

# Formale Grundlagen der Informatik 1

## Kapitel 20

### Zeit- und Platzkomplexität

Frank Heitmann  
heitmann@informatik.uni-hamburg.de

27. Juni 2016

## Motivation

Bisher haben wir mit TMs bzw. Algorithmen

- Probleme gelöst/entschieden/berechnet.

Dabei war

- entscheidbar = gut
- nicht entscheidbar = schlecht

Aber ...

## Motivation

Nur zu wissen, *dass* ein Problem gelöst werden kann, reicht im Allgemeinen nicht!

### Die Aufgabe...

Wie viel

- **Zeit** und
- **Platz**

brauchen wir zur Lösung des Problems?

## Motivation

Dies wird abhängen

- Von dem **Problem**
- Von der **Größe der Probleminstanz**

### Bemerkung

Eine Probleminstanz ist eine mögliche Eingabe. Wollen wir z.B. Zahlen addieren, so ist dies als Sprache  $L = \{ \langle x, y, z \rangle \mid x + y = z \}$ . Eine Probleminstanz ist dann ein Tripel (2, 3, 5) (Ja-Instanz) oder (2, 3, 6) (Nein-Instanz).

Die Größe hängt hier von der Länge der Zahlen ab.

## Zeitkomplexität bei DTMs

## Definition (Zeit- und Platzkomplexität der DTM)

Sei  $M$  eine DTM.

- ① Für eine Rechnung benötigt  $M$  so viel Zeit, wie in der Rechnung einzelne Konfigurationen durchlaufen werden.
- ②  $M$  benötigt in einer Rechnung soviel Platz, wie *verschiedene* Felder besucht werden.

## Beispiel

In

$$z_0ab \vdash Az_1 \vdash ABz_2\# \vdash ABXz_3\# \vdash ABz_3X$$

werden 5 Zeiteinheiten benötigt und 4 Felder benutzt (das vierte Feld wird nicht geändert und bleibt leer).

## Zeitkomplexität bei DTMs

## Definition (Zeit- und Platzkomplexität der DTM (Fortsetzung))

Seien  $t, s$  Funktionen von  $\mathbb{N}$  nach  $\mathbb{R}$ . Eine DTM  $M$  ist

- ①  $t(n)$ -zeitbeschränkt genau dann, wenn

$$t(n) \geq \max\{k \mid \exists w \in L(M) : n = |w| \text{ und } M \text{ akzeptiert in } k \text{ Schritten}\}$$

- ②  $s(n)$ -platzbeschränkt genau dann, wenn

$$s(n) \geq \max\{k \mid \exists w \in L(M) : n = |w| \text{ und } M \text{ akzeptiert mit Platzbedarf } k\}$$

## Anmerkung zur Zeitkomplexität

## Anmerkung

- ① Man bezeichnet dies als **worst-case**-Komplexität, da alle Eingaben der Länge  $n$  betrachtet werden und  $t(n)$  mindestens so groß sein muss, wie der Zeitbedarf im schlimmsten Fall (analog für den Platzbedarf und  $s(n)$ ).
- ②  $t$  und  $s$  sind nur deswegen Abbildungen auf die *reelle Zahlen*, damit wir später mit Funktionen wie  $n^2 + n/2$  arbeiten können.
- ③ Oft ignoriert man Anfangswert und erlaubt auch  $t(n) = n^2 - 5 \cdot n$ , was erst ab  $n \geq 4$  einen sinnvollen Wert ergibt. Außerdem rundet man  $t(n)$  und  $s(n)$  ab, um auch Brüche u.ä. zu erlauben.

## Anmerkung zur Zeitkomplexität

## Beispiel

Ist eine DTM  $t(n)$ -zeitbeschränkt mit

$$t(n) = n^2 - n/2,$$

dann darf eine Rechnung

- auf einem Wort  $w$  der Länge 10 nur maximal  $100 - 5 = 95$  Schritte benötigen.
- Auf einem Wort der Länge 3 nur maximal  $9 - 1,5 = 7,5 \approx 7$  Schritte.

## Anmerkung

Es wird dann stets eine möglichst genaue Schranke  $t$  gesucht!

## Zeitkomplexität bei NTMs

## Definition (Zeit- und Platzkomplexität der NTM)

Sei  $M$  eine NTM. In einer fest vorgegebenen Erfolgsrechnung

- ① benötigt eine NTM so viel Zeit, wie in der Rechnung einzelne Konfigurationen durchlaufen werden und
- ② soviel Platz, wie *verschiedene* Felder besucht werden.

## Anmerkung

Das ist erstmal so wie bei DTMs, aber wir halten hier eine (Erfolgs-)Rechnung fest!

## Zeitkomplexität bei NTMs

## Definition (Zeit- und Platzkomplexität der NTM (Fortsetzung))

Eine NTM  $M$  akzeptiert ein  $w \in L(M)$  mit

- ① der **Zeitbeschränkung**  $t \in \mathbb{R}$  genau dann, wenn die kürzeste Erfolgsrechnung  $\lceil t \rceil$  Schritte hat
- ② der **Platzbeschränkung**  $s \in \mathbb{R}$  genau dann, wenn die Erfolgsrechnung mit dem geringsten Platzbedarf  $\lceil s \rceil$  Felder besucht.

## Zeitkomplexität bei NTMs

## Definition (Zeit- und Platzkomplexität der NTM (Fortsetzung))

Seien  $t, s$  Funktionen von  $\mathbb{N}$  nach  $\mathbb{R}$ . Eine NTM  $M$  ist

- ①  **$t(n)$ -zeitbeschränkt** genau dann, wenn

$$t(n) \geq \max\{k \mid \exists w \in L(M) : n = |w| \text{ und } M \text{ akzeptiert mit Zeitbeschränkung } k\}$$

- ②  **$s(n)$ -platzbeschränkt** genau dann, wenn

$$s(n) \geq \max\{k \mid \exists w \in L(M) : n = |w| \text{ und } M \text{ akzeptiert mit Platzbeschränkung } k\}$$

## Bandkompression und lineares speed-up

## Satz (Bandkompression)

Zu jeder  $s(n)$ -platzbeschränkten TM und jedem  $c \in \mathbb{R}$ ,  $c > 0$  gibt es eine äquivalente  $c \cdot s(n)$ -platzbeschränkte TM.

## Satz (lineares speed-up)

Zu jeder  $t(n)$ -zeitbeschränkten TM mit  $\inf_{n \rightarrow \infty} \frac{t(n)}{n} = \infty$  und jedem  $c \in \mathbb{R}$ ,  $c > 0$  gibt es eine äquivalente  $c \cdot t(n)$ -zeitbeschränkte TM.

## Hinweis

Beide Beweise gehen indem mehrere Bandzellen zu einer zusammengefasst werden. Man beachte, dass es in beiden Fällen nur um einen linearen Faktor geht!

## Zusammenfassung: Zeit- und Platzkomplexität

## Zusammenfassung

- Ziel: Zeit- und Platzbedarf eines Algorithmus messen.
- Die Funktion  $t(n)$  gibt an, wie lange eine TM/ein Algorithmus bei einer Eingabe der Länge  $n$  im *schlimmsten Fall* braucht.
- Die Funktion  $s(n)$  entsprechend für den Platzbedarf.
- Bei DTMs für ein Wort die eine Rechnung betrachten, bei NTMs die beste.

## Zusammenfassung: Zeit- und Platzkomplexität

## Zusammenfassung

- Wir werden noch sehen, dass es nicht weiter schlimm ist, dass wir hier vor allem auf Erfolgsrechnung schauen. In den uns interessierenden Fällen kann man die Schranken dann auch für die Nicht-Erfolgsrechnungen einführen. Damit dann:
  - Eine TM ist  $t(n)$ -zeitbeschränkt heißt: Ein Wort der Länge  $n$  wird nach  $t(n)$  Schritten akzeptiert oder abgelehnt.
  - Eine TM ist  $s(n)$ -platzbeschränkt heißt: Eine Rechnung auf einem Wort der Länge  $n$  benutzt maximal  $s(n)$  Bandzellen.

## Motivation

Wir führen jetzt **Komplexitätsklassen** ein. Diese sind ähnlich wie *REG*, *CF* usw. Sprachfamilien. Mit ihnen werden aber Probleme erfasst, die in einer bestimmten Zeit-/Platzschranke gelöst werden können.

## Komplexitätsklassen

## Definition (Komplexitätsklasse)

Für  $s, t : \mathbb{N} \rightarrow \mathbb{R}$  mit  $t(n) \geq n + 1$  und  $s(n) \geq 1$ :

$$\begin{aligned}
 DTIME(t(n)) &:= \{L \mid L = L(A) \text{ für eine } t(n)\text{-zeitbeschränkte DTM } A\} \\
 NTIME(t(n)) &:= \{L \mid L = L(A) \text{ für eine } t(n)\text{-zeitbeschränkte NTM } A\} \\
 DSPACE(s(n)) &:= \{L \mid L = L(A) \text{ für eine } s(n)\text{-platzbeschränkte DTM } A\} \\
 NSPACE(s(n)) &:= \{L \mid L = L(A) \text{ für eine } s(n)\text{-platzbeschränkte NTM } A\}
 \end{aligned}$$

## Komplexitätsklassen

## Definition (Komplexitätsklasse)

$$P := \{L \mid \text{es gibt ein Polynom } p \text{ und eine } p(n)\text{-zeitbeschränkte DTM } A \text{ mit } L(A) = L\}$$

$$NP := \{L \mid \text{es gibt ein Polynom } p \text{ und eine } p(n)\text{-zeitbeschränkte NTM } A \text{ mit } L(A) = L\}$$

$$PSPACE := \{L \mid \text{es gibt ein Polynom } p \text{ und eine } p(n)\text{-platzbeschränkte DTM } A \text{ mit } L(A) = L\}$$

$$NPSPACE := \{L \mid \text{es gibt ein Polynom } p \text{ und eine } p(n)\text{-platzbeschränkte NTM } A \text{ mit } L(A) = L\}$$

## Anmerkung

Diese Klassen sind in der Informatik von besonderer, zentraler Bedeutung!

## P und NP

Wegen der Speedup- und Kompressionsätze gilt:

$$P = \bigcup_{i \geq 1} DTIME(n^i)$$

$$NP = \bigcup_{i \geq 1} NTIME(n^i)$$

$$PSPACE = \bigcup_{i \geq 1} DSPACE(n^i)$$

$$NPSPACE = \bigcup_{i \geq 1} NSPACE(n^i)$$

## Teilmengenbeziehungen

## Satz

Es gilt

$$DTIME(f) \subseteq NTIME(f)$$

$$DSPACE(f) \subseteq NSPACE(f)$$

$$P \subseteq NP$$

$$PSPACE \subseteq NPSPACE$$

Der Beweis folgt direkt aus den Definitionen...

## Zusammenfassung: Komplexitätsklassen

## Zusammenfassung

- In den Komplexitätsklassen werden Probleme zusammengefasst, die in einer Zeit-/Platzschranke gelöst werden können.
- Besonders wichtig sind  $P$  und  $NP$ .
- Ist ein Problem in  $P$  heißt das: Es gibt ein Polynom  $p$ , so dass eine DTM existiert, die das Problem mit Zeitschranke  $p$  löst (also bei einer Eingabe der Länge  $n$  maximal  $p(n)$  Schritte benötigt).
- Bei  $NP$  ebenso, aber mit einer NTM statt einer DTM.

## Von TMs zu realen Computern

Bis hierhin haben wir uns mit der Komplexität bei TMs beschäftigt.

Man kann aber zeigen, dass

- ein Computer eine TM simulieren kann
- eine TM einen Computer simulieren kann

## Anmerkung

Man zeigt hierzu, wie eine TM eine gängige imperative Programmiersprache simuliert (bzw. eine Von-Neumann-Architektur) - und andersherum. Siehe [HMU].

## Von TMs zu realen Computern

## Anmerkung

Bei (Un-)Entscheidbarkeit hatten wir das schon diskutiert. Daher ließen sich die Resultate dort auf unsere realen Computer übertragen (mit der Konsequenz, dass wichtige Probleme nicht von uns berechnet werden können).

Dass dies nicht nur für unsere Computer gilt, sondern für jedes erdenkbare Rechnermodell ist die *Church-Turing-These*.

Die *erweiterte Church-Turing-These* sagt nun, dass die Modelle alle in Polynomialzeit ineinander umwandelbar sind! (Insb. also auch obige Umwandlung  $TM \leftrightarrow \text{Computer}$ , für die man die Gültigkeit der Aussage der These zeigen kann.)

## Von TMs zu realen Computern - Zusammengefasst

## Church-Turing-These

Die Klasse der Turing-berechenbaren Funktionen stimmt mit der Klasse der intuitiv berechenbaren Funktionen überein.

## Folgerung

Alle hinreichend mächtigen Berechnungsmodelle sind äquivalent.

## Erweiterte Church-Turing-These

Sind  $B_1$  und  $B_2$  zwei Rechnermodelle, so gibt es ein Polynom  $p$ , so dass  $t$  Rechenschritte von  $B_1$  bei einer Eingabe der Länge  $n$  durch  $p(t, n)$  Rechenschritte von  $B_2$  simuliert werden können.

## Folgerung

Zwei (Turing-)äquivalente Modelle lassen sich mit polynomiellen Mehraufwand ineinander umwandeln.

## Zur Bedeutung von P

## Wichtige Anmerkung

Damit erhält die Komplexitätsklasse  $P$  eine besondere Bedeutung! Alles was in  $P$  ist, kann mit jedem Berechnungsmodell in  $P$  berechnet werden (auch wenn die Polynome unterschiedlich sind). Außerdem kann man statt mit TMs bei Algorithmen auch mit gängigen Programmiersprachen argumentieren, so fern einen nur interessiert, ob ein Problem in  $P$  ist (und nicht der exakte Zeitbedarf auf einem festen Rechnermodell).

## Wichtige Anmerkung

Die herausragende Bedeutung von  $P$  liegt auch darin, dass man i.A. davon ausgeht, dass **in  $P$  die Probleme liegen, die wir effizient lösen können**. Wenn wir also für ein Problem einen langsameren Algorithmus haben, versuchen wir immer einen zu finden, der das Problem in Polynomialzeit löst!

## Zu TMs und Algorithmen

## Wichtige Anmerkung

Wir betonen: In vielen Fällen, z.B. wenn es um Entscheidbarkeit geht oder auch wenn es wie jetzt um  $P$  und  $NP$  geht, kann man bisweilen von Turingmaschinen abstrahieren und stattdessen mit z.B. Algorithmen argumentieren. Will man z.B. zeigen, dass  $P$  gegenüber Vereinigung abgeschlossen ist, muss man nicht zwingend mit TMs arbeiten (kann man aber) und stattdessen mit Algorithmen argumentieren.

## Landau-Notation

Wir merken uns

- Polynome sind wichtig!

Meist interessieren uns aber Faktoren vor einem Polynom nicht und auch die "kleineren Teile" in einem Polynom interessieren uns nicht. Die Polynome

- $10 \cdot n^3 + 2 \cdot n^2 - 50 \cdot n + 25$  und
- $n^3$

verhalten sich "im Grunde genommen gleich". (Wenn man sehr große Werte einsetzt, kommt da nämlich fast das gleiche raus...)

## Zur Bedeutung von P (und NP)

Weiterhin zur Bedeutung von P:

- Man geht allgemein davon aus, dass ein Problem, das in Polynomialzeit gelöst werden kann, *effizient* lösbar ist.
  - In Polynomialzeit bedeutet, es gibt ein Polynom  $p$ , so dass eine Instanz der Länge  $n$  in  $p(n)$  Schritten gelöst werden kann.
- Ein Polynom wächst mit  $n$  in einem vertretbaren Aufwand.
  - Für z.B.  $n^2$  ist  $10^2 = 100$  und  $50^2 = 2500$ .
  - Für aber  $2^n$  ist  $2^{10} \approx 1000$  und  $2^{50} \approx 1$  Billarde.
- Für die meisten Probleme, für die man einen Algorithmus in P hat, findet man einen, mit einem kleinen Polynom.

Zur Bedeutung von NP später mehr...

## Landau-Notation

Um das zu formalisieren, führen wir die  $O$ - bzw. Landau-Notation ein. Warum man das braucht sieht man gut bei Algorithmen....

## Vorschau

Nachfolgendes als kleiner Ausblick auf AuD, wo dies viel gebraucht wird. Für uns wird vor allem wichtig sein, dass  $O(f(n))$  für  $c \cdot f(n)$  steht mit einer Konstanten  $c$  und für eine Funktion  $f$  wie z.B.  $f(n) = n^2$ . Das  $O$  "verschluckt" also die Konstante.

## Algorithmenanalyse...

**Algorithmus 1** Lineare Suche

```

1: for i = 0 to n do
2:   if a[i] == max mustermann then
3:     return true
4:   end if
5: end for
6: return false

```

Wir behandeln nachfolgend *Zeit-* und *Platzkomplexität* (im uniformen Maß). Eine (elementare) Anweisung zählt dabei als eine Zeiteinheit. Eine benutzte (elementare) Variable als eine Platzeinheit.

Zeit- und Platzbedarf wird *abhängig von der Länge n der Eingabe* gezählt. Wir arbeiten hierfür mit Funktionen  $f : \mathbb{N} \rightarrow \mathbb{R}$ .

## O-Notation - Motivation

Wir werden nachfolgend *konstante Faktoren* ignorieren. Diese "verschwinden" in der *O-Notation* (auch *Landau-Notation*).

- ⇒ Konzentration auf das wesentliche.
- ⇒ Abstrahiert von verschiedenen Rechnerarchitekturen.
- ⇒ Guter Vergleich von Algorithmen möglich (praxisbewährt).
- ⇒ Ferner werden wir "anfängliche Schwankungen" ignorieren können.
- ABER:  $5 \cdot n$  und  $5000 \cdot n$  wird als "im Prinzip" gleich angesehen werden!

## O-Notation - Definition

## Definition (O-Notation (und Verwandte) - Die wichtigen)

- $O(g(n)) = \{f \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \leq c \cdot |g(n)|\}$
- $\Omega(g(n)) = \{f \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \geq c \cdot |g(n)|\}$
- $\Theta(g(n)) = \{f \mid \exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c_1 \cdot |g(n)| \leq |f(n)| \leq c_2 \cdot |g(n)|\}$

## Bemerkung

$f$  ist dabei stets eine Funktion von  $\mathbb{N}$  nach  $\mathbb{R}$ , also  $f : \mathbb{N} \rightarrow \mathbb{R}$ .

## Merkhilfe

## Bemerkung

Eine kleine *Merkhilfe* (nicht mehr!)

- $f \in O(g) \approx f \leq g$
- $f \in \Omega(g) \approx f \geq g$
- $f \in \Theta(g) \approx f = g$

Wir sagen, dass  $f$  asymptotisch kleiner gleich, größer gleich bzw. gleich  $g$  ist, wenn  $f \in O(g)$ ,  $f \in \Omega(g)$  bzw.  $f \in \Theta(g)$  gilt.

## O-Notation - Beispiel

$$\Theta(g(n)) = \{f \mid \exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : \\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

## Beispiel

Wir wollen  $\frac{1}{2}n^2 - 3n \in \Theta(n^2)$  zeigen. Wir suchen also Konstanten  $c_1, c_2, n_0$ , so dass

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

für alle  $n \geq n_0$  erfüllt ist. Teilen durch  $n^2$  führt zu

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

. Dies kann man z.B. mit  $c_1 = \frac{2}{8}, c_2 = 1, n_0 = 24$  erfüllen.

## Wichtige Funktionen

## Bemerkung

Wichtige Funktionen in der Algorithmik/Algorithmenanalyse:

- Konstante Funktionen
- Logarithmen ( $\log, \ln$ )
- Wurzelfunktionen ( $\sqrt{n}$ )
- Polynome (beachte:  $\sqrt{n} = n^{1/2}$ )
- Exponentialfunktionen ( $2^n$ )
- ... Kombinationen davon

In welchem Zusammenhang stehen diese bzgl. der O-Notation?  
( $\Rightarrow$  Modul A&D!)

## Zum Logarithmus

## Nebenbemerkung

Mit  $\log$  werden wir stets eine Variante des Logarithmus zur Basis 2 meinen, nämlich:

$$\log(n) := \begin{cases} 1 & , \text{ falls } n \leq 1 \\ \lfloor \log_2(n) \rfloor + 1 & , \text{ sonst.} \end{cases}$$

Damit ist  $\log(n)$  die Länge der Binärdarstellung einer natürlichen Zahl  $n$ . (Die Anzahl von Speicherstellen/Schritten ist stets eine natürliche Zahl!) Andere Logarithmen werden nur selten benötigt und sind bis auf einen konstanten Faktor ohnehin gleich. Die Rechengesetze wie z.B.  $\log(x \cdot y) = \log(x) + \log(y)$  und  $\log(x^r) = r \cdot \log(x)$  sind oft hilfreich.

## Algorithmus 1: Lineare Suche

## Algorithmus 2 Algorithmus 1

```

1: for  $i = 0$  to  $n$  do
2:   if  $a[i] == \text{max mustermann}$  then
3:     return true
4:   end if
5: end for
6: return false

```

## Analyse

Laufzeit ist linear in der Länge  $n$  des Arrays, d.h. in  $O(n)$  oder genauer sogar in  $\Theta(n)$ . (Korrektheit ist noch zu zeigen. Dazu in A&D mehr...)

## Algorithmus 2: Fakultät

**Algorithmus 3**  $\text{fac}(n)$ 

```

1:  $res = 1$ 
2: for  $i = 1$  to  $n$  do
3:    $res = res \cdot i$ 
4: end for
5: return  $res$ 

```

## Analyse

Die Laufzeit ist in  $O(n)$ . Aber Achtung! Dies ist **exponentiell in der Größe der Eingabe!** Diese ist nämlich nur in  $O(\log n)$ .

## Vorgehen

Vorgehen, wenn wir einen Algorithmus analysieren:

- ① Überlegen bzgl. welcher *Kenngroße* der Eingabe wir messen wollen. Diese kann sich von der Größe der Eingabe unterscheiden! (Beispiel: Anzahl Knoten eines Graphen).
  - ② Laufzeit/Speicherbedarf bzgl. dieser Kenngroße ausdrücken.
  - ③ Nochmal überlegen, ob dies die Aussage des Ergebnisses verfälscht (so wie bei der Fakultätsberechnung oben).
- ⇒ I.A. sollte sich die eigentliche Eingabegröße leicht durch die Kenngroße ausdrücken lassen und der Unterschied sollte nicht zu groß sein.
- + Beim Graphen mit  $n$  Knoten ist die Adjazenzmatrix in  $O(n^2)$ .
  - Ist eine Zahl  $n$  die Eingabe, so ist die Eingabegröße in  $O(\log n)$ . Eine Laufzeit von  $O(n)$  wäre also exponentiell in der Eingabe!

## Zusammenfassung

Wir merken uns vor allem:

Ist  $f(n)$  eine Funktion (z.B.  $f(n) = n^2$ ), dann bedeutet  
Die Laufzeit des Algorithmus/der TM ist in  $O(f(n))$ .

dass es eine Konstante  $c$  gibt, so dass bei einer Eingabe der Länge  $n$  die Turingmaschine/der Algorithmus maximal  $c \cdot f(n)$  Schritte macht.

## Fragen

**Algorithmus 4** Find Maximum

```

1:  $max = 1$ 
2: for  $i = 2$  to  $n$  do
3:   if  $A[i] > A[max]$  then
4:      $max = i$ 
5:   end if
6: end for
7: return  $max$ ;

```

Laufzeit? 1.  $n$       2.  $O(n)$       3.  $O(n-1)$   
4.  $O(3 \cdot n)$     5.  $O(n^2)$     6. nichts davon

## Fragen

**Algorithmus 5** Sortieren mit Max

---

```

1: for  $i = n$  downto 1 do
2:    $idx = \max(A)$ 
3:    $B[i] = A[idx]$ 
4:    $A[idx] = 0$ 
5: end for
6: return  $B$ 

```

---

Laufzeit? 1.  $O(n)$     2.  $O(n^2)$     3.  $O(n \cdot \log n)$   
 4.  $O(\log n)$     5.  $O(2^n)$     6. nichts davon

## Fragen

Das P in NP steht für Polynomialzeit. Wofür steht das N?

- ① nicht(-polynomiell)
- ② nicht-deterministisch

## Fragen

Was gilt?

- ①  $P \subseteq NP$
- ②  $NP \subseteq P$
- ③  $P \cap NP = \emptyset$
- ④ unvergleichbar!

## Fragen

**Algorithmus 6** Suchraum durchsuchen (deterministisch!)

---

```

1: for all  $A \subseteq S$  do
2:   if  $\sum_{x \in A} x = \sum_{x \in \bar{A}} x$  then
3:     return true
4:   end if
5: end for
6: return false

```

---

Laufzeit? 1.  $O(n)$     2.  $O(n^2)$     3.  $O(n \cdot \log n)$   
 4.  $O(n^3)$     5.  $O(2^n)$     6. nichts davon

## Fragen

**Algorithmus 7** Suchraum durchsuchen (nichtdeterministisch!?)

```

1: for all  $A \subseteq S$  do
2:   if  $\sum_{x \in A} x = \sum_{x \in \bar{A}} x$  then
3:     return true
4:   end if
5: end for
6: return false

```

Wie schnell könnt ihr das gleiche Problem *nichtdeterministisch*

lösen? 1.  $O(\log n)$  2.  $O(n)$  3.  $O(n \cdot \log n)$   
 4.  $O(n^2)$  5.  $O(2^n)$  6. nichts davon

## Zur Nachbereitung

## Zur Nachbereitung

- 1  $O(n)$  ( $n$  ist die Länge des Arrays  $A$ )
- 2  $O(n^2)$  (ebenso)
- 3 nichtdeterministisch
- 4  $P \subseteq NP$  gilt. (Ob auch  $NP \subseteq P$  gilt, ist offen...)
- 5  $O(2^n)$  ( $n$  ist die Größe der Menge  $S$ )
- 6  $O(n)$  (wie eben - im Algorithmus die Menge  $A$  raten!)

## Die Klasse P

$$\begin{aligned}
 P &:= \{L \mid \text{es gibt ein Polynom } p \text{ und eine} \\
 &\quad p(n)\text{-zeitbeschränkte DTM } A \text{ mit } L(A) = L\} \\
 &= \bigcup_{i \geq 1} \text{DTIME}(n^i)
 \end{aligned}$$

## Anmerkung

- Ein Algorithmus (eine TM)  $A$  akzeptiert eine Sprache  $L$  in *polynomialer Zeit*, wenn sie von  $A$  akzeptiert wird und zusätzlich ein  $k \in \mathbb{N}$  existiert, so dass jedes  $x \in L$  mit  $|x| = n$  in Zeit  $O(n^k)$  akzeptiert wird.
- Soll  $A$  die Sprache  $L$  entscheiden, wird für jeden String  $x$  der Länge  $n$  in Zeit  $O(n^k)$  entschieden, ob  $x \in L$  gilt (oder nicht).

## Akzeptieren und Entscheiden

## Satz

$$P = \{L \mid L \text{ wird von einem Algo. in Polynomialzeit entschieden}\}$$

## Beweis.

Die Richtung von rechts nach links ist klar (Warum?), es ist also zu zeigen, dass jede in Polynomialzeit akzeptierbare Sprache auch in Polynomialzeit entscheidbar ist. Sei  $A$  ein Algorithmus der  $L$  in Zeit  $O(n^k)$  akzeptiert. Es gibt dann eine Konstante  $c$ , so dass  $A$  die Sprache in höchstens  $c \cdot n^k$  Schritten akzeptiert. Ein Algorithmus  $A'$ , der  $L$  entscheidet, berechnet bei Eingabe  $x$  zunächst  $s = c \cdot |x|^k$  und simuliert  $A$  dann  $s$  Schritte lang. Hat  $A$  akzeptiert, so akzeptiert auch  $A'$ , hat  $A$  bisher nicht akzeptiert, so lehnt  $A'$  die Eingabe ab. Damit entscheidet  $A'$  die Sprache  $L$  in  $O(n^k)$ .  $\square$

## Akzeptieren und Entscheiden

## Satz

$$P = \{L \mid L \text{ wird von einem Algo. in Polynomialzeit entschieden}\}$$

## Anmerkung

Man beachte, dass der obige Beweis nicht konstruktiv war! Wenn man weiss, dass ein Algorithmus mit polynomialer Laufzeit existiert, weiss man nicht zwingend wie dieser aussieht geschweige denn, wie die Schranke aussieht.

## Wichtige Anmerkung

Wir gehen nachfolgend davon aus, dass mit einer Zeitschranke  $t(n)$  stets gemeint ist, dass die TM bei einer Eingabe der Länge  $n$  nach  $t(n)$  Schritten spätestens anhält. (Insb. für die Klassen  $P$  und  $NP$  ist das in Ordnung!)

## Ein Problem in P

Typisches Beispiel eines Problems in P:

$$\text{PATH} = \{ \langle G, s, t, k \rangle \mid \begin{array}{l} G = (V, E) \text{ ist ein ungerichteter Graph,} \\ s, t \in V, \\ k \geq 0 \text{ ist eine ganze Zahl und} \\ \text{es existiert ein } s\text{-}t\text{-Pfad in } G, \\ \text{der aus höchstens } k \text{ Kanten besteht.} \end{array} \}$$

## Verifikation in polynomialer Zeit

Ein sehr ähnliches Problem – oder?

$$\text{L-PATH} = \{ \langle G, s, t, k \rangle \mid \begin{array}{l} G = (V, E) \text{ ist ein ungerichteter Graph,} \\ s, t \in V, \\ k \geq 0 \text{ ist eine ganze Zahl und} \\ \text{es existiert ein } s\text{-}t\text{-Pfad in } G, \\ \text{der aus } \textit{mindestens} \ k \text{ Kanten besteht.} \end{array} \}$$

## Anmerkung

Schwierig zu lösen (zumindest effizient), aber gegeben ein Pfad, kann schnell *überprüft* werden, ob er die Kriterien erfüllt.

## Ausblick

Was wir getan haben und noch tun werden:

- ① Wir wollten über Rechnung/Algorithmen reden können
  - Dafür haben wir die TM eingeführt
  - und argumentiert, dass sie das gleiche kann wie Computer
  - dass das vermutlich für alle Berechnungsmodelle gilt sagt die Church-Turing-These.
- ② Dann wollten wir über das Berechenbare reden
  - was mit TMs gut ging
  - bis wir gemerkt haben, dass es unentscheidbare Probleme gibt.
- ③ Dann wollten wir zumindest die entscheidbaren effizient lösen
  - dazu haben wir Zeit- und Platzkomplexität eingeführt
  - und dort wird uns dann die  $NP$ -Vollständigkeit unsere Grenzen aufzeigen...