decreased the total length of the long clauses with more than 3 literals. This can happen only linearly many times.

We can also get rid of clauses with 1 or 2 literals in a polynomial number of steps. Namely, ee can make a clause A artificially longer, similarly as above: Take a fresh variable u and replace A with $(A \vee u) \wedge (A \vee \bar{u})$, this time without splitting the clause. It follows that the version of 3SAT with *exactly* 3 literals per clause remains $\mathcal{NP}$-complete.

Next, what about 2SAT? The above reduction cannot produce an equiv-

alent 2CNF. (Do you see why not?) Actually, 2SAT is in $\mathcal{P}$. It can even be solved in linear time, through a rather nontrivial graph algorithm that we cannot show here.

Some Further $\mathcal{NP}$-Complete Problems

3SAT is an excellent starting point for further NP-completeness proofs. The limitation to three literals per clause makes it nice to handle. Next we reduce 3SAT to Independent Set, thus proving in one go the NP-completeness of Vertex Cover, Independent Set, and Clique.

Let us be given an instance of 3SAT, more precisely, a 3CNF with n variables and m clauses, and with exactly 3 literals in every clause. The reduction constructs a graph as follows.
(1) For each variable we create a pair of nodes (for the negated and unnegated variable), joined by an edge.
(2) For each clause we create a triangle, with the 3 literals as nodes.
These pairs and triangles have together $2n + 3m$ nodes.
(3) An edge is also inserted between any node in a pair (1) and any node in a triangle (2) which are labeled with identical literals.

One can show that the problem instances are equivalent: The 3CNF formula is satisfiable if and only if this graph has an independent set of $k = m + n$ nodes. This needs a little thinking, but the proof steps are straightforward. (For a better understanding it can be advisable to take a little example of a 3CNF, draw the resulting graph, and verify the claimed equivalence for the example and then in general.)

Reductions from a problem X to a probelm Y like this are called "gadget constructions" in the literature, because the building blocks of an instance of X are encoded by "gadgets", which are the building blocks of instances of Y. In our case, variables and clauses of a 3CNF are encoded by node pairs and triangles, which are our gadgets. You are not expected to find such gadget constructions yourself, but only to understand given ones.

The $\mathcal{NP}$-completeness of many problems has been established by chains of such reductions, among them the famous **Traveling Salesman** problem, **Coloring** the nodes of a graph with 3 colors, several partitioning, packing and covering problems, numerical problems like **Subset Sum** and (hence) **Knapsack**, various scheduling problems, and many others. $\mathcal{NP}$-complete problems appear in all branches of combinatorics and optimization. We give another natural example that is also useful for further reductions:

Problem: Set Packing

Given: a family of subsets of a finite set.

Goal: Select as many as possible of the given subsets that are pairwise disjoint.

This resembles Interval Scheduling, but now the given subsets can be any sets, rather than being intervals in an ordered set.

We show that Set Packing is NP-complete, by a polynomial-time reduction from Independent Set: Given a graph $G = (V, E)$ and a number k, we construct the following family of subsets $S(v) \subset E$, one for every node $v \in V$: We define $S(v)$ to be the set of all edges incident with v, the "star with center v", so to speak. Note that two stars are disjoint if and only if their centers are not adjacent. Thus, G contains an independent set of k nodes if and only if we can select k pairwise disjoint sets from this family.

Exponential Time Hypothesis (ETH)

The ETH claims, roughly speaking, that no algorithm can solve 3-SAT in a time better than $O(a^n)$, where $a > 1$ is some constant. (At least, this is a variant of ETH that is easy to formulate. There are other variants of different strengths.)

Like the $\mathcal{P} \neq \mathcal{NP}$ hypothesis, it is not known whether ETH is true, however it is widely accepted as a hypothesis. ETH would obviously imply $\mathcal{P} \neq \mathcal{NP}$, but the converse is not clear.

ETH has the advantage that it claims some explicit lower time bound. Conditional on ETH one can prove lower bounds also for other problems via reductions. However one must be more careful with the time bounds of these reductions.

Some Frequent Misconceptions

To prove $\mathcal{NP}$-completeness of a problem Y, one must give a polynomial-time reduction **from** a known $\mathcal{NP}$-complete problem X to Y, not a reduction from Y **to** X. Remember that Y is harder than X (more precisely: at least as hard) and not easier.

An explanation why the direction of reductions is often confused might be a misconception around the word "reduction". In every-day use, to "reduce" something usually means to make it smaller, and this may be misunderstood as "making the complexity smaller", but here it is the other way round! The word "transformation" would perhaps avoid this misunderstanding, but "reduction" is the established term.

Furthermore notice that polynomial-time reducibility is not a symmetric relation. If X is reducible to Y, this does in general not imply that Y is also reducible to X. A reduction goes in only one direction, but *inside* a reduction one must show equivalence of the instances, which involves two directions: (1) If x is Yes then $f(x)$ is Yes, and (2) if $f(x)$ is Yes then x is Yes. Moreover, this must hold true for every instance x of X, whereas not every instance y of Y is required to be some $f(x)$. In other words, function f is not necessarily surjective. All these aspects are easy to confuse, but this is only a matter of carefully learning the definitions and reflecting why they are as they are.

Sometimes it is claimed in reports that $\mathcal{NP}$ means "not polynomial", which is complete nonsense. Finally, one should carefully distinshuish between "$\mathcal{NP}$-problems" (that is, problems in $\mathcal{NP}$, which also includes $\mathcal{P}$), and "$\mathcal{NP}$-complete problems". Here, sloppy naming can easily produce wrong statements.

Similarly, sometimes it is claimed that $\mathcal{P} \neq \mathcal{NP}$ would imply that $\mathcal{NP}$-complete problems can only be solved in exponential time. However, such exponential lower bounds are not known.

$\mathcal{NP}$-Completeness; Wrap-Up

What should you (at least) have learned about $\mathcal{NP}$-completeness in a basic algorithms course? You should:

- have understood the concepts on a technical level (not only vaguely that $\mathcal{NP}$-complete problems are "somehow difficult"),
- have understood their relevance,
- be able to carry out *simple* reductions (doing complicated reductions is clearly something for specialized scientists in the field),
- know some representative $\mathcal{NP}$-complete problems,
- know where to find more material.

If, in practice, a computational problem is encountered that apparently does not admit a fast algorithm, it is a good idea to look up existing lists of $\mathcal{NP}$-complete problems. Maybe the decision version Y of the problem at hand is close enough to some problem X in a list, and a polynomial-time reduction from X to Y can be established. Then it is clear that Y must be treated with heuristics, with suboptimal but fast approximation algorithms, or with exact but slow algorithms.

A classic reference with hundreds of $\mathcal{NP}$-complete problems is:
Garey, Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Francisco 1979.

Other repositories are on the Web.

Appendix

NP Quiz Continued: True or False? (And Why?)

The remarks in the Appendix of Lecture Notes 1 apply also here.

- "Earlier we have seen dynamic programming algorithms for Subset Sum and Knapsack, but now $\mathcal{NP}$-completeness is claimed. This is a contradiction. Something must be wrong with this theory."
- "The Clique problem is polynomial-time reducible to the Knapsack problem."
- "The Knapsack problem is polynomial-time reducible to the Clique problem."
- "The Independent Set problem is $\mathcal{NP}$-complete for arbitrary graphs. Interval graphs are special graphs. It follows that the Independent Set problem for interval graphs is $\mathcal{NP}$-complete."
- "If, for some problem, we know some powerful heuristic that solves typical instances efficiently, we can conclude that this problem is not $\mathcal{NP}$-complete, because those problems are difficult to solve."

Algorithms. Lecture Notes 10

Graph Traversals

From now on, $G = (V, E)$ denotes a graph with $n = |V|$ nodes and $m = |E|$ edges.

Graph traversals are techniques to visit all nodes in a graph $G = (V, E)$ in a fast and systematic way. They provide a basis for several efficient graph algorithms. We consider directed graphs $G = (V, E)$ and denote a directed edge from u to v by (u, v). Note that undirected graphs may be considered as special directed graph where both directed edges (u, v) and (v, u) exist, for every pair of adjacent nodes u and v.

Perhaps the simplest traversal strategy is **Breadth-First-Search (BFS)**. (Don't forget the "d" in "breadth" ...) It starts in one node s which is put in a queue and marked. In every step, BFS takes the next node u from queue and visits *all* unmarked nodes v such that $(u, v) \in E$. Every such v is put in the queue and marked. BFS stops as soon as the queue is empty.

We study some properties of BFS. First of all, BFS partitions the set of nodes into layers L_i, $i \geq 0$, inductively defined as follows. L_0 contains only the start node s, and L_{i+1} contains all nodes v such that: an edge $(u, v) \in E$ for some $u \in L_i$ exists, and v is not already in an earlier layer. It is easy to see that BFS, implemented with a queue, processes the nodes exactly layer by layer. More importantly, the layers provide some useful structure: Edges (u, v), with $u \in L_i$, $v \in L_j$ go at most to the next layer, that is, $j \leq i + 1$. (But j can be arbitrarily smaller than i.) It follows that L_i contains exactly the nodes with (directed) distance i from s, in other words, the nodes reachable from s on a directed path with i (but not fewer than i) edges. Hence BFS as such yields an algorithm for the Shortest Paths problem, provided that all edges have unit length.

Take some time to think about the proof of the last assertion. One must verify two things for every node $t \in L_i$: (a) There *exists* some directed path of length i from s to t. (b) There is *no shorter* directed path from s to t.

BFS also gives rise to a directed tree which contains all marked nodes and a certain subset of the edges from E: Whenever a node v is found for the first time, via the edge (u, v), we insert this edge in the tree. In fact, this yields a tree rooted at s, since every node except s has exactly one predecessor. We refer to it as the **BFS tree**. Note that all edges in the BFS tree go from a layer to the next layer (but in general not all edges to the next layer are in the BFS tree).

To analyze the time for BFS, note that every edge is considered only once. The crucial step is to determine the nodes v with $(u, v) \in E$, for a given u. The time for this operation depends on the way the graph is represented. When adjacency lists are used, we simply need to traverse the list for u, thus we spend only constant time on every edge. We conclude that BFS needs $O(m)$ time. (This simple argument in the time analysis is very common, for a number of graph algorithms.) If an adjacency matrix is used, we need $O(n^2)$ time, which is in general worse. Namely, for the node u considered in each step we have to check all matrix entries in u's row, even in the case that almost all of them are 0.

The other standard graph traversal strategy is **Depth-First-Search (DFS)**. It starts in a node s and follows a directed path of yet unexplored nodes, as long as possible. When it reaches a dead end (where all successor nodes of the current node are already explored), it goes one step back on the path, looks for another unexplored successor node, and so on.

The most compact formulation is a recursive procedure DFS(u) with start node u as input parameter (the main program is to call DFS(s)): As long as unmarked nodes v with $(u, v) \in E$ exist, choose one such v, mark v, and call DFS(v). – Since each recursive call is done only after termination of the previous call, this gives the desired depth-first behaviour. DFS can also be written as an iterative program, but then the stack must be implemented explicitly.

DFS exhibits some similarities to BFS. The time for DFS is $O(m)$ when adjacency lists are used to collect all successors of a node. A **DFS tree** can be defined as follows: Edge (u, v) belongs to the DFS tree if DFS(u) calls DFS(v). Such edges (u, v) are said to be **tree edges**. Indeed, they form a tree, since v becomes the input parameter of a recursive call only once, and then v gets marked.

But there are also major differences to BFS. They concern the positions of edges from E which are *not* in the DFS tree:

In undirected graphs, such edges can only go from a node to an ancestor node in the DFS tree. This follows easily from the rules of DFS. We call them **back edges**. There exist no **cross edges**, that is, edges joining nodes from different paths of the DFS tree. (Why not? To understand the reason, assume for contradiction that a cross edge exists, and see how DFS would have produced it ...)

In directed graphs this issue is somewhat more complicated. Directed edges which are not in the DFS tree can be divided into three types: **forward edges** going from a node to some descendant node, **back edges** going from a node to some ancestor node, and **cross edges** going from a node to another node on an "earlier" directed path of the DFS tree. – These structural properties are useful in several graph algorithms based on DFS.

Another remark: We had observed that BFS solves the Shortest Path problem in the case of unit edge lengths. By way of contrast, DFS has nothing to do with shortest paths! (Actually, this is a frequent misconception, so make sure to avoid that mistake.)

Problem: Undirected Graph Connectivity

An undirected graph is **connected** if there exists a path between any two nodes. The **connected components** are the maximal connected subgraphs.

Given: an undirected graph $G = (V, E)$.

Goal: Decide whether G is connected. If not, compute the connected components.

Problem: Strong Connectivity in Directed Graphs

A directed graph is **strongly connected** if there exists a *directed* path from every node to every node. The **strongly connected components** are the maximal strongly connected subgraphs.

Given: a directed graph $G = (V, E)$.

Goal: Decide whether G is strongly connected. If not, compute the strongly connected components.

Motivations:

If the graph models states of a system and possible transitions between them, strong connectivity means it is always possible to recover every state, i.e., the system has no irreversible moves. The street map of a city with one-way streets should be strongly connected as well, or the traffic planners made a mistake.

Some Applications of BFS and DFS: Connectivity

Testing connectivity of a graph can easily be misjudged as a trivial problem, but without some systematic strategy we would aimlessly walk around in the labyrinth of the graph and use much more time than necessary. Graph traversal solves several connectivity problems efficiently, as we will see now.

BFS starting in node s in a graph G reaches exactly those nodes being reachable from s on directed paths. The same is true for DFS. In particular, if G is undirected, then the traversal explores exactly the connected component of G which contains s. This gives an $O(m)$ algorithm to test whether an undirected graph G is connected: Run BFS or DFS, with an arbitrary start node. G is connected if and only if all nodes are reached. We can also determine the connected components of G in $O(m+n)$ time: If the search has aborted without finding all nodes, restart the search in a yet unmarked node, and so on.

Connectivity is more intricate in directed graphs. Still, strong connectivity can be checked in $O(m)$ time. But first we give a naive algorithm: Run BFS (or DFS) twice for evry start node s: once on the given directed graph and once on the reversed graph where all directe edges (u, v) are replaced with their opposite edges (v, u). Thus we find all nodes t being reachable from s, and we find all nodes t from which s is reachable. The graph is strongly connected if and only if the result is positive for all pairs s, t. This algorithm needs $O(nm)$ time, for running BFS n times.

But a little thinking yields a much better algorithm: Run BFS twice (as above), but with only one arbitrary but fixed start node s. The graph is strongly connected if and only if both BFS runs reach all nodes. This is correct because, in a strongly connected graph, one can get from every node to every node also via the fixed node s. (Here we are only interested in the existence of some path, not in its length, hence detours are not an issue.) This algorithm needs only $O(m)$ time. Note that a little mental work (in algorithm design and correctness proof) has saved a factor n

If the graph is *not* strongly connected, then the last algorithm determines the strongly connected component which contains s: It is the set of nodes reached in both the given graph and the reversed graph. One can obviously extend this algorithm, in order to partition the graph into its strongly connected components. Hovewer, we may need $O(nm)$ time again: In the worst case, the graph may have many small strongly connected components, but we may need $O(m)$ time to determine each one in this way. Actually, it is possible to compute even all strongly connected components in $O(m)$ time by some sophisticated use of DFS, but we have to skip this theme.

Problem: Graph Coloring

Given a set of k colors, a **k-coloring** of a graph assigns a color to each vertex, so that adjacent vertices get different colors. A graph is **k-colorable** if it admits a k-coloring. The 2-colorable graphs are exactly the bipartite graphs.

Given: an undirected graph $G = (V, E)$ and an integer k.

Goal: Construct some k-coloring of G, or report that G is not k-colorable.

Motivations:

Imagine that a person who is not exactly an expert in botany gets a set of plants, and he is told that they belong to two different species. He does not always see whether two plants belong to the same species or not, however, *some* pairs of plants are obviously different. Is it possible for him to divide the set correctly and efficiently? This can be translated into the 2-coloring problem: Every species (class, category, etc.) is represented by a "color". The plants (or whatever objects) are nodes of a graph $G = (V, E)$, where any two nodes that are *known* to belong to different classes are joined by an edge. The 2-colorable graphs are also called bipartite graphs.

Various problems dealing with packing, frequency assignment, job assignment, scheduling, partitioning, etc., can be considered as Graph Coloring, where the graph models pairwise conflicts. Note that Interval Partitioning problem is a special case of Graph Coloring, with the goal to minimize the number of colors: Intervals are nodes, two nodes are adjacent if the corresponding intervals overlap, and the "colors" are copies of the resource.

One Graph and Two Colors

We conclude with another simple application of BFS: The 2-coloring problem is solvable in $O(m)$ time. The key observation is: If a node gets one color, then all adjacent nodes *must* get the other color, and so on. This is, a bit implicitly, already the correctness proof of the following algorithm. BFS merely serves as a framework to organize the enforced coloring efficiently.

Now the algorithm in detail: We compute the BFS tree and the layers. Then, all nodes in the layers L_i, i even, get one color, and all nodes in the layers L_i, i odd, get the other color. Since each node in L_{i+1} is joined to some node in L_i via some edge of the BFS tree, essentially only one valid 2-coloring can exist in each connected component. The only degree of freedom is that we can swap the two colors. (Alternatively one may also use DFS, but then the details are, of course, different.)

The idea cannot be extended to $k > 2$ colors, because the color of a node does no longer determine the color of all neigbored nodes. We have the choice between different colors, and it is not clear how we could safely avoid later coloring conflicts.

Actually, k-coloring is $\mathcal{NP}$-complete for every $k \geq 3$. This can be shown by a reduction from 3SAT being somewhat similar to the reduction from 3SAT to Independent Set.

Problem: Minimum Spanning Tree

A **spanning tree (MST)** in an undirected graph $G = (V, E)$ is a tree that contains all nodes of V (it "spans" the graph) and a subset of edges taken from E.

Given: a connected undirected graph $G = (V, E)$ where every edge has some positive cost.

Goal: Construct a spanning tree T in G with minimum total cost (sum of costs of all edges in T).

Motivations:

This is a basic network design problem. It appears when certain sites have to be connected in the cheapest way by streets, cables, virtual links, or whatever. Edge costs may represent lengths, costs of material, or other

costs of the links. Note that a minimum-cost connected spanning subgraph of G is always a tree, since if there were a cycle, we could remove some edge without destroying connectivity.

Appendix

BFS/DFS Example

Some minimalistic example illustrates the differences of the two main traversal techniques. Consider an undirected graph with node set $V = \{a, b, c, d\}$ and edge set $E = \{ab, ac, bc, bd\}$. (Here we use a common "lazy" notation for undirected edges; for instance, ab means the edge between a and b.) Recall that an undirected edge equals a pair of opposite directed edges.

BFS with start node a yields $L_0 = \{a\}$, simply by definition, $L_0 = \{b, c\}$, as these are all neighbors of a, and $L_2 = \{d\}$, as this node has a neighbor in L_1, namely b. The layers are uniquely determined. In this example, the BFS tree is also uniquely determined (containing all edges except bc). In larger examples, however, the BFS tree may depend on the ordering of nodes within the layers, because a node can in general have several neighbors in the previous layer.

What happens in DFS with start node a? This heavily depends on the order in which we consider the adjacent nodes. If we continue with c, then we will follow the path $a - c - b - d$. Then this path is also the DFS tree, and the edge ab is a back edge. If we, alternatively, visit b immediately after a, we get a DFS tree with two overlapping paths: $a - b - c$ and $a - b - d$. The edge ac becomes a back edge on the first path. Note that there are (of course) no cross edges.

Algorithms. Lecture Notes 11

Algorithms for Minimum Spanning Trees (MST)

The MST problem is one of the most prominent algorithmic problems on graphs. It is not only important in its own right. Here we also use it as yet another illustration of some general issues in algorithm design.

Assume in the following that no two edge costs are equal.

Let us do some rough problem analysis first. Remember that a greedy approach is preferable *if* some greedy rule works for the problem at hand. We would like to specify edges that we can safely put in an MST. It would be natural to choose "somehow" the cheapest edges that do not form cycles. However, there are several ways to turn this idea into an algorithm, and we must prove correctness, probably by some exchange argument.

In fact, there is an elegant lemma about MST, proved by an exchange argument. The exact form of this lemma is not so obvious. (But here we skip the trial-and-error development steps.) The idea is that a spanning tree is still a connected graph, hence it must contain some edge between any two subsets of nodes that are complements. What if we always take the cheapest such edge?

Here comes the lemma: If we partition the node set V arbitrarily in two non-empty sets X and Y, then the cheapest edge $e = (x, y)$ with $x \in X$ and $y \in Y$ must belong to any MST.

For the proof, assume that T is an MST not containing e. Insertion of e in T yields a cycle C. This cycle C must contain another edge $f = (u, v)$ with $u \in X$ and $v \in Y$. By removing f from the cycle we get another spanning tree. Since e was cheaper than f, this spanning tree is cheaper than T, a contradiction.

There exist two different greedy algorithms for MST following the above idea. They are attributed to their inventors Prim and Kruskal, and both are easy to describe in one sentence:

Prim's algorithm starts from an arbitrary node (a "tree" with empty edge set), and always adds a cheapest edge that extends the current tree to a larger tree.

Kruskal's algorithm starts from an empty edge set and always inserts a cheapest edge that does not create cycles with already selected edges.

Equivalently, Kruskal's algorithm always adds a cheapest edge that connects two distinct connected components of already selected edges. In other words, it maintains a **forest** (a partitioning of the node set into a union of trees) and adds a cheapest edge to the forest in every step. For clarity, note that every isolated node without edges is also considered a tree in the forest. The initial forest has no edges.

To prove that these algorithms never choose an erroneous edge, we apply the above lemma. Let T denote the MST.

In every step of Prim's algorithm, we define X to be the set of nodes spanned by the already chosen edges. (Before the first step, X contains only the start node.) Since Prim's algorithm chooses the cheapest edge e between X and Y, we conclude that e belongs to T.

In every step of Kruskal's algorithm, we denote by F the set of the already chosen edges. Let $e = (x, y)$ to be the edge that the algorithm would choose next. We put the connected component of F containing x in X, and we put the connected component of F containing y in Y. All other connected components of F are put in X or Y arbitrarily. Now we have partitioned the node set in two disjoint non-empty sets X and Y, We claim that e is the cheapest edge between X and Y (and thus e belongs to T). If there were a cheaper edge f between X and Y, then f would also satisfy the condition to connect two distinct components of F. But then Kruskal's algorithm would prefer f to e, a contradiction.

Thanks to the lemma, these proofs are elegant, aren't they?

Finally, we can et rid of the restriction that all edge costs be distinct: If edges with equal costs appear in G, we add sufficiently small distinct costs (so caleld perturbations) to them. The above result shows that the algorithms are correct when applied to these distinct costs. But the resulting edge set is an MST also for G with the original edge costs, if the sum of perturbations is small enough, compared to the edge costs. For both algorithms this simply means that we can replace the phrase "*the* cheapest edge" with "*some* (arbitrary) cheapest edge".

A third, less prominent greedy algorithm for MST starts from the given edge set E and always *deletes* the most expensive edge, keeping the graph connected. Correctness is proved along the same lines. We do not further

study this third algorithm. For dense graphs it is slower than the others, as it has to delete most edges.

Implementing Prim's Algorithm

In every step we must find the cheapest edge between the tree X and the remaining set node set Y. A naive implementation that determines every such edge from scratch needs $O(nm)$ time; n steps, each considering m edges. One feels that this is very redundant. X and Y change only by a single node in every step, so why should we compare all edge costs again and again?

A better idea is to maintain a small set of "candidate" edges: Specifically, for every node $y \in Y$ we may store, in some data structure, the cheapest edge (x, y) connecting y with X. Clearly, the cheapest edge between X and Y is one of these candidate edges. Updating the data structure requires $|Y| < n$ comparisons in each step: One node v moves from Y to X. Hence we only have to check for every node $y \in Y$ whether the edge (v, y) is cheaper than the currently stored edge (x, y). Altogether this improves the total running time to $O(n^2)$.

An alternative implementation works with a data structure that explicitly provides what we need in every step, namely the cheapest element in the set H of edges between X and Y. It uses a standard data structure that supports fast insertion and deletion of items, and fast delivery of the currently smallest item in H. It is known as **priority queue**. Now the key observation is: Every edge enters H only once and leaves H only once. (Why?) That is, we have to perform $O(m)$ insert and delete operations. The minimum element in F must be determined $n - 1$ times. It is possible to create a priority queue that executes every operation in $O(\log m)$ time, where m is the maximum size of the set to maintain. Thus, Prim's algorithm can run in $O(m \log m)$ time, which can also be written as $O(m \log n)$.

Both implementations of Prim's algorithm are justified: $O(n^2)$ is somewhat faster if the graph is dense (has a quadratic number of edges), but otherwise $O(m \log n)$ is considerably faster.

Implementing Kruskal's algorithm in a good way is more tricky and therefore not considered in this section.

Problem: Detecting Directed Cycles

A **directed cycle** in a directed graph is a cycle that can be traversed respecting the orientation of the edges. In other words, it is a sequence of nodes $v_1, v_2, v_3, \ldots, v_n, v_1$ where every (v_i, v_{i+1}) and (v_n, v_1) is a directed edge. A *directed acyclic graph* (DAG) is a directed graph without directed cycles. DAGs should not be confused with trees which are connected graphs without *any* cycles (which are in general undirected).

Given: a directed graph $G = (V, E)$.

Goal: Find a directed cycle in G, or report that G is a DAG.

Motivations:

Directed cycles are undesirable in plans of tasks where directed edges (u, v) model pairwise precedence relations (task u must be done before task v). These tasks can be calculations in a program or logic circuit, jobs in a project, steps in a manufacturing process, etc. In such models, directed cycles indicate errors in the design.

Some systems in Artificial Intelligence, so-called partial order planners, automatically create plans to achieve some goal, given a formal description of the goal and of available actions. Part of the construction algorithms are tests for directed cycles. If such cycles are detected in a plan, some actions must be removed, and the corresponding partial goals must be realized in a different way, avoiding new cycles.

Problem: Topological Order

A **topological order** of a directed graph $G = (V, E)$ is an order of all nodes of V so that all directed edges go to the right. In other words, for every directed edge (u, v), node u appears earlier than node v in the order.

Given: a directed graph $G = (V, E)$.

Goal: Construct a topological order of G, or report that G does not admit a topological order.

Motivations:

The nodes are jobs with pairwise dependency constraints, as above. Any topological order is a possible order of executing these jobs without violating the precedence constraints.

Directed Acyclic Graphs (DAGs) and Topological Order

We continue with fast algorithms that detect directed cycles or construct a topological order (if existing) in a directed graph G.

Looking at the specifications of these problems, perhaps one can easily guess that G is a DAG if and only if G allows a topological order. The "if" direction is obvious: If all edges go in the same direction, one can never close a directed cycle. The "only if" direction is more interesting, and it has a **constructive proof** in the sense that the proof also shows how to obtain a topological order, provided that G is a DAG. The proof can be done by induction on the number of nodes, as shown below.

Observe that the first node v in a topological order must not have any incoming edges (u, v). Conversely, an arbitrary node v without incoming edges can be put at the first position of a topological order. Here comes the inductive argument why this is true: Remove v and all incident edges from G. The remaining graph is still a DAG. (No directed cycles can be created by removing parts of a graph.) Hence, G without v has a topological order (by the induction hypothesis), and by setting v in front of this topological order, we get a topological order of the entire graph G.

The resulting algorithm is very simple: For $k = 1, \ldots, n$, put an arbitrary node v without incoming edges at the k-th position of the topological order, and remove v and all incident edges from the graph.

This algorithm is obviously correct if it goes through. But how do we know that there always exists such a node v to continue with? Assume by way of contradiction that every node has an incoming edge. Then we can traverse a path of such edges in opposite direction. But since G is finite, we must sometimes meet a node again, contradicting the assumption that G has no directed cycle.

A remarkable detail is that we can always take an *arbitrary* node v without incoming edges. As the proof shows, we can never get stuck and miss a topological order by an unlucky choice of v in some step. In a sense, we can consider this algorithm a greedy algorithm (taking local decisions without looking ahead), although it is not concerned with an optimization problem.

Fast Construction of a Topological Order

But how much time is needed? We first consider an alternative way, also in order to mention some application of DFS.

A possible $O(n+m)$ time algorithm to construct a topologocial order of a given DAG uses DFS and is based on another equivalence: G is a DAG if and only if directed DFS, with an arbitrary start node, does not yield any back edges. To see the "only if" part, note that a back edge (u, v) together with the tree edges on the path from v to u form a cycle. The proof of the "if" direction gives a method to construct a topological order: Run DFS again, but with two modifications: Ignore all edges that are not in the DFS tree, and call the children of each node in reverse order (i.e., compared to the first run). Append each node to the result, as soon as it is marked as explored. Since there are no back edges, and all cross edges go "to the left", it is not hard to see that we actually get a topological order.

As you may have noticed, this DFS-based algorithm is conceptually more complicated than the one based on the equivalence we proved in the previous section. We mention it only for comparison. (Therefore, do not worry if the details are unclear.) Instead let us aim for an efficient implementation of the indegree-based algorithm instead. The in-degree of a node v in a directed graph is the number of incomig directed edges (u, v).

Remember that, in every step, an arbitrary node with in-degree 0 (within the remainder of the graph) can be chosen as the next node in a topological order. But how do we find such a node in every step? Naive search from scratch is unnecessarily slow. But some good ideas can be obtained straightforwardly: Removal of parts of a graph can only decrease the in-degrees of nodes. This suggests counting and queuing: In the beginning, count all incoming edges at every node. This is done in $O(n + m)$ time. Put all nodes with in-degree 0 in a queue. In every step, take any node u from queue, and subtract 1 from the in-degrees of all nodes v with $(u, v) \in E$. Since this is done only once for every edge (u, v), updating the in-degrees costs $O(m)$ time in total, provided that G is represented by adjacency lists. Altogether, we can recognize DAGs and construct a topological order in $O(n+m)$ time, also without DFS.

Problem: Longest Paths

Given: an undirected or directed graph $G = (V, E)$, the lengths $l(u, v)$ of all edges $(u, v) \in E$, and a start ("source") node $s \in V$.

Goal: For all nodes $x \in V$, compute a (directed) path from s to x with maximum length, but such that no node appears repeatedly on the path.

The Shortest Paths problem with a source s is similarly defined.

Motivations:

Finding longest paths is not a silly problem. In particular, it makes much sense on DAGs. For example, if the DAG is the plan of a project with parallelizable tasks modelled by the edges, and the edge lengths are execution times, then the longest path in the graph gives the necessary execution time (makespan) for the whole project. It is also known as the critical path.

Algorithms. Lecture Notes 12

Shortest and Longest Paths in DAGs

Shortest paths in directed graphs with unit edge lengths can be computed by BFS, as we have seen. An extension of this shortest-path algorithm to directed graphs with arbitrary edge lengths is Dijkstra's algorithm that we do not present here. (It may be known from data structure courses. Also remember that this course is primarily about algorithm design and analysis techniques, not about specific algorithms.) Anyway:

The Shortest Paths problem in DAGs is much easier to solve than in general graphs. We can take advantage of a topological order, constructed in $O(n + m)$ time. Since paths must go strictly from left to right, we may suppose that the source s is the first node in the topological order. All nodes to the left of s can be ignored. Let $l(u, v)$ denote the given length of the directed edge (u, v), provided that it exists, and let $d(u, v)$ denote the length of a shortest path from u to v. Assume that we have already computed the values $d(s, x)$ for the first $k - 1$ nodes x in the topological order. Let z denote the k-th node. Then we have $d(s, z) = \min d(s, x) + l(x, z)$, where the minimum is taken for all x to the left of z. Correctness is evident, since the predecessor of z on the path must be one of the mentioned nodes x. This dynamic programming algorithm needs only $O(m)$ time, because we look at every edge only once.

Remark for those who know Dijkstra's algorithm well (others may skip it): The first $k - 1$ nodes in the topological order are, i general, not the $k - 1$ nodes closest to s, because topological order and edge lengths are not related. Therefore, the above algorithm is not Dijkstra's algorithm applied to DAGs. It treats the nodes in a different order.

Next we also want to find *longest* paths from s to all nodes in a DAG. Amazingly, we can apply the same algorithm, replacing min with max. Why is this correct? Think about this question. (In general directed graphs we cannot simply take Dijkstra's algorithm and replace min with max. This

would not yield the longest path. Actually the problem is NP-complete in general graphs. What is different in DAGs – why does it work here?)

We remark that the dynamic programming algorithms for many other problems (e.g. Sequence Comparison) can be interpreted as shortest- or longest-paths calculations in certain DAGs derived from those problems. More formally, we can reduce them to Shortest (or Longest) Paths in DAGs. You may revisit some of these problems and figure out how their DAGs look. Nevertheless it is still advantageous to use special-purpose algorithms for those problems, because their DAGs have some highly regular structures, such that memoizing optimal values in arrays is in practice faster than (unnecessarily) dealing with data structures for arbitrary DAGs.

Union-and-Find

This is an addendum to Kruskal's algorithm for MST. In an endeavor to achieve a good time bound we face two problems: finding the cheapest edge, and checking whether it creates cycles together with previously chosen edges. (In that case, the algorithm skips the edge and goes to the next cheapest edge, and so on.) The first problem is easily solved: In a preprocessing phase we can sort the edges by ascending costs, in $O(m \log n)$ time, and inside Kruskal's algorithm we merely traverse this sorted list.

Checking cycles is more tricky. Remember that the already selected edges build a forest. Every node belongs to exactly one tree in this forest. The key observation is that a newly inserted edge does not create cycles if and only if it connects two nodes from different trees in this forest.

Thus, we would like to have a data structure that maintains partitionings of a set (here: of the node set V) into subsets (here: the forests), each denoted by a label, and supports the following operations: Find(i) shall return the label of the subset of the partitioning that contains the element i. Union(A, B) shall merge the subsets with labels A and B, that is, replace these sets with their union $A \cup B$ and assign a label to it. (In the following we will not clearly distinguish between a set and its label, just for convenience.) Such a data structure is not only needed in Kruskal's algorithm. It also appears in, e.g., the minimization of the set of internal states of finite automata with specified input-output behaviour. We cannot treat the latter subject here. This is just mentioned to point out that the Union-and-Find data structure is of broader interest and is not a "one-trick pony".

The problem of making an efficient data structure for Union-and-Find

is nontrivial. A natural approach is to store all elements, together with the labels of sets they belong to, in an array. Then, Find(i) is obviously performed in $O(1)$ time, by looking at the table entry of i. To make the Union(A, B) operations fast, too, we could store every set A separately in a list, along with the size $|A|$. (Without that, we would have to collect the elements of A, which are spread out in the array ...) Now, each element appears twice: in the global array and in the "compact" list of the set it belongs to. These two copies of each element may be joined by pointers.

Now we describe how to perform Union(A, B): Suppose that $|A| \leq |B|$; the other case is symmetric. It is natural to change the labels of all elements in the smaller set from A to B, as this minimizes the work needed to update the partitioning. That is, we traverse the list of A, use the pointers to find these elements also in the array, change their labels to B, and finally we merge the lists of A and B and add their sizes. Note that the union is now named B.

The analysis of this data structure is quite interesting. Any single Union(A, B) operation can require $O(n)$ steps, namely if the smaller set A is already a considerable fraction of the entire set. However, we are not so much interested in the worst-case complexity of every single Union operation. In Kruskal's algorithm we need $n - 1 = O(n)$ Union operations and $O(m)$ Find operations. The latter ones cost $O(m)$ time altogether. The remaining question is how much time we need *in total* for all Union operations. Intuitively, the aforementioned worst case cannot occur very often, therefore we should not rush and conclude a poor $O(n^2)$ bound.

Instead of staring at the worst case for every single Union operation we change the viewpoint and ask how often every element is relabed and moved! That is, we sum up the elementary operations in a different way. An element is relabeled in a Union operation only if it belongs to the smaller set. Hence, after this Union operation it belongs to a new set of at least double size. It follows immediately that every element is relabeled at most $\log_2 n$ times, because this is the largest possible number of doublings. Thus we get a time bound $O(n \log n)$ for all $n - 1$ Union operations together. This is within the $O(m \log n)$ bound that we already needed to sort the edges in Kruskal's algorithm.

Thus, the above Union-and-Find data structure is "good enough" for Kruskal's algorithm. However, a faster Union-and-Find structure would further improve the physical running time and might also be useful within other algorithms. We briefly sketch a famous Union-and-Find data struc-

ture that is faster. (The following paragraph is extra material and may be skipped.)

We represent the sets of the partitioning as trees whose nodes are the elements. Every tree node except the root has a pointer to its parent, and the root stores the label of the set. Beware: These trees should not be confused with the trees in the forest within Kruskal's algorithm. Instead, they are formed and processed as follows. When Find(i) is called, we start in node i and walk to the root (where we find the label of the set), following the pointers. When Union(A, B) is called, then the root of the smaller tree is "adopted" as a new child by the root of the bigger tree. This works in $O(1)$ time, since only one new pointer must be created. By the same doubling argument as before, the depth of any node can increase at most $\log_2(k+1)$ times during the first k Union operations. As a consequence, every Find operation needs at most $O(\log k)$ time. Now we can perform every Union operation in $O(1)$ time and every Find operation in $O(\log k)$ time, where k is the total number of these data structure operations. In the really good implementation, however, trees are also modified upon Find(i) operations: Root r adopts all nodes on the path from i to r as new children! This "path compression" is not much more expensive than just walking the path, but it makes the paths for future Find operations much shorter. It can be shown that, with path compression, k Union and Find operations need together only $O(k)$ time (rather than $O(k \log k)$), subject to an extra factor that grows so extremely slowly that we can ignore it in practice. The time analysis is intricate and must be omitted here, but the structure itself is rather easy to implement.

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:

They are 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.

A Greedy Algorithm for Interval Partitioning

Let the subsets $X_1, X_2, X_3, \ldots$ initially be empty. We sort the intervals such that $s_1 < \ldots < s_n$, and we consider them in this order. (As opposed to the Interval Scheduling algorithm, they are sorted by their start points – this is intended!) We always put the current interval x into the subset X_i with the smallest possible index i. That is, we choose the smallest i such that x does not intersect any other interval in X_i.

Here we omit the details of an efficient implementation and come to the interesting point: Why is this greedy rule correct? In fact, optimality may be proved again by an exchange argument, but here we illustrate another nice proof technique: We give a simple bound for the value d to be optimized, and then we show that our solution attains this bound, hence it is optimal.

Specifically, let d be the maximum number of intervals I for which some point p exists which is contained in all these intervals ($\forall I : \ p \in I$). On the one hand, since these d intervals must be put in d distinct subsets, any solution needs at least d subsets, even an optimal solution. On the other hand, our greedy algorithm uses only d subsets: Whenever a new interval x starts, at most $d - 1$ earlier intervals can intersect x, because any such interval must contain the start point of x. Hence we can always put x in some of the first d subsets.

This proof is an example of **duality**, a principle that plays a central role in optimization. Given a 'primal" minimization problem, we set up a "dual" maximization problem, which is chosen in such a way that the maximum dual value is a lower bound on the primal values. Then, if some primal solution happens to attain this bound, this solution is optimal.

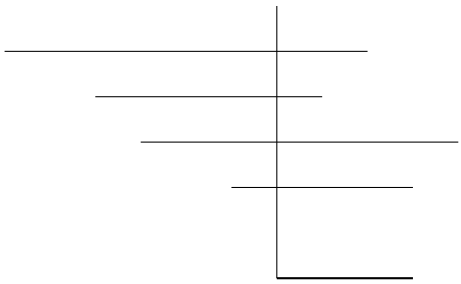


Figure 1: Illustration of the correctness proof for the Interval Partitioning greedy algorithm. Let $d = 5$. The current interval I (the thick line) has conflicts with at most 4 intervals that started earlier, because all these conflicting intervals contain I's start point, and at most 5 intervals can share a point. Hence we can put I in some of the 5 sets.

Algorithms. Lecture Notes 13

Space-Efficient Sequence Comparison

This section deals with an algorithm where dynamic programming and divide-and-conquer work nicely together. We also address the space complexity of a problem (unlike the rest of the course).

Suppose $m \leq n$. We have seen an algorithm that aligns two sequences $A = a_1 \dots a_n$ and $B = b_1 \dots b_m$ in $O(nm)$ time. Unfortunately, it also needs $O(nm)$ space, which can be prohibitive for applications in molecular biology where n, m are huge numbers. What can we do about that?

We may implement the dynamic programming algorithm in such a way that it requires only $O(m)$ space: For computing the values $OPT(i, j)$ we need only the previous row of the array of OPT values, but we can forget all earlier values.

But this gives us *only the score* $OPT(n, m)$ of a best alignment. If we are supposed to deliver an optimal alignment as well, we need (potentially) all $OPT(i, j)$ values for the backtracing procedure, since we do not know in advance the optimal path through the array. We could maintain the best alignments of prefixes along with the $OPT(i, j)$ values, but then we are back to $O(nm)$ space complexity.

The striking idea to overcome the space problem is to determine one entry (or "node") in the middle of the optimal path. We get it from the scores, which can be computed in small space by dynamic programming (as we have seen above). Once we know one node of the optimal path, we can split our problem instance in two independent instances and solve them recursively, one after another. Thus, everything happens in small memory space, while the divide-and-conquer structure ensures that we do not lose too much time. Below we describe the resulting algorithm in more detail.

Let $k \approx m/2$. We compute the scores (edit distanes) $OPT(j, k)$ for all j by dynamic programming, in $O(nm)$ time and $O(m)$ space. The same is done for the reversed sequences $a_n \dots a_1$ and $b_m \dots b_1$. The half sequence $b_1 \dots b_k$ *must* be aligned to $a_1 \dots a_j$, for some yet unknown j, and the other

half of B to the rest of A. After that splitting, the two optimal alignments are completely independent. In order to find the optimal cut-off point j, we can simply add the scores of these two alignments and pick an index j where the sum of scores is minimized. Clearly, the minimum sum is found in $O(n)$ time and space. Finally we divide B at position k, and we divide A at the optimal position j just determined, and we make two recursive calls to solve the sub-instances.

We never need more than $O(n)$ space simultaneuously. The time complexity is given by the recurrence $T(n, m) = 2T(n, m/2) + O(nm)$, since divide-and-conquer is done on a sequence B of length m, and $O(nm)$ time is still needed to compute the scores. Note that this recurrence has two variables. Without the argument n and without factor n in the last termn, we would have the standard recurrence $T(m) = 2T(m/2) + O(m)$ with solution $T(m) = O(m \log m)$. Our n can be treated as a "constant" factor that appears in every recursion level, thus we can immediately conclude that $T(n, m) = O(mn \log m)$. Actually, a slightly more careful analysis yields an $O(nm)$ time *and* $O(n)$ space bound.

Problem: Clustering with Maximum Spacing

A **clustering** of a set of (data) points is simply a partitioning into disjoint subsets of points, called **clusters**. Some distance function is defined between the points. The distance of two point sets A and B is the minimum distance of two points $a \in A$ and $b \in B$. The **spacing** of a clustering is the minimum distance of two clusters (or equivalently, the minimum distance of any two points from different clusters).

Given: a set of n points in some geometric space, and an integer $k < n$. The pairwise distances of points are known, or they can be easily computed from their coordinates.

Goal: Construct a clustering with k clusters and maximum spacing.

Motivations:

Clustering in general has many applications in data reduction, pattern recognition, classification, data mining, and related fields. Coordinates of points are often numerical features of objects. Every cluster shall consist of "similar" objects, whereas objects in different clusters shall be "dissimilar". However, we have to make these intuitive notions precise. There exist myriads of meaningful quality measures for clusterings, and each one gives rise to an algorithmic problem: to find a clustering that optimizes this quality measure.

Many clustering problems can be formulated as graph problems, where the data objects are nodes. For instance, Graph Coloring can be seen as a clustering problem: The desired number k of clusters is given, and every cluster must fulfill some "internal" criterion, namely, not to contain any pair of dissimilar nodes. Spacing is an "external" quality measure. It demands that any two clusters be far away from each other, while nothing is explicitly said about the inner structure of clusters.

Clustering with Maximum Spacing via MST

Kruskal's MST algorithm has a nice application and interpretation in the field of clustering problems. Suppose that the nodes of our graph are data points, and the edge costs are the distances. (The graph is complete, that is, all possible edges exist.) A clustering with maximum spacing (i.e., maximized minimum distance between any two clusters) can be found as follows: Do $n - k$ steps of Kruskal's algorithm and take the node sets of the so obtained k trees $T_1, \ldots, T_k$ as clusters.

We prove that the obtained spacing d is in fact optimal: Consider any partitioning into k clusters $U_1, \ldots, U_k$. There must exist two nodes p, q in some T_r that belong to different clusters there, say $p \in U_s$, $q \in U_t$. Due to the rule of Kruskal's algorithm, all edges on the path in T_r from p to q have cost at most d. But one of these edges joins two different "U clusters", hence the spacing of the other clustering can never exceed d.