

Informatik I: Einführung in die Programmierung

25. Laufzeitanalyse von Algorithmen

Albert-Ludwigs-Universität Freiburg



Bernhard Nebel
24.01.2017

1 Motivation



- Motivation
- Laufzeit von Algorithmen
- O-Notation
- Bestimmung der asymptotischen Laufzeit
- Skalierbarkeit
- Komplexitätstheorie
- Quadratische Falle
- Zusammenfassung

24.01.2017

B. Nebel – Info I

3 / 42

Motivation



- Wir haben Werkzeuge kennen gelernt, mit den denen man Laufzeit-Flaschenhälse in Programmen identifizieren kann.
- Wir haben auch ein paar Ideen, wie man die Laufzeit verbessern kann.
- Dies ist aber im wesentlichen auf der Ebene des **konkreten Programms**.
- Wenn der dem Programm zu Grunde liegende **Algorithmus** schlecht (ineffizient) ist, dann bringen kleine Laufzeitverbesserungen wenig.
- In der Informatik **argumentiert** man deshalb gerne auf der Ebene von **Algorithmen**.
- Wie gut **skaliert** ein Algorithmus: Wie stark **wächst** die Laufzeit (oder der Speicherplatzbedarf) mit der Größe der Eingabe?

- Motivation
- Laufzeit von Algorithmen
- O-Notation
- Bestimmung der asymptotischen Laufzeit
- Skalierbarkeit
- Komplexitätstheorie
- Quadratische Falle
- Zusammenfassung

24.01.2017

B. Nebel – Info I

4 / 42

2 Laufzeit von Algorithmen



- Motivation
- Laufzeit von Algorithmen
- O-Notation
- Bestimmung der asymptotischen Laufzeit
- Skalierbarkeit
- Komplexitätstheorie
- Quadratische Falle
- Zusammenfassung

24.01.2017

B. Nebel – Info I

6 / 42

Laufzeit von Algorithmen



- Wie misst man die Laufzeit von Algorithmen?
- Man identifiziert die (etwa gleich teuren) Grundoperationen (z.B. Vergleiche, arithmetische Operationen, Zuweisungen usw.) und bestimmt, wie häufig sie bei der Ausführung des Algorithmus A bei einer bestimmten Eingabe x ausgeführt werden.
- Dies sei die (abstrakte) Laufzeit von A auf x : $T_A(x)$.
- Darauf basierend kann man über alle Eingaben der Größe n gehen und die Laufzeit für die Größe n im **besten**, im **schlechtesten** und im **mittleren** Fall bestimmen:
 - **Bester Fall**: $T_A^b(n) = \min\{T_A(x) : |x| = n, x \text{ Eingabe für } A\}$
 - **Schlechtester Fall**:
 $T_A^w(n) = \max\{T_A(x) : |x| = n, x \text{ Eingabe für } A\}$
 - **Mittlerer Fall**: Sei $q_n(x)$ die Wahrscheinlichkeit, dass x unter den Eingaben der Länge n auftritt:
 $T_{A,q_n}^a(n) = \sum_{|x|=n, x \text{ Eingabe für } A} T_A(x) q_n(x)$

Motivation
Laufzeit von Algorithmen
O-Notation
Bestimmung der asymptotischen Laufzeit
Skalierbarkeit
Komplexitätstheorie
Quadratische Falle
Zusammenfassung

Beispiel: Suche in einer Liste



Wir wollen feststellen, ob in einer Liste von n Elementen ein bestimmtes Element vorhanden ist. Dies können wir durch folgenden Algorithmus (formuliert in Python) erreichen:

```
def search(e1, li):  
    for e in li:  
        if e == e1: return True  
    return False
```

Ist ein Schleifendurchlauf, ein Test und Rückgabe jeweils eine Operation mit den Zeitkosten 1, dann können wir folgende Laufzeiten konstatieren:

- **Bester Fall**: $T_A^b(n) = 3$ (gesuchtes Element an erster Stelle)
- **Schlechtester Fall**: $T_A^w(n) = 2n + 1$
- **Mittlerer Fall**: Falls $m > n$ mögliche Eingaben für das element-Argument möglich sind und diese gleichverteilt sind, dann gilt: $T_{A,q_n}^a(n) = \frac{(m-n)}{m} \cdot (2n + 1) + \frac{n}{m} \cdot \sum_{i=1}^n \frac{1}{n} \cdot (2i + 1)$.

Motivation
Laufzeit von Algorithmen
O-Notation
Bestimmung der asymptotischen Laufzeit
Skalierbarkeit
Komplexitätstheorie
Quadratische Falle
Zusammenfassung

Laufzeitwachstum



- Wir sehen, dass die Laufzeit im schlechtesten und mittleren Fall **linear** mit der Größe der Eingabe wächst.
- Hier ist es auch ganz unerheblich, ob z.B. ein return-Statement mehr Zeitkosten als ein Vergleich benötigt. Die echten Operatorkosten sind weitgehend egal.
- Für das Laufzeitwachstum (man spricht auch vom **asymptotischen Laufzeitverhalten**) sind i.W. die Anzahl der Schleifendurchläufe entscheidend.
- Man betrachtet dabei meist den **schlechtesten Fall**, da er einfach zu bestimmen ist und eine Garantie abgibt.
- Der **mittlere Fall** ist meist nur schwierig zu bestimmen und man benötigt viele Annahmen.
- Der **beste Fall** ist meist nicht sehr aussagekräftig.

Motivation
Laufzeit von Algorithmen
O-Notation
Bestimmung der asymptotischen Laufzeit
Skalierbarkeit
Komplexitätstheorie
Quadratische Falle
Zusammenfassung

Ein besserer Algorithmus?



Wenn wir feststellen wollen, ob ein Element in einer Liste vorhanden ist, geht das auch so:

```
def fast_search(e1, li):  
    if e1 in li: return True  
    return False
```

- Wenn wir annehmen, dass der `in`-Test nur Zeitkosten 1 hat, dann wäre unser Test im schlechtesten und mittleren Fall tatsächlich schneller.
- Das stimmt aber **nicht**, wenn es sich um Listen handelt!
- Auch Python muss die Liste von vorne nach hinten durchsuchen und jedes einzelne Element anschauen, macht dies aber schneller, z.B. in Zeit $0.1n$ statt $2n + 1$.
- Python (und viele andere Sprachen) enthalten Operationen, deren Zeitkosten durchaus nicht konstant sind, sondern z.B. linear von den Daten abhängen.

Motivation
Laufzeit von Algorithmen
O-Notation
Bestimmung der asymptotischen Laufzeit
Skalierbarkeit
Komplexitätstheorie
Quadratische Falle
Zusammenfassung

Suche in sortierten Listen (1)



- Nehmen wir an, dass die Liste sortiert ist, so gibt es einen effizienteren Suchalgorithmus: Die **binäre Suche**.
- Wir gehen ähnlich wie bei einer Suche im Telefonbuch vor:
 - 1 Wir betrachten das ganze Buch als **interessant**.
 - 2 Wir wählen im **interessanten Bereich** die **mittlere Seite** und schauen ob der gesuchte Name da steht. Falls ja, sind wir fertig.
 - 3 Falls der Name später in der Lexikon-Ordnung kommt, dann konzentrieren wir uns auf die **hintere Hälfte**.
 - 4 Ansonsten auf die **vordere Hälfte**.
 - 5 Die neue ausgewählte Hälfte ist unser **neuer interessanter Bereich** und wir machen mit Schritt 2 weiter.
- Wie schnell können wir in 2^n Seiten feststellen, ob ein Name vorhanden ist (im schlechtesten Fall?)

Motivation
Laufzeit von Algorithmen
O-Notation
Bestimmung der asymptotischen Laufzeit
Skalierbarkeit
Komplexitätstheorie
Quadratische Falle
Zusammenfassung

Suche in sortierten Listen (2)



- Wir haben zwei Variablen `left` und `right`, die uns den **interessanten Bereich** beschreiben.
 - Die Mitte ist jeweils $(left+right)//2$.
 - Wenn der interessante Bereich leer ist ($left > right$), dann ist das Element nicht vorhanden.
- ```
def bin_search(e1, sli):
 left, right = 0, len(sli) - 1
 while left <= right:
 mid = (left+right)//2
 if sli[mid] < e1: left = mid + 1
 elif sli[mid] > e1: right = mid - 1
 else: return True
 return False
```
- Wie viele Schleifendurchläufe brauchen wir hier?

Motivation  
Laufzeit von Algorithmen  
O-Notation  
Bestimmung der asymptotischen Laufzeit  
Skalierbarkeit  
Komplexitätstheorie  
Quadratische Falle  
Zusammenfassung

## Analyse der binären Suche



- Bei jedem **Schleifendurchlauf** wird der interessierende Bereich **halbiert**, ggfs. plus eins. Da  $mid + 1$  oder  $-1$ , ist es immer höchstens die Hälfte.
- D.h. wir haben maximal  $\lceil \log_2 n \rceil$  **Schleifendurchläufe**.
- In jedem Schleifendurchlauf haben wir maximal 2 Zuweisungen, 3 Vergleiche, eine Addition, und eine Division. Seien die Zeitkosten dafür jeweils 1. Dann sind die Schleifenkosten 7 Einheiten.
- Dazu kommen 2 Zuweisungen, eine Subtraktion, eine Längenbestimmung und zum Schluss ein `return`-Statement.
- Zeitkosten im **schlechtesten Fall**:  $5 + 7 \lceil \log_2 n \rceil$ .

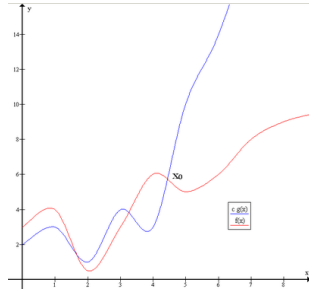
Motivation  
Laufzeit von Algorithmen  
O-Notation  
Bestimmung der asymptotischen Laufzeit  
Skalierbarkeit  
Komplexitätstheorie  
Quadratische Falle  
Zusammenfassung

## 3 Die O-Notation



Motivation  
Laufzeit von Algorithmen  
O-Notation  
Bestimmung der asymptotischen Laufzeit  
Skalierbarkeit  
Komplexitätstheorie  
Quadratische Falle  
Zusammenfassung

- Mit  $O(g)$  bezeichnet man die Menge von Funktionen  $f$ , für die gilt:  
 $\exists c \in \mathbf{R}^+, x_0 \in \mathbf{R}^+, \forall x > x_0 : f(x) \leq cg(x)$ .
- D.h.  $O(g)$  umfasst alle Funktionen  $f$ , die nicht schneller wachsen als  $g$  (wenn man konstante Faktoren ignoriert und endliche Anfangsstücke vernachlässigt).



Motivation  
 Laufzeit von Algorithmen  
 O-Notation  
 Bestimmung der asymptotischen Laufzeit  
 Skalierbarkeit  
 Komplexitätstheorie  
 Quadratische Falle  
 Zusammenfassung

- Beispiel: Für  $f(n) = n^2$  und  $g(n) = 25n$  gilt:
- $g \in O(f)$ , da
  - für  $c = 25$  und  $n_0 = 1: \forall n > n_0 : g(n) \leq cf(n)$ , da
  - $25n \leq 25n^2$  für alle  $n > 1$ ;
- $f \notin O(g)$ :
  - Wir nehmen an, dass  $f \in O(g)$  gilt.
  - Seien  $c$  und  $n_0$  so gewählt, dass die Bedingung  $\forall n > n_0 : f(n) \leq cg(n)$ , erfüllt ist, also gilt:
  - $n^2 \leq 25cn$  für alle  $n > n_0$ ;
  - Wähle  $n_1 = 25c + 1$ ; dann gilt aber für alle  $n > n_1$ :  $n^2 > 25cn$ , was ein Widerspruch ist.
  - Unsere Annahme  $f \in O(g)$  muss also falsch sein, d.h.  $f \notin O(g)$ .

Motivation  
 Laufzeit von Algorithmen  
 O-Notation  
 Bestimmung der asymptotischen Laufzeit  
 Skalierbarkeit  
 Komplexitätstheorie  
 Quadratische Falle  
 Zusammenfassung

- Man schreibt oft  $f = O(g)$ , meint aber  $f \in O(g)$ .
- Insbesondere folgt aus  $f = O(g)$  nicht  $O(g) = f$ !
- Statt „ $O(f)$  mit  $f(n) = n^2 + 2n + 4$ “ schreibt man  $O(n^2 + 2n + 4)$ .
- Einfache Regeln:
  - $f = O(f)$  (= bedeutet  $\in$ )
  - $O(O(f)) = O(f)$  (= bedeutet hier und im weiteren  $\subseteq$ )
  - $O(kf) = O(f)$  für eine Konstante  $k > 0$
  - $O(k + f) = O(f)$  für eine Konstante  $k \geq 0$
  - Additionsregel:  $O(f) + O(g) = O(\max\{f, g\})$
  - Multiplikationsregel:  $O(f) \cdot O(g) = O(f \cdot g)$

Motivation  
 Laufzeit von Algorithmen  
 O-Notation  
 Bestimmung der asymptotischen Laufzeit  
 Skalierbarkeit  
 Komplexitätstheorie  
 Quadratische Falle  
 Zusammenfassung

- Additionsregel:  $O(f) + O(g) = O(\max\{f, g\})$ .
- Mit  $O(f) + O(g)$  ist gemeint: Die Klasse aller Funktionen  $f' + g'$  mit  $f' \in O(f)$  und  $g' \in O(g)$ .
- Sei also  $f' \in O(f)$  und  $g' \in O(g)$ .
- D.h. es ex.  $c_1, c_2, n_1$  und  $n_2$  mit:  
 $f'(n) \leq c_1 \cdot f(n)$  für alle  $n \geq n_1$  und  
 $g'(n) \leq c_2 \cdot g(n)$  für alle  $n \geq n_2$ .
- Setze  $n_0 = \max\{n_1, n_2\}$  und  $c = c_1 + c_2$ .
- Dann gilt offensichtlich:  $f'(n) + g'(n) \leq c \cdot \max\{f(n), g(n)\}$  für alle  $n \geq n_0$ .
- Die Additionsregel ist relevant für die **Hintereinanderausführung** von Anweisungen in Programmen.

Motivation  
 Laufzeit von Algorithmen  
 O-Notation  
 Bestimmung der asymptotischen Laufzeit  
 Skalierbarkeit  
 Komplexitätstheorie  
 Quadratische Falle  
 Zusammenfassung

# Die Multiplikationsregel



- Multiplikationsregel:  $O(f) \cdot O(g) = O(f \cdot g)$ .
- Sei  $f' \in O(f)$  und  $g' \in O(g)$ .
- D.h. es ex.  $c_1, c_2, n_1$  und  $n_2$  mit:  
 $f'(n) \leq c_1 \cdot f(n)$  für alle  $n \geq n_1$  und  
 $g'(n) \leq c_2 \cdot g(n)$  für alle  $n \geq n_2$ .
- Setze  $n_0 = \max\{n_1, n_2\}$  und  $c = c_1 \cdot c_2$ .
- Dann gilt offensichtlich:  $f'(n) \cdot g'(n) \leq c \cdot f(n) \cdot g(n)$  für alle  $n \geq n_0$ .
- Die Multiplikationsregel ist relevant für die **Ineinanderschachtelung** von Schleifen in Programmen.

Motivation  
Laufzeit von Algorithmen  
O-Notation  
Bestimmung der asymptotischen Laufzeit  
Skalierbarkeit  
Komplexitätstheorie  
Quadratische Falle  
Zusammenfassung

# Beispiele



- 1  $2n^3 + 3n^2 + 10n + 2 = O(n^3)$  (betrachte  $c = 17$  und  $n_0 = 1$ )  
*Regel:* Bei Polynomen dominieren die Terme mit dem höchsten Exponenten.
- 2  $\frac{n^3+n}{n^4-2} = O(n^{-1})$  (betrachte  $c = 4$  und  $n_0 = 2$ )
- 3  $n^k = O(e^n)$  für fixes  $k$  (wähle  $c = k!$  und  $n_0 \geq 0$ ).  
Denn  $\frac{n^k}{k!} \leq \sum_{i=0}^{\infty} \frac{n^i}{i!} = e^n$ .  
*Regel:* Polynome werden durch die Exponentialfunktion dominiert.
- 4  $O(2^{1000}) = O(1)$ : Alle konstanten Funktionen sind äquivalent.

Motivation  
Laufzeit von Algorithmen  
O-Notation  
Bestimmung der asymptotischen Laufzeit  
Skalierbarkeit  
Komplexitätstheorie  
Quadratische Falle  
Zusammenfassung

# Hierarchie von Größenordnungen



- $O(1)$ : konstante Funktionen
- $O(\log n)$ : logarithmische Funktionen
- $O(n)$ : lineare Funktionen
- $O(n \log n)$ : log-lineare Funktionen
- $O(n^2)$ : quadratische Funktionen
- $O(n^k)$  für bel., festes  $k \in \mathbb{N}$ : polynomielle Funktionen
- $O(k^n)$ : für bel., festes  $k \in \mathbb{N}$ : exponentielle Funktionen

Motivation  
Laufzeit von Algorithmen  
O-Notation  
Bestimmung der asymptotischen Laufzeit  
Skalierbarkeit  
Komplexitätstheorie  
Quadratische Falle  
Zusammenfassung

# 4 Bestimmung der asymptotischen Laufzeit



Motivation  
Laufzeit von Algorithmen  
O-Notation  
Bestimmung der asymptotischen Laufzeit  
Skalierbarkeit  
Komplexitätstheorie  
Quadratische Falle  
Zusammenfassung

## Bestimmen der Größenordnung der Laufzeit für Programmstück A



- A ist einfache Zuweisung oder I/O-Anweisung:  $O(1)$
- A ist eine Folge von Anweisungen oder Folge von Operationen: Additionsregel anwenden.
- A ist eine if-Anweisung:
  - „if cond: B“: Additionsregel für Laufzeit von cond und Laufzeit von B.
  - „if cond: B; else: C“: Maximum der Laufzeit von B und C. Dann Additionsregel für cond und das Maximum.
- A ist eine Schleife „while cond: B“. Bestimme Maximum der Laufzeit von cond und B innerhalb der Schleifenausführung. Multipliziere mit Anzahl der Schleifenausführungen.
- Wenn A for-Schleife ist, entsprechend.
- A ist Funktionsaufruf: Bestimme den Laufzeitaufwand für die aufgerufene Funktion.

Motivation  
 Laufzeit von Algorithmen  
 O-Notation  
 Bestimmung der asymptotischen Laufzeit  
 Skalierbarkeit  
 Komplexitätstheorie  
 Quadratische Falle  
 Zusammenfassung

## Binäre Suche – noch einmal



```
def bin_search(e1, sli):
 left, right = 0, len(sli) - 1
 while left <= right:
 mid = (left+right)//2
 if sli[mid] < e1: left = mid + 1
 elif sli[mid] > e1: right = mid - 1
 else: return True
 return False
```

- Der Aufwand innerhalb der Schleife ist konstant (unabhängig von der Größe von sli).
- Die Schleife wird  $\lceil \log_2 n \rceil$  ausgeführt.
- Der Algorithmus hat eine Laufzeit von  $O(\log n)$  (in der Größe der Liste) – er hat **logarithmische Laufzeit**.
- Achtung: Natürlich ist es auch korrekt zu sagen, der Algorithmus hat eine Laufzeit von  $O(n^2)$
- ... aber man gibt immer die kleinste obere Schranke an.

Motivation  
 Laufzeit von Algorithmen  
 O-Notation  
 Bestimmung der asymptotischen Laufzeit  
 Skalierbarkeit  
 Komplexitätstheorie  
 Quadratische Falle  
 Zusammenfassung

## 5 Skalierbarkeit



Motivation  
 Laufzeit von Algorithmen  
 O-Notation  
 Bestimmung der asymptotischen Laufzeit  
 Skalierbarkeit  
 Komplexitätstheorie  
 Quadratische Falle  
 Zusammenfassung

## Skalierbarkeiten: Maximale Eingabelänge pro Zeiteinheit



**Annahme:** Ein Rechenschritt pro  $\mu\text{sec}$ . Dann folgt bei einer Laufzeit von  $T(n)$  die maximale Eingabelänge für gegebene Rechenzeit:

| $T(n)$       | 1 Sek.    | 1 Min.     | 1 Std.        |
|--------------|-----------|------------|---------------|
| $n$          | 1.000.000 | 60.000.000 | 3.600.000.000 |
| $n \log_2 n$ | 62.746    | 2.801.417  | 133.378.058   |
| $n^2$        | 1000      | 7.745      | 60.000        |
| $n^3$        | 100       | 391        | 1.532         |
| $2^n$        | 19        | 25         | 31            |

Hier sieht man, dass **konstante Faktoren** tatsächlich nicht so interessant sind.

Motivation  
 Laufzeit von Algorithmen  
 O-Notation  
 Bestimmung der asymptotischen Laufzeit  
 Skalierbarkeit  
 Komplexitätstheorie  
 Quadratische Falle  
 Zusammenfassung

**Annahme:** Bisher war die maximale Eingabelänge  $p$ . Nach einem Technologiesprung um den **Faktor 10** ergibt sich folgende maximale Eingabelänge:

| $T(n)$       | alt | neu (10× schneller) |
|--------------|-----|---------------------|
| $n$          | $p$ | $10p$               |
| $n \log_2 n$ | $p$ | fast $10p$          |
| $n^2$        | $p$ | $3.16p$             |
| $n^3$        | $p$ | $2.15p$             |
| $2^n$        | $p$ | $p + 3.3$           |

Motivation  
 Laufzeit von Algorithmen  
 O-Notation  
 Bestimmung der asymptotischen Laufzeit  
 Skalierbarkeit  
 Komplexitätstheorie  
 Quadratische Falle  
 Zusammenfassung

- Wir haben bisher nur die Zeit als wesentliche Ressource gemessen. Man kann aber auch:
  - den Verbrauch an Speicherplatz bestimmen,
  - den Kommunikationsaufwand (die Bandbreite) bestimmen.
- Es gibt außerdem weitere asymptotische Notationen, die aber in der Informatik weniger häufig auftauchen:
- $f = \Omega(g)$ , wenn  $g = O(f)$ , wenn also  $f$  mindestens so schnell wächst wie  $g$
- $f = \Theta(g)$ , wenn  $f$  und  $g$  genauso schnell wachsen, wenn also  $f = \Omega(g)$  und  $f = O(g)$ .
- $f = o(g)$ , falls  $g \neq 0$  und  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ , d.h.  $g$  wächst viel stärker als  $f$ .
- $f = \omega(g)$ , falls  $g = o(f)$ .
- **Bemerkung:** In der Zahlentheorie wird die Notation auch benutzt, es ex. aber einige Unterschiede.

Motivation  
 Laufzeit von Algorithmen  
 O-Notation  
 Bestimmung der asymptotischen Laufzeit  
 Skalierbarkeit  
 Komplexitätstheorie  
 Quadratische Falle  
 Zusammenfassung

Motivation  
 Laufzeit von Algorithmen  
 O-Notation  
 Bestimmung der asymptotischen Laufzeit  
 Skalierbarkeit  
 Komplexitätstheorie  
 Quadratische Falle  
 Zusammenfassung

- Bisher haben wir den Laufzeitbedarf (besser: Laufzeitwachstum) von **Algorithmen** untersucht.
- Man kann diese Frage aber eine Stufe abstrakter stellen und nach dem Laufzeitbedarf eines **Problems** fragen.
- Beispiel: Welches Laufzeitwachstum hat der **beste Algorithmus** für das Suchen eines Elements in einer sortierten Liste?
- Hier quantifizieren wir über **alle möglichen Algorithmen** für das Problem!
- Das Gebiet der **Komplexitätstheorie** in der Theoretischen Informatik beschäftigt sich damit, Antworten auf solche Fragen zu finden.
- Das bekannte **Milleniumsproblem**, ob **P = NP** ist, entstammt diesem Gebiet.

Motivation  
 Laufzeit von Algorithmen  
 O-Notation  
 Bestimmung der asymptotischen Laufzeit  
 Skalierbarkeit  
 Komplexitätstheorie  
 Quadratische Falle  
 Zusammenfassung

## NP-Vollständigkeit



- Es gibt eine Menge von algorithmischen Problemen, bei denen es **einfach** ist zu überprüfen, ob eine gegebene Struktur eine Lösung ist. Es ist aber kein Algorithmus bekannt, der **schnell** eine Lösung findet.
- Beispiel: Ist eine gegebene **Boolesche Formel erfüllbar** (SAT), d.h. gibt es eine Belegung der Booleschen Variablen, die die Formel wahr macht:  $(a \vee b) \wedge (c \vee \neg a)$
- Bei gegebener Belegung ( $a = 1, b = 0, c = 1$ ) einfach überprüfbar. Eine erfüllende Belegung kann man (vermutlich nur) durch Ausprobieren finden.
- Solche Probleme kann man formal charakterisieren und bezeichnet sie als **NP-vollständig**.
- Wenn **P = NP**, dann kann man Lösungen in Polynomialzeit finden.
- Wenn **P  $\neq$  NP**, wird man nie effiziente Algorithmen finden.

Motivation  
Laufzeit von Algorithmen  
O-Notation  
Bestimmung der asymptotischen Laufzeit  
Skalierbarkeit  
Komplexitätstheorie  
Quadratische Falle  
Zusammenfassung

## 7 Quadratische Falle



Motivation  
Laufzeit von Algorithmen  
O-Notation  
Bestimmung der asymptotischen Laufzeit  
Skalierbarkeit  
Komplexitätstheorie  
Quadratische Falle  
Zusammenfassung

## Ausblick: Die quadratische Falle vermeiden



- Wir hatten gesehen, dass quadratische Laufzeiten **dramatisch schlechter** als lineare oder log-lineare Laufzeiten sind – speziell wenn die Eingaben etwas größer werden.
- Vermeide sie deshalb wenn möglich!
- Beispiel:
  - Es kommt ein Datenstrom mit monoton wachsenden Zahlen herein, der möglicherweise Doppelungen enthält.
  - Alle auftretenden Zahlen sollen in einer Liste aufsteigend gespeichert werden.
- Mögliche Lösung:

```
def record_data(newelement, li):
 if not newelement in li:
 li.append(newelement)
```
- Welche asymptotische Laufzeit ergibt sich bei  $n$  Aufrufen?
- Wie kann man es besser machen?

Motivation  
Laufzeit von Algorithmen  
O-Notation  
Bestimmung der asymptotischen Laufzeit  
Skalierbarkeit  
Komplexitätstheorie  
Quadratische Falle  
Zusammenfassung

## Nicht in die quadratische Falle gefallen!



- Die asymptotische Laufzeit for `record_data` ist  $O(n^2)$ , da der `in`-Test linear in der Länge der Liste ist.
- Bessere Lösung:

```
def record_data_fast(newelement, li):
 if newelement != li[-1]:
 li.append(newelement)
```
- Alternativ, wenn z.B. die Daten nicht monoton wachsen oder fallen, andere Datenstruktur benutzen.
- `dict` und `set` haben (erwartete) **konstante** Zugriffszeit!

Motivation  
Laufzeit von Algorithmen  
O-Notation  
Bestimmung der asymptotischen Laufzeit  
Skalierbarkeit  
Komplexitätstheorie  
Quadratische Falle  
Zusammenfassung



- Die **abstrakte** Laufzeit von Algorithmen auf einer Eingabe bestimmt man durch Zählen der *Basisoperationen*.
- für die **Skalierbarkeit** interessiert uns, wie schnell die Laufzeit mit der Größe der Eingabe (meist im schlechtesten Fall) wächst.
- Konstante Faktoren und endliche Anfangsstücke interessieren uns nicht.
- **Landausche O-Notation!**
- Unterscheide lineares, quadratisches, polynomielles und exponentielles Wachstum!
- Vermeide die **quadratische Falle**, die sich aus Basisoperationen mit nicht-konstanter Laufzeit ergeben!