Practicalities

- Code: 456314.0
- Previous knowledge
  - 456313.0 Algoritmer
  - Some course on Algorithms
  - Tuesdays and Thursdays, 13:15-15:00
  - Cobol (B3040) and Fortran (A3058)
- Components:
  - Lectures: 8 lectures (16h)
  - Exercises: 4 exercise sessions
  - Programming: 4 tasks
How to pass the exam

• The grade can be obtained 100% from passing the exam
• The grade can alternatively be obtained
  • 50% from passing the exam
  • 50% by programming 2 approximation algorithms and 2 randomized algorithms
    • These algorithms must be from those studied during the course
Material

  - A few copies in the ICT-library
  - Basically chapters 11 and 13; but first chapters 1, 2, 8, 9, 10 (briefly)

- Other

- [http://users.abo.fi/lpetre/ARA14/](http://users.abo.fi/lpetre/ARA14/)
Algorithms...

- The core of computer business
  - Computer science
  - Computer/software engineering
  - Information systems
- Some problems are simple
  - Simple methods to solve
  - Efficient (fast)
- Some problems are not simple
  - No known algorithm
Algorithms...

- The core of computer business
  - Computer science
  - Computer/software engineering
  - Information systems
- Some problems are simple
  - Simple methods to solve
  - Efficient (fast)
- Some problems are not simple
  - No known exact algorithm
  - Not efficient
Today

- Algorithm Revision
  - Algorithms for the **stable matching** problem
  - Five illustrative algorithm problems
  - Efficiency of algorithms
Stable matching problem

- 1962
  - David Gale and Lloyd Shapley -> mathematical economists
  - Could one design a job recruiting process that is self-enforcing?
- 1950s: National Resident Matching Program

Given
- Set of preferences among employers and applicants
- Can we assign applicants to employers so that for every employer E and every applicant A who is not scheduled to work for E, we have at least one of:
  - E prefers every one of its accepted applicants to A
  - A prefers the current situation over working at E
Matching Residents to Hospitals

**Goal.** Given a set of preferences among hospitals and medical school students, design a self-reinforcing admissions process.

**Unstable pair:** applicant $x$ and hospital $y$ make an unstable pair if:
- $x$ prefers $y$ to its assigned hospital.
- $y$ prefers $x$ to one of its admitted students.

** Stable assignment.** Assignment with no unstable pairs.
- Natural and desirable condition.
- Individual self-interest will prevent any applicant/hospital deal from being made.
Stable Matching Problem

**Goal.** Given \( n \) men and \( n \) women, find a "suitable" matching.
- Participants rate members of opposite sex.
- Each man lists women in order of preference from best to worst.
- Each woman lists men in order of preference from best to worst.

<table>
<thead>
<tr>
<th>Men's Preference Profile</th>
<th>Women's Preference Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Xavier</strong></td>
<td><strong>Amy</strong></td>
</tr>
<tr>
<td></td>
<td>Bertha</td>
</tr>
<tr>
<td><strong>Zeus</strong></td>
<td>Amy</td>
</tr>
<tr>
<td></td>
<td>Bertha</td>
</tr>
<tr>
<td></td>
<td>Clare</td>
</tr>
<tr>
<td><strong>Yancey</strong></td>
<td>Amy</td>
</tr>
<tr>
<td></td>
<td>Bertha</td>
</tr>
<tr>
<td></td>
<td>Clare</td>
</tr>
</tbody>
</table>

**Men's Preference Profile**

**Women's Preference Profile**
Stable Matching Problem

**Perfect matching:** everyone is matched monogamously.
- Each man gets exactly one woman.
- Each woman gets exactly one man.

**Stability:** no incentive for some pair of participants to undermine assignment by joint action.
- In matching $M$, an unmatched pair $m$-$w$ is **unstable** if man $m$ and woman $w$ prefer each other to current partners.
- Unstable pair $m$-$w$ could each improve by eloping.

**Stable matching:** perfect matching with no unstable pairs.

**Stable matching problem.** Given the preference lists of $n$ men and $n$ women, find a stable matching if one exists.
Questions

1. Does there exist a stable matching for every set of preference lists?
2. Given a set of preference lists, can we efficiently construct a stable matching if one exists?
**Stable Matching Problem**

**Q. Is assignment X-C, Y-B, Z-A stable?**

**Men's Preference Profile**

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xavier</td>
<td>Amy</td>
<td>Bertha</td>
<td>Clare</td>
</tr>
<tr>
<td>Yancey</td>
<td>Bertha</td>
<td>Amy</td>
<td>Clare</td>
</tr>
<tr>
<td>Zeus</td>
<td>Amy</td>
<td>Bertha</td>
<td>Clare</td>
</tr>
</tbody>
</table>

**Women's Preference Profile**

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>Yancey</td>
<td>Xavier</td>
<td>Zeus</td>
</tr>
<tr>
<td>Bertha</td>
<td>Xavier</td>
<td>Yancey</td>
<td>Zeus</td>
</tr>
<tr>
<td>Clare</td>
<td>Xavier</td>
<td>Yancey</td>
<td>Zeus</td>
</tr>
</tbody>
</table>
Stable Matching Problem

Q. Is assignment X-C, Y-B, Z-A stable?
A. No. Bertha and Xavier will hook up.

<table>
<thead>
<tr>
<th>Men's Preference Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1st</strong></td>
</tr>
<tr>
<td>Xavier</td>
</tr>
<tr>
<td>Yancey</td>
</tr>
<tr>
<td>Zeus</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Women's Preference Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1st</strong></td>
</tr>
<tr>
<td>Amy</td>
</tr>
<tr>
<td>Bertha</td>
</tr>
<tr>
<td>Clare</td>
</tr>
</tbody>
</table>
Stable Matching Problem

Q. Is assignment X-A, Y-B, Z-C stable?
A. Yes.

**Men’s Preference Profile**

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xavier</td>
<td>Amy</td>
<td>Bertha</td>
<td>Clare</td>
</tr>
<tr>
<td>Yancey</td>
<td>Bertha</td>
<td>Amy</td>
<td>Clare</td>
</tr>
<tr>
<td>Zeus</td>
<td>Amy</td>
<td>Bertha</td>
<td>Clare</td>
</tr>
</tbody>
</table>

**Women’s Preference Profile**

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>Yancey</td>
<td>Xavier</td>
<td>Zeus</td>
</tr>
<tr>
<td>Bertha</td>
<td>Xavier</td>
<td>Yancey</td>
<td>Zeus</td>
</tr>
<tr>
<td>Clare</td>
<td>Xavier</td>
<td>Yancey</td>
<td>Zeus</td>
</tr>
</tbody>
</table>
Propose-And-Reject Algorithm


Initialize each person to be free.
while (some man is free and hasn't proposed to every woman) {
    Choose such a man m
    w = 1st woman on m's list to whom m has not yet proposed
    if (w is free)
        assign m and w to be engaged
    else if (w prefers m to her fiancé m')
        assign m and w to be engaged, and m' to be free
    else
        w rejects m
}
Proof of Correctness: Termination

**Observation 1.** Men propose to women in decreasing order of preference.

**Observation 2.** Once a woman is matched, she never becomes unmatched; she only "trades up."

**Claim.** Algorithm terminates after at most $n^2$ iterations of while loop.

**Pf.** Each time through the while loop a man proposes to a new woman. There are only $n^2$ possible proposals. •

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
</tr>
</thead>
<tbody>
<tr>
<td>Victor</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>Wyatt</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>E</td>
</tr>
<tr>
<td>Xavier</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>Yancey</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>Zeus</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>V</td>
</tr>
<tr>
<td>Bertha</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>V</td>
<td>W</td>
</tr>
<tr>
<td>Clare</td>
<td>Y</td>
<td>Z</td>
<td>V</td>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>Diane</td>
<td>Z</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>Erika</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
</tbody>
</table>

$n(n-1) + 1$ proposals required
Proof of Correctness: Perfection

Claim. All men and women get matched.

Pf. (by contradiction)

- Suppose, for sake of contradiction, that Zeus is not matched upon termination of algorithm.
- Then some woman, say Amy, is not matched upon termination.
- By Observation 2, Amy was never proposed to.
- But, Zeus proposes to everyone, since he ends up unmatched. □
Proof of Correctness: Stability

Claim. No unstable pairs.
Pf. (by contradiction)

- Suppose A-Z is an unstable pair: each prefers each other to partner in Gale-Shapley matching $S^*$.

- **Case 1**: Z never proposed to A.
  $\Rightarrow$ Z prefers his GS partner to A.
  $\Rightarrow$ A-Z is stable.

- **Case 2**: Z proposed to A.
  $\Rightarrow$ A rejected Z (right away or later)
  $\Rightarrow$ A prefers her GS partner to Z.
  $\Rightarrow$ A-Z is stable.

- In either case A-Z is stable, a contradiction. •
Summary

Stable matching problem. Given n men and n women, and their preferences, find a stable matching if one exists.

Gale-Shapley algorithm. Guarantees to find a stable matching for any problem instance.

Q. If there are multiple stable matchings, which one does GS find?
Stable matching problem

- Enough precision to
  - ask concrete questions
  - start thinking about an algorithm to solve the problem
- Design algorithm for problem
- Analyze algorithm
  - Correctness
  - Bound on running time
- Fundamental design techniques
Five representative problems

- Interval scheduling
- Weighted interval scheduling
- Bipartite matching
- Independent set
- Competitive facility location
Interval Scheduling

**Input.** Set of jobs with start times and finish times.

**Goal.** Find maximum cardinality subset of mutually compatible jobs.

jobs don't overlap
Weighted Interval Scheduling

**Input.** Set of jobs with start times, finish times, and weights.

**Goal.** Find maximum weight subset of mutually compatible jobs.
Bipartite Matching

**Input.** Bipartite graph.

**Goal.** Find *maximum cardinality* matching.

![Diagram of bipartite matching](image)
Independent Set

Input. Graph.
Goal. Find maximum cardinality independent set.

subset of nodes such that no two joined by an edge
Competitive Facility Location

**Input.** Graph with weight on each node.

**Game.** Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.

**Goal.** Select a maximum weight subset of nodes.

Second player can guarantee 20, but not 25.
Five Representative Problems

Variations on a theme: independent set.

Interval scheduling: \( n \log n \) greedy algorithm.
Weighted interval scheduling: \( n \log n \) dynamic programming algorithm.
Bipartite matching: \( n^k \) max-flow based algorithm.
Independent set: NP-complete.
Competitive facility location: PSPACE-complete.
Algorithm analysis

- How do resource requirements change when input size increases?
  - Time, space
  - Notational machinery
- Problems of **discrete** nature
  - Implicit searching over large space of possibilities
  - **Goal**: efficiently find solution satisfying conditions
- **Focus on** running time
Algorithm efficiency

1. An algorithm is efficient if, when implemented, it runs quickly on real-input instances

Problems with this
- Where we implement an algorithm
- How well we implement an algorithm

BETTER definition
- Platform-independent
- Instance-independent
- Predictive value with respect to increasing input size
Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size $N$.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size $N$.

- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.
Algorithm efficiency

1. An algorithm is efficient if, when implemented, it runs quickly on real-input instances

2. An algorithm is efficient if it achieves a better worst-case performance, at an analytical level, then brute-force search

→ Brute-force search provides no insight into the structure of the problem we are studying!
→ Definition somewhat vague
→ ”better performance”?
Polynomial-Time

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.
- Typically takes $2^N$ time or worse for inputs of size $N$.
- Unacceptable in practice.

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor $C$.

There exists constants $c > 0$ and $d > 0$ such that on every input of size $N$, its running time is bounded by $c N^d$ steps.

Def. An algorithm is poly-time if the above scaling property holds.

Choose $C = 2^d$
Worst-Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Exceptions.

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

\[ \text{simplex method} \]
Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>$n = 30$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>$n = 50$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 10,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100,000$</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000,000$</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>
More on the definition of efficiency in terms of poly-time

- This definition is negatable: we can say *when there is no efficient algorithm for a particular problem*
- Previous definitions were subjective
  - First definition turned efficiency into a moving target
  - The poly-time definition is more absolute
- Promotes the idea that problems have an intrinsic level of computational tractability
  - Some admit efficient solutions, some do not
Asymptotic Order of Growth

Basic assumption: an algorithm’s worst-case running time on inputs of size $n$ grows at a rate at most proportional to some function $f(n)$.

Upper bounds. $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.
Properties

Transitivity.

- If \( f = O(g) \) and \( g = O(h) \) then \( f = O(h) \).
- If \( f = \Omega(g) \) and \( g = \Omega(h) \) then \( f = \Omega(h) \).
- If \( f = \Theta(g) \) and \( g = \Theta(h) \) then \( f = \Theta(h) \).

Additivity.

- If \( f = O(h) \) and \( g = O(h) \) then \( f + g = O(h) \).
- If \( f = \Omega(h) \) and \( g = \Omega(h) \) then \( f + g = \Omega(h) \).
- If \( f = \Theta(h) \) and \( g = O(h) \) then \( f + g = \Theta(h) \).
Asymptotic Bounds for Some Common Functions

**Polynomials.** $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

**Polynomial time.** Running time is $O(n^d)$ for some constant $d$ independent of the input size $n$.

**Logarithms.** $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 0$.

- can avoid specifying the base

**Logarithms.** For every $x > 0$, $\log n = O(n^x)$.

- log grows slower than every polynomial

**Exponentials.** For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$.

- every exponential grows faster than every polynomial
Linear Time: $O(n)$

**Linear time.** Running time is at most a constant factor times the size of the input.

**Computing the maximum.** Compute maximum of $n$ numbers $a_1, \ldots, a_n$.

```plaintext
max ← a_1
for i = 2 to n {
    if (a_i > max)
        max ← a_i
}
```
Linear Time: $O(n)$

**Merge.** Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.

Claim. Merging two lists of size $n$ takes $O(n)$ time.

Pf. After each comparison, the length of output list increases by 1.

```plaintext
i = 1, j = 1
while (both lists are nonempty) {
    if ($a_i \leq b_j$) append $a_i$ to output list and increment $i$
    else append $b_j$ to output list and increment $j$
}
append remainder of nonempty list to output list
```
O(n log n) Time

O(n log n) time. Arises in divide-and-conquer algorithms. Also referred to as linearithmic time.

Sorting. Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.

Largest empty interval. Given n time-stamps \(x_1, \ldots, x_n\) on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.
Quadratic Time: \(O(n^2)\)

- Quadratic time. Enumerate all pairs of elements.

- Closest pair of points. Given a list of \(n\) points in the plane \((x_1, y_1), \ldots, (x_n, y_n)\), find the pair that is closest.

- \(O(n^2)\) solution. Try all pairs of points.

\[
\begin{align*}
\min & \leftarrow (x_1 - x_2)^2 + (y_1 - y_2)^2 \\
\text{for} \ i &= 1 \ \text{to} \ n \ \{ \\
\quad & \text{for} \ j = i+1 \ \text{to} \ n \ \{ \\
\quad & \quad d \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2 \\
\quad & \quad \text{if} \ (d < \min) \\
\quad & \quad \quad \min \leftarrow d \\
\quad & \} \\
\} \\
\end{align*}
\]

don't need to take square roots

- Remark. \(\Omega(n^2)\) seems inevitable, but this is just an illusion.
Cubic Time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given $n$ sets $S_1, \ldots, S_n$ each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pairs of sets, determine if they are disjoint.

```plaintext
foreach set $S_i$ {
    foreach other set $S_j$ {
        foreach element $p$ of $S_i$ {
            determine whether $p$ also belongs to $S_j$
        }
        if (no element of $S_i$ belongs to $S_j$)
        report that $S_i$ and $S_j$ are disjoint
    }
}
```
Polynomial Time: $O(n^k)$ Time

Independent set of size $k$. Given a graph, are there $k$ nodes such that no two are joined by an edge?

$k$ is a constant

$O(n^k)$ solution. Enumerate all subsets of $k$ nodes.

```
foreach subset S of k nodes {
    check whether S is an independent set
    if (S is an independent set)
        report S is an independent set
}
```

- Check whether $S$ is an independent set = $O(k^2)$.
- Number of $k$ element subsets $= \binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \leq \frac{n^k}{k!}$
- $O(k^2 \cdot n^k / k!) = O(n^k)$.

poly-time for $k=17$, but not practical
Exponential Time

Independent set. Given a graph, what is maximum size of an independent set?

O(n^2 2^n) solution. Enumerate all subsets.

S* ← φ
foreach subset S of nodes {
    check whether S in an independent set
    if (S is largest independent set seen so far)
        update S* ← S
}

}
Summing up

- Finding algorithms for practical problems
  - Depends on the problem
  - The efficiency of the algorithm varies

- Next time 21.1.2014
  - There is (some) hope for NP-completeness

- Following time 23.1.2014
  - Exercise set 1: Solve the following exercises from the course book: problems 1, 2, and 4 from Chapter 1; problems 1 and 2 from Chapter 2 and problem 6 from Chapter 11.