# Algorithmen für schwierige Probleme

Britta Dorn

**Wintersemester 2011/12**

20. Oktober 2011

# Komplexitätbasics

## P

Klasse von Entscheidungsproblemen ("Gibt es...?"), die in polynomieller Laufzeit von einer deterministischen Turingmaschine entschieden werden können.

Gilt als Klasse der praktisch lösbaren Probleme (auch wenn das Polynom furchtbar ist).

Problem in polynomieller Zeit lösbar: $\exists$ Algorithmus, der es in Zeit löst, die höchstens polynomiell mit der Größe der Eingabe ($n$, z.B. Größe des Graphs, also Anzahl Knoten) des Problems wächst, d.h.: $\exists c$, so dass Laufzeit $\mathcal{O}(n^c)$.

**Beispiele:**

- Sortieren: $\mathcal{O}(n^2)$, $\mathcal{O}(n \log n)$
- Anzahl Kanten eines Graphen: $\mathcal{O}(n^2)$
- Matrixmultiplikation: $\mathcal{O}(n^3)$
- Prüfen, ob gegebene Zahl eine Primzahl ist: $\mathcal{O}(n^{12})$

# Komplexitätbasics

## NP

Klasse von Entscheidungsproblemen, die in **polynomieller** Laufzeit von einer **nicht**deterministischen Turingmaschine entschieden werden können.

NP = nichtdeterministisch polynomielle Zeit!!

Einfacher: Probleme, für die gilt: falls man die Lösung hat, kann man in polynomieller Zeit prüfen, ob sie stimmt.
Woher hat man die Lösung?
$\hookrightarrow$ von einer nichtdeterministischen Turingmaschine (schlau): diese rät die Lösung und prüft dann deterministisch nach (in Polynomzeit), ob sie stimmt.

Klar: $P \subseteq NP$, Bild

# Komplexitätbasics

## NP

Klasse von Entscheidungsproblemen, die in **polynomieller** Laufzeit von einer **nicht**deterministischen Turingmaschine entschieden werden können.

Aber: bis heute unklar, ob man für Probleme in $NP \setminus P$ die Lösung in Polynomzeit finden kann, wenn man keine nichtdet. Turingmaschine ist (nicht gut raten kann).
Alle bekannten Algorithmen für diese Probleme haben exponentielle Laufzeit ($\mathcal{O}(c^n)$).

Offen: Gilt $P = NP$? 1 Mio US$, Milleniumsproblem

Achtung: NP ist nicht die Klasse der nicht in Polynomzeit lösbaren Probleme!!

# Komplexitätbasics

## NP-schwer

Probleme, die mindestens so schwer sind wie jedes beliebige Problem in NP.

Wie vergleicht man die Schwierigkeit von Problemen? Durch eine Polynomzeitreduktion.

Vergleiche unbekanntes Problem U mit Problem L, von dem wir wissen, dass es schwer ist. Zeige, dass sich L so umschreiben lässt, dass es aussieht wie U. Wenn man dann U lösen würde, hätte man automatisch L gelöst, und das war ja schwer! Also ist U mindestens so schwer wie L.

# Komplexitätbasics

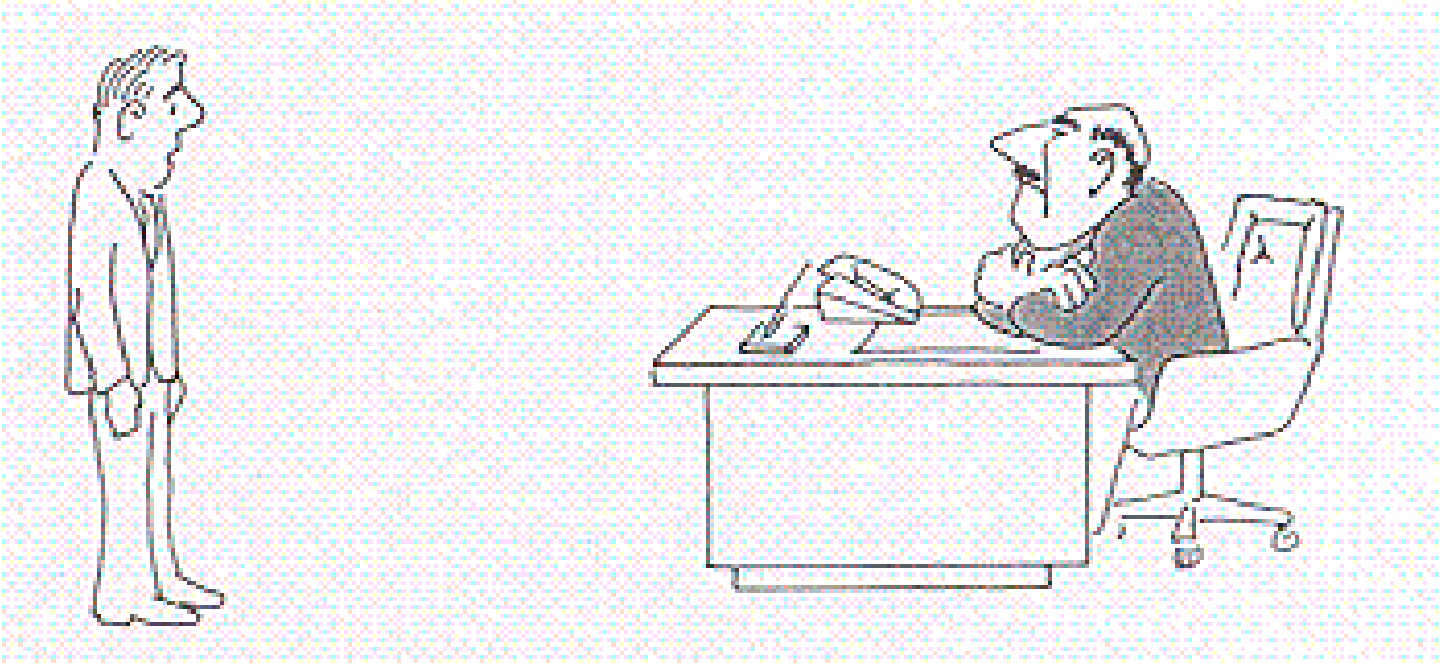
## NP-vollständig

Probleme, die NP-schwer sind und in NP liegen.

Warum ist das wichtig?

- Realität: kennen
- Milleniumsproblem, Allgemeinwissen
- Job: Comic

# NP-Vollständigkeit

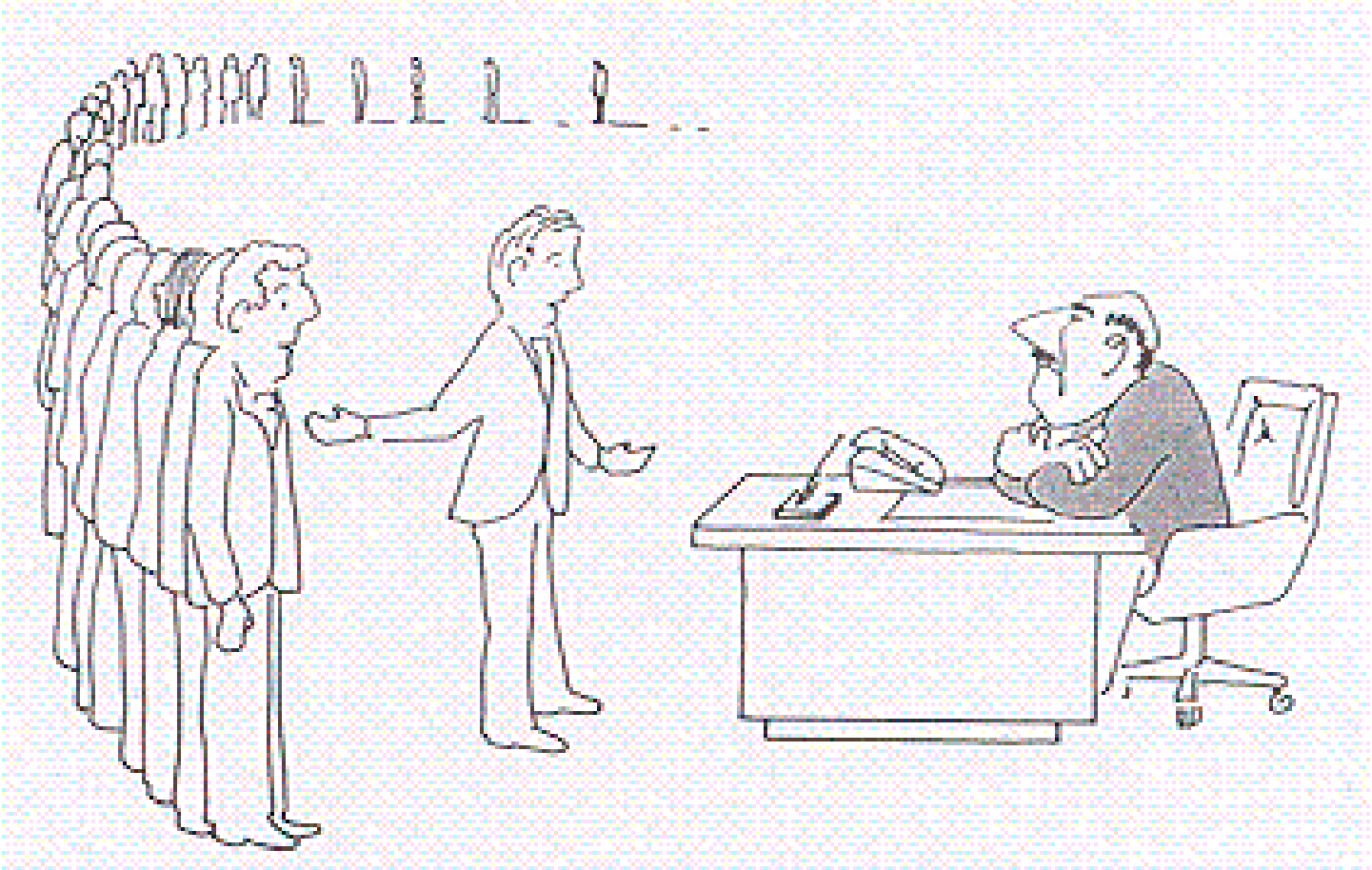


“I can’t find an efficient algorithm, I guess I’m just too dumb.”

# NP-Vollständigkeit

“I can’t find an efficient algorithm, because no such algorithm is possible!”

# NP-Vollständigkeit



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

# 1.2 Beispiel: ML

ML — funktionale Programmiersprache, für die es relativ effiziente Compiler gibt.
Aufgabe der Compiler: Verträglichkeit von Typdeklarationen prüfen.

**Problem ist EXP-vollständig**
**Aber: Praxis?!? Es funktionert trotzdem!**

## Grund

Verschachtelungstiefe $k$ (nesting depth) der Typdeklarationen.
($k \leq 5$, wenn Programm von einem Menschen geschrieben wurde)
$\rightarrow$ **Parametrisierte Algorithmen!**

# Kontrolle zur Definition 1.4

Welche der folgenden Laufzeiten lassen darauf schließen, dass die entsprechenden Probleme fixed-parameter tractable sind?

1. $O(2^k \cdot n)$
2. $O(2^k \cdot n^{20})$
3. $O(n^2)$
4. $O(n^{20})$
5. $O(2^k \cdot 2^n)$
6. $O(1.5^k \cdot n^{1.5})$
7. $O(1.5^k \cdot 1.5^n)$
8. $O(n^k)$
9. $O(2^{nk})$
10. $O(2^k \cdot n^k)$

# Lösung

# Kontrolle zur Definition 1.4

Welche der folgenden Laufzeiten lassen darauf schließen, dass die entsprechenden Probleme fixed-parameter tractable sind?

1. $O(2^k \cdot n)$
2. $O(2^k \cdot n^{20})$
3. $O(n^2)$
4. $O(n^{20})$
5. $O(2^k \cdot 2^n)$
6. $O(1.5^k \cdot n^{1.5})$
7. $O(1.5^k \cdot 1.5^n)$
8. $O(n^k)$
9. $O(2^{nk})$
10. $O(2^k \cdot n^k)$

# Datenreduktion und Problemkerne