Computationally Hard Problems

Administration

Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Fall 2016

Dates, Places and People

Course 02249 – 7.5 ECTS
Computationally Hard Problems
Lectures: Tuesday 8:15–10:00 in 303A/41
Exercise groups: Tuesday 10:15–12:00 in 306/holdområde stuen vest

Lecturer:

Carsten Witt, office: building 322, room 012, Tel. 45 25 37 22
e-mail: (use Piazza)

Literature

There are electronic lecture notes: version 2.5.

Download from the Campusnet group for the course.

If necessary, notes will be updated during the semester.

Comments, suggestions for improvement etc. are *highly welcome.*

Additional reading: books on complexity theory and randomized algorithms (see references in lecture notes)

Exercises, Office Hours

Exercise groups are conducted by

Andrea Dittadi (teaching assistant)

as well as myself and take place in building 306, holdområde stuen vest.

We encourage you to use Piazza
(`http://piazza.com/dtu.dk/fall2016/02249/home`) for discussion.
Please contact also the teachers through Piazza.

If you would like to talk to me personally: my office hours are on Fridays, 12:30–13:30.

Or contact me to make an appointment.

Exercises

There will be seven assignments and a little project (i. e., a larger homework) in between.

Every assignment consists of two parts: a **non**-mandatory and a **mandatory** one.

The mandatory exercises are graded on a scale from 0–2.

Only the six best results are counted, so one can get up to 12 points.

The project counts 6 points, so in total you can get 18 points. You have three weeks (excluding fall break) to work on the project.

The **mandatory** exercises (except the project assignment) have to be handed in **individually**.

Exercises

The assignments are made available as PDF on Campusnet (linked from Piazza) every Tuesday at 9.00.

The solutions to the mandatory assignments have to be handed in the Monday after at 21:00 the latest.

The solutions should contain your name and study number and date. If you discussed your solution with fellow students: state with whom in your solution.

Submit your solutions electronically via Campusnet ("Assignment/Opgaver" menu). Meet the deadline.

Exam and Entry Conditions

There is a written four-hour exam on Wednesday, 14.12.2016.

The grade is based on the exam and the mandatory assignments – though roughly 80 % for the exam.

We carry out an overall assessment (helhedsvurdering).

The final grade follows 7-step scale.

[12, 10, 7, 4, 2, 0, -3]

You will get some old exam problems during the course/exercises.

Remark on the Prerequisites

- The participants of this course come from different places.
- Their background with respect to Computer Science therefore may be different.
- The notation and terminology varies.
- There will some short "crash courses" to supplement missing items and find a common language.
- You will be asked to fill out a little questionnaire.

Remark on the Slides

- Some of these slides have been used before, some are used for first time and might contain errors.
- Slides are a teaching aid to support lecturing.
- Slides are NOT an aid for stand-alone learning.
- They are not always self-explanatory; one needs comments given in the lectures.
- Some topics will be presented by other means (e. g. blackboard) and are also relevant.

Computationally Hard Problems

Motivation, Definitions and Notation

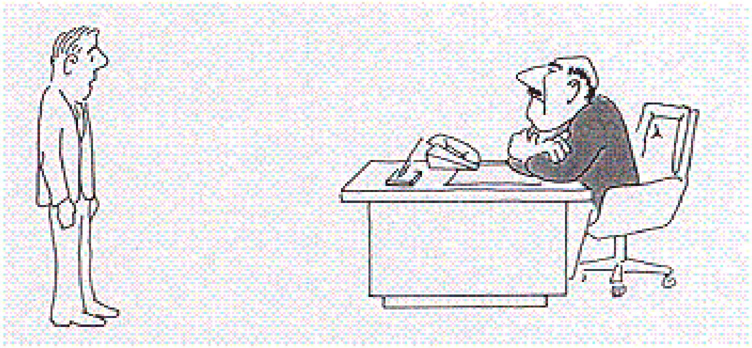
Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Fall 2016

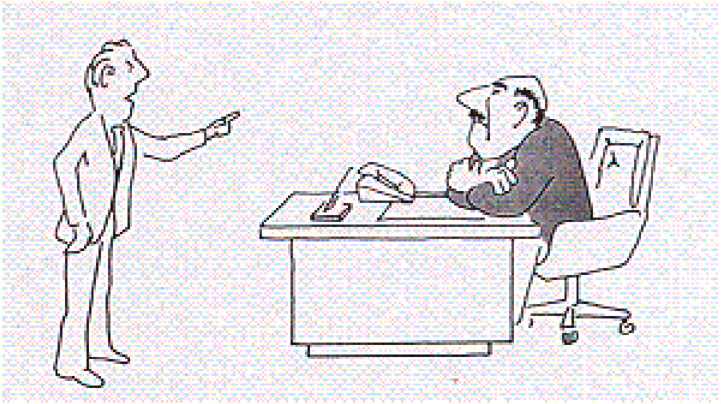
Motivation for this Course (1/4)

Example from the book by Garey and Johnson (1979):



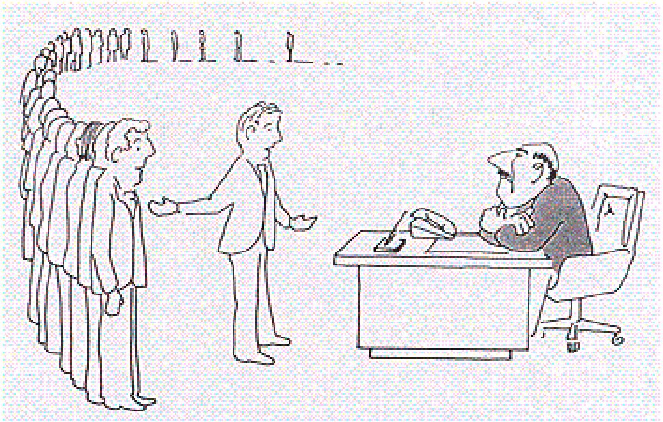
"I can't find an efficient algorithm *[a fast program]*, I guess I'm just too dumb."

Motivation for this Course (2/4)



"I can't find an efficient algorithm, because I can prove that no such algorithm is possible."

Motivation for this Course (3/4)



“I can’t find an efficient algorithm, but neither can all these famous people.”

Motivation for this Course (4/4)

This was the situation in late 1970s. Today we know how to attack hard problems by providing solutions

- that are provably very good but not necessarily perfect (approximation),
- that are provably very good or even perfect with a very high probability (randomization),
- that are typically good even though we cannot prove that (heuristics).

The course will not only enable you to prove hardness, but also show how to attack hard problems by the above means.

Course Content: More Details

This course is about the limits of efficient algorithms (programs) and the benefits of randomized algorithms; more precisely:

- General definitions and notation
- Some basics of statistics/probability theory
- Randomized algorithms complexity classes
- Hard problems
- Solving problems with randomized algorithms

What exactly is a problem?

Typical Problems (Described Informally)

Problems like these appear in every engineering area:

- Planning a fuel-efficient tour for a plane/ship/robot
- Cutting pieces out of a steel plate with minimal waste
- Optimizing the payload for a ship/rocket
- Designing a production process to minimize the time/cost
- Minimizing the number of tests to check all faults in a circuit
- Scheduling tasks on a computer (or other) system to minimize the used resources
- Minimizing the area of a circuit
- Checking whether a program is correct
- many thousands more

Means of Describing Things Formally

- The course deals with problems and their “hardness”: *computational complexity*.
- The complexity is the amount of computational resources needed to solve them.
- The problems are the input to algorithms (programs).
- They have to be described in a formal, unambiguous way.
- Such a description is given in form of *formal languages*. We briefly repeat/introduce the basic concepts.
- It is natural to expect that “larger problems take longer to solve.”
- Our formal description serves also to measure the *problem size*.

Formal Languages

- An *alphabet* is a **finite** set of symbols (also called *letters*). We often use the Greek letter Σ to denote alphabets.
- A *word* $\boldsymbol{w} = w_1 \cdots w_n$ over an alphabet Σ is a finite sequence of symbols of Σ (colloquially: a string).
- The *length* $|\boldsymbol{w}|$ of a word $\boldsymbol{w} = w_1 \cdots w_n$ is the number of symbols it contains: $|\boldsymbol{w}| = n$.
- By Σ^n we denote the set of all words over Σ of length exactly n, $n \in \mathbb{N}$. The set Σ^0 contains the *empty word* ε.
- By Σ^* we denote the set of all words over Σ, i.e., $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$.
- A *language* L over Σ is a collection of words over Σ, i.e., $L \subseteq \Sigma^*$.

Example

Let the alphabet be $\Sigma = \{a, b\}$.

Then

$$\Sigma^3 = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$$

The set $L_{=2b,3}$ of all words from Σ^3 which have exactly 2 b's is a finite language.

$$L_{=2b,3} = \{abb, bab, bba\}$$

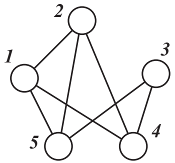
The set $L_{=2b}$ of all words from Σ^* which have exactly 2 b's is an infinite language.

$$L_{=2b} = \{bb, abb, bab, bba, aabb, abab, abba, baab, baba, bbaa, \\ aaabb, aabab, abaab, \dots\}$$

Example: Graphs

A language for undirected graphs.

We use (the right parts of) the rows of the adjacency matrix.



	1	2	3	4	5
1	0	1	0	1	1
2	1	0	0	1	1
3	0	0	0	1	1
4	1	1	1	0	0
5	1	1	1	0	0

Coding of G: 1011011110

The rows are recovered by taking $n-1$, $n-2$, $\dots$, 2, 1 letters.

$$L_{\text{graphs}} = \bigcup^{\infty} \{0,1\}^{n(n-1)/2} = \{\boldsymbol{w} \in \{0,1\}^k \mid \exists n\colon k = n(n-1)/2\}$$

Example: Clauses

Let $x_1, \dots, x_n$ be boolean variables (true/false as known from propositional logic)

The negation of x is denoted by $\overline{x}$ (others use $\neg x$).

A *clause* is an OR expression of variables or negated variables.

$$c = x_7 \vee x_3 \vee \overline{x}_{71}$$

To code a set of clauses use $+$ and $-$ to indicate (non-)negation and $0, \dots, 9$ to describe the index. The hash symbol $\#$ is separator.

We list clauses in increasing order (separated by $\#$) and write down the indices of their variables, predeceded by $+$ or $-$. Any word not matching this specification is not in the language.

The alphabet is $\Sigma_c = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \#, +, -\}$.

Example: Clauses

Consider the following set C with three clauses:

$$\begin{aligned} C &= \{c_1, c_2, c_3\} \\ c_1 &= \overline{x}_1 \vee \overline{x}_3 \vee x_5 \vee x_{42} \vee x_2 \\ c_2 &= x_2 \vee \overline{x}_4 \vee \overline{x}_5 \\ c_3 &= x_2 \vee \overline{x}_3 \vee x_5 \vee x_{42} \end{aligned}$$

This is then encoded by the word w:

$$w = \texttt{-1-3+5+42+2\#+2-4-5\#+2-3+5+42}$$

How To Write Down a Formal Language

We have seen two examples:

- $L_{\text{graphs}} = \bigcup_{n=0}^{\infty} \{0,1\}^{n(n-1)/2} = \left\{ \boldsymbol{w} \in \{0,1\}^k \mid \exists n \colon k = n(n-1)/2 \right\}$
- "We list clauses in an arbitrary but fixed order (separated by $\#$) and write down the indices of their variables, predeceded by $+$ or $-$. Any word not matching this specification is not in the language L_{clauses}."

The first one is formal and clear (?!). The second one is more informal. We could also have written a regular expression (if you don't know regular expresssions, don't care): $L_{\mathsf{clauses}} = ([\backslash + \mid -][1-9][0-9]^*)^+(\#([\backslash + \mid -][1-9][0-9]^*)^*)^*$.

Is that clearer?

Often, a unambiguous English description is preferable and recommended for solutions to exercises.

A Generic Problem: The Word Problem

Definition

Given a language L over some alphabet Σ, the *word problem* for L is the problem to decide for every word $w \in \Sigma^*$ whether $w \in L$.
An algorithm *solves* the word problem for L if given a word $w \in \Sigma^*$ it decides whether $w \in L$ or $w \notin L$.

Example

Consider the problem of deciding whether a graph has a triangle. Then we want to solve the word problem for the language L_{triag}:

$$\begin{aligned} L_{\mathrm{triag}} &= \{\boldsymbol{w} \in \{0,1\}^k \mid \exists n\colon k = n(n-1)/2 \\ &\qquad \wedge \text{ the graph coded by } \boldsymbol{w} \text{ has a triangle}\} \end{aligned}$$

Example: Solving the Word Problem

Consider the alphabet $\Sigma := \{0, 1\}$. Given a word $\boldsymbol{w} \in \Sigma^*$, decide whether $\boldsymbol{w} \in L_{\mathrm{triag}}$.

To do this:

- Read $\boldsymbol{w}$ from left to right to determine its length k.
- $n := 1$. While NOT $(k = n(n-1)/2$ or $n > k)$ Do $n := n + 1$.
  If $k \neq n(n-1)/2$, answer NO and STOP.
- Build upper triangle of adjacency matrix by taking out i successive elements from $\boldsymbol{w}$ to form the last i elements of row $n - i$, $i = n - 1$ down to 1. Make lower triangle symmetrical.
- Enumerate all triples $i < j < k$. Check if all three edges $\{i, j\}$, $\{j, k\}$ and $\{i, k\}$ exist for a triple. If so, YES, otherwise NO.

Similar to the solution of a typical (exam) exercise; formulation as above in simple, unambiguous English (no pseudocode) is preferred.

Size of an Input to a Problem

We already know the length of a word.

The definition of the size of a word/input (in the following denoted by $\boldsymbol{X}$) varies with problem under consideration. In any case the size of a word $\boldsymbol{X} \in \Sigma^*$ has to be proportional to the word length, i. e., to the number of symbols of Σ appearing in $\boldsymbol{X}$.

Definition

Let Σ be an alphabet and let $\boldsymbol{X} \in \Sigma^*$ be a word over Σ that is an input to some algorithm. Then we denote the *size* by $\|\boldsymbol{X}\|$.

Example: Size of a Word

Example

Consider the language L_{graphs} of graphs. A suitable measure of input size is the word length. This is exactly the possible number of edges in the graph represented by the input.

Note that there are then no words of size m in L_{graphs} if m is not of the form $k(k-1)/2$ for some $k \in \mathbb{N}$.

Example: Size of a Word

Example

Consider the language L_{clauses} clauses from the Example above. Again one can take the word length to be the input size: $\|\boldsymbol{X}\| = |\boldsymbol{X}|$.

The coding of an individual variable x_k requires $\lceil \log_{10}(k+1) \rceil$ letters.

If the number of variables is "somewhat small", e. g. if the largest number fits into a word of the computer's architecture, one can also define $\|\boldsymbol{X}\|$ by the total number of occurrences of all variables in the clauses, i. e., the number of plus and minus signs in $\boldsymbol{X}$.

Algorithms

Definition

An *algorithm* is a unambiguous, abstract description of a method for solving a problem or performing a task.
The *semantics* of the algorithms has to be clearly defined, i.e., its input-output behaviour has to be uniquely defined.

The terms unambiguous and abstract should be understood in the following way.

unambiguous	Anyone following the description will get the same result.
abstract	The description is independent of the specific environment in which the actions are performed.

Example

Example

Output: one loaf of white bread of 750 g

Input:

500	g	wheat flour
300	ml	water
30	g	yeast
5	g	salt
5	g	sugar

Algorithm: Mix yeast with 100 g flour, 100 ml water and the sugar and let rise for 10 min.
Mix this and all other ingredients and knead for 10 min.
Form a ball and let rise until the volume has doubled.
Knead again, form a loaf and let rise for 40 min.
Then bake in a pre-heated oven for 40 min at 220°C.

Complexity of an Algorithm

Definition

The *complexity of an algorithm* is its consumption of resources such as

- **Time** still the most important one.
- Memory
- Accesses to background storage
- Network usage
- Number of parallel processes
- Weighted combinations of some of the above
- Power
- ...

Measure of Time

- The time complexity of an algorithm cannot be measured in seconds
- because such absolute measurements are machine and date dependent.
- Instead one uses a measure that is machine independent (assuming a model of a machine, e. g., the Word RAM model):
- One unit of time is equivalent to one *atomic computational step* as, for example:
  - Assignment of "small" data types ($a \leftarrow b$)
  - Arithmetic operations on "small" numbers ($a + b$, $a * b$)
  - Comparison and branching (IF ($a < b$) THEN ... ELSE ...)
  - Reading an input bit, writing an output bit

Measure of Time

The following operations will **not** be considered as atomic computational steps:

- computation of trigonometric functions
- computing logarithms, exponentials
- arithmetic operations on large numbers; longer than one machine word
- copy-assignments of large data structures
- reading/writing a whole file

Running Time

For the discussion of the running time we use the following example: Consider words (strings) over the alphabet $\Sigma = \{a, b\}$.

Let $L_{\geq 2b} \subseteq \Sigma^*$ be the language of all strings containing at least 2 letters b.

We want to check whether a string $\boldsymbol{X} \in \Sigma^*$ is in $L_{\geq 2b}$, i.e., contains at least two b.

Algorithm Scan does this by scanning $\boldsymbol{X}$ from left to right and stopping at the second b saying "bingo", or "NO" when reaching the end of $\boldsymbol{X}$ without finding two b.

One computational step is checking a letter and moving one position right (can be done in constant time).

Running Time

Definition

Let A be an algorithm.

- The *running time* $T^A(\boldsymbol{X})$ of an algorithm A for a fixed input $\boldsymbol{X}$ is the number of atomic computational steps that A performs on input $\boldsymbol{X}$ until it stops
- If A does not stop on $\boldsymbol{X}$ then $T^A(\boldsymbol{X}) = \infty$.

Example:

$$T^{\text{Scan}}(\mathsf{bbaaaba}) = 2 \quad T^{\text{Scan}}(\mathsf{aabaaabbaab}) = 7$$
$$T^{\text{Scan}}(\mathsf{aaaaaaaaaaaaaaaaaaaaaaaaab}) = 26$$