

8.1 Dynamische Mengen

8.2 Charakterisierung von Aufwänden

8.3 Aufwände von Algorithmen

8.4 Komplexität von Problemen



Innensicht

Algorithmenentwurf und Algorithmen (incl. Aufwände)

- Suche und Sortieren (Reihen, Folgen, Bäume)
- Graphen
- Dyn. Programmieren
- Geometr. Algorithmen

Prozesse

- Prozesse
- Prozesskommunikation
- Parallelität/Synchronisation
- Java: Thread-Programmierung

Außensicht

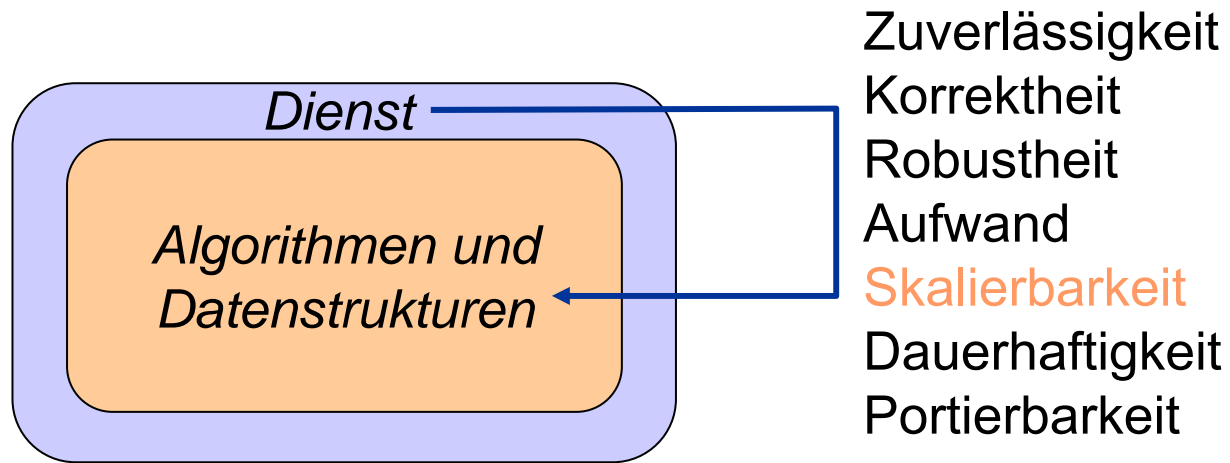
Skalierbarkeit und Persistenz

- Mengenorientierung
- Assoziativer Zugriff, Schlüsselbegriff
- Aufwände
- Polymorphie
- Schema
- Deskr. Sprachen (SQL)

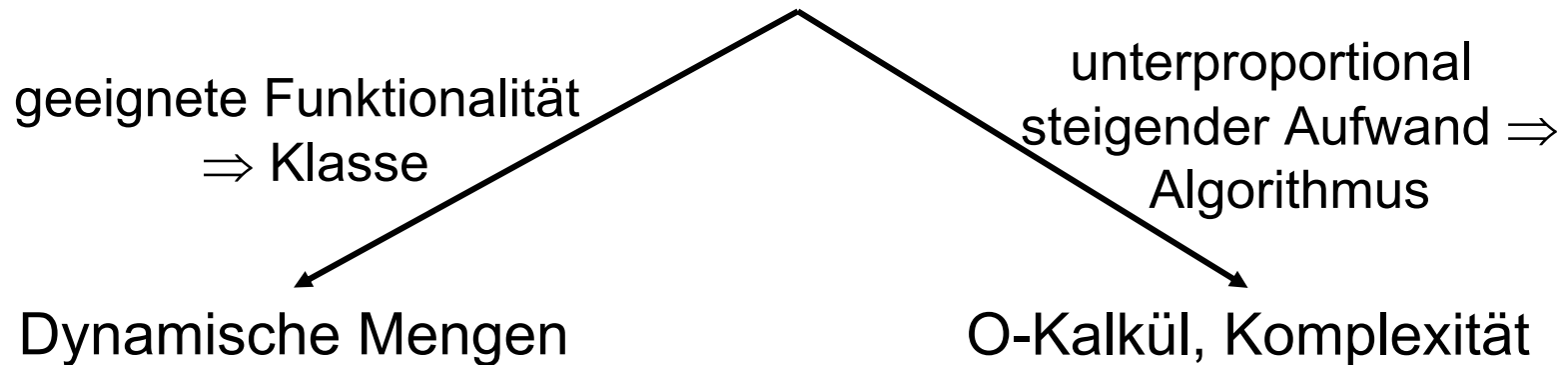
Autonome Dienste

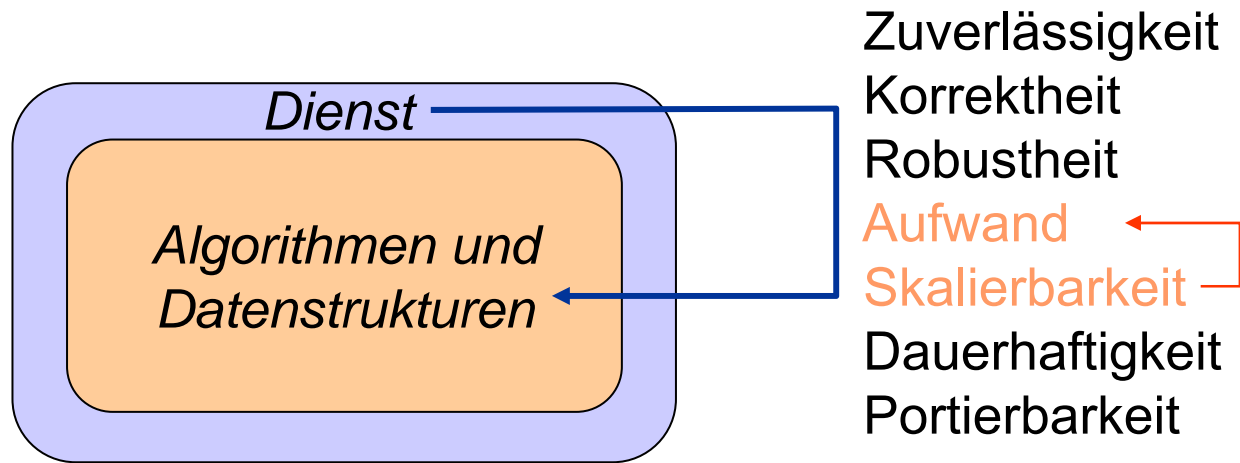
- Enge/lose Kopplung
- Protokolle/Automaten
- Verteilung



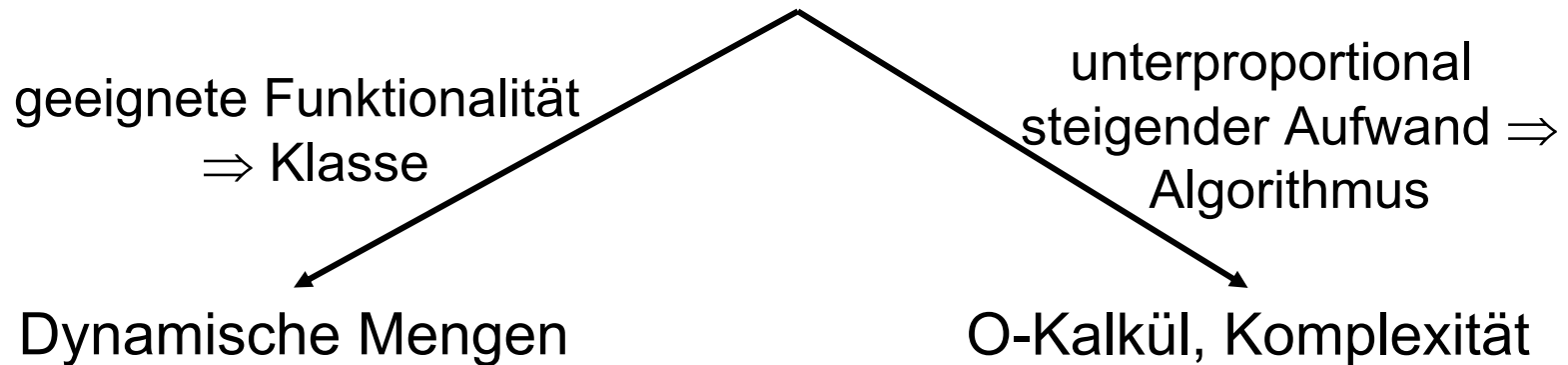


Skalierbarkeit: Beherrschbarkeit des Wachstums





Skalierbarkeit: Beherrschbarkeit des Wachstums



Offenheit: Der Umfang der Zustandsinformationen ist nach oben unbegrenzt.

- Objekte oder Datenelemente können jederzeit hinzukommen oder verschwinden.
- Somit ist deren Zahl nicht vorhersagbar.

Statische Datenstrukturen sind hier ungeeignet, da diese nur das Speichern einer festen Anzahl von Datenelementen erlauben.

Im Gegensatz dazu erlaubt eine dynamische Datenstruktur, beliebig viele Datenelemente aufzunehmen:

- Ein Verwalter derartiger Mengen muss als Dienstgeber Operatoren für das Einfügen, Löschen und Auffinden von Mengenelementen anbieten.
- Angesichts der Offenheit kann nicht jedem Objekt oder Datenelement eine eigene Variable zugeordnet werden.
- Daher bedarf es Operatoren für navigierenden und/oder assoziativen Zugriff zum Auslesen einer Untermenge der hinterlegten Elemente.



(aus Informatik I):

Kapselung:

```
module Set ( Set, CreateSet, single, insert,
             isElem, delete, union, intersect,
             diff, isEmpty) where
```

Datentyp:

```
data Set  $\alpha$  = CreateSet | Add (Set  $\alpha$ )  $\alpha$ 
```

Signatur:

single	$:: \alpha \rightarrow \text{Set } \alpha$
insert	$:: \text{Eq } \alpha \parallel \text{Set } \alpha \rightarrow \alpha \rightarrow \text{Set } \alpha$
delete	$:: \text{Eq } \alpha \parallel \text{Set } \alpha \rightarrow \alpha \rightarrow \text{Set } \alpha$
union	$:: \text{Eq } \alpha \parallel \text{Set } \alpha \rightarrow \text{Set } \alpha \rightarrow \text{Set } \alpha$
intersect	$:: \text{Eq } \alpha \parallel \text{Set } \alpha \rightarrow \text{Set } \alpha \rightarrow \text{Set } \alpha$
diff	$:: \text{Eq } \alpha \parallel \text{Set } \alpha \rightarrow \text{Set } \alpha \rightarrow \text{Set } \alpha$
isElem	$:: \text{Eq } \alpha \parallel \alpha \rightarrow \text{Set } \alpha \rightarrow \text{Bool}$
isEmpty	$:: \text{Set } \alpha \rightarrow \text{Bool}$



single x = Add CreateSet x
insert s x = union s (single x)
delete s x = diff s (single x)

union CreateSet s = s
union (Add s x) z
 | isElem x z = union s z
 | otherwise = Add (union s z) x

intersect CreateSet s = CreateSet
intersect (Add s x) z
 | isElem x z = Add (intersect s z) x
 | otherwise = intersect s z



$\text{diff CreateSet } s = \text{CreateSet}$

$\text{diff (Add } s \ x) \ z$

$\begin{array}{l} | \text{ not (isElem } x \ z) \\ | \text{ otherwise} \end{array} = \begin{array}{l} \text{Add (diff } s \ z) \ x \\ \text{diff } s \ z \end{array}$

$\text{isElem } x \ \text{CreateSet} = \text{False}$

$\text{isElem } x \ (\text{Add } s \ y)$

$\begin{array}{l} | \ x == y \\ | \text{ otherwise} \end{array} = \begin{array}{l} \text{True} \\ \text{False} \end{array}$

$\text{isEmpty CreateSet} = \text{True}$

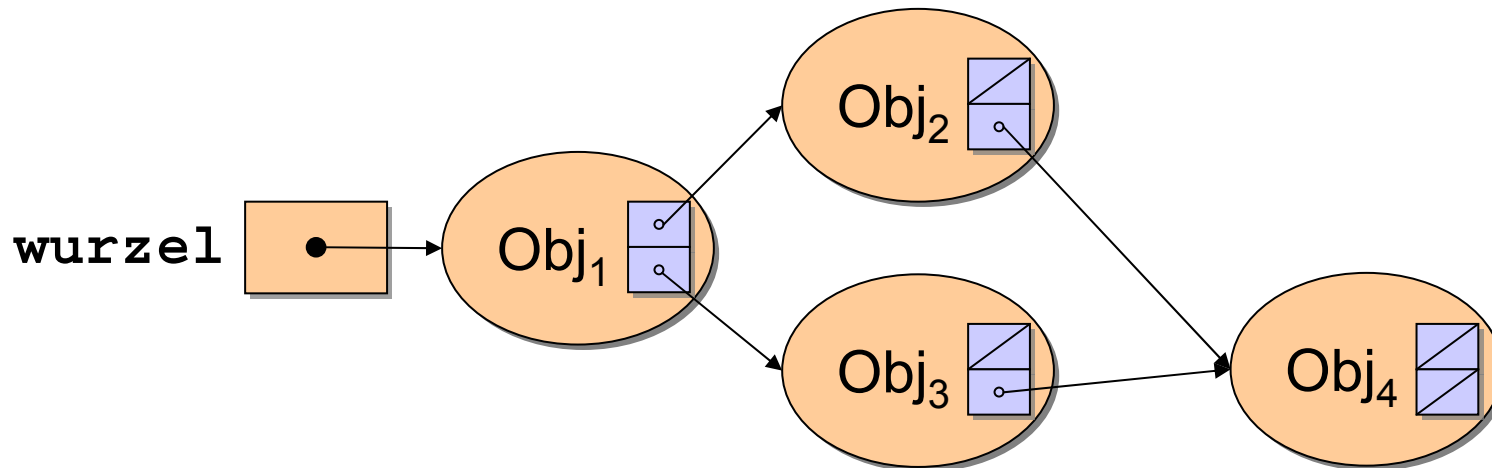
$\text{isEmpty (Add } s \ x) = \text{False}$

Vielfachmenge: Mengen, in denen Elemente mehrfach vorkommen können.



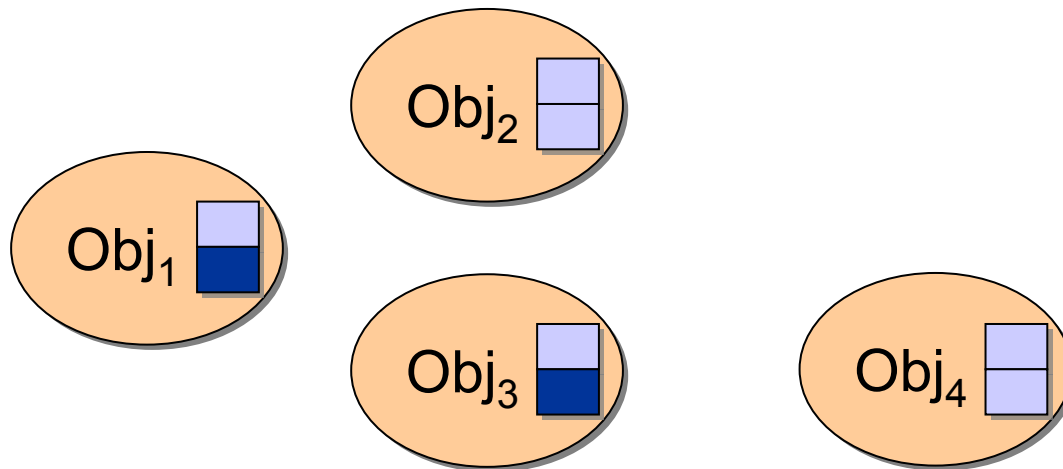
Navigierender Zugriff

- Die Datenelemente liegen in Form von Objekten vor, die untereinander über Referenzen (Ausprägungen von UML-Assoziationen) verbunden sind.
- Bestimmte Objekte werden als Einstiegspunkte (Wurzelobjekte) ausgezeichnet, von denen alle weiteren Objekte erreichbar sind.
- Die übrigen Elemente können durch Fortschreiten entlang der Referenzen erreicht werden (Navigation).



Assoziativer Zugriff

- Alternativ oder bei Fehlen von Assoziationen benötigt man ein inhaltliches Merkmal (z.B. der Inhalt eines Feldes), mit dem sich ein Objekt oder Datenelement eindeutig ansprechen lässt.
- Assoziativer Zugriff: Inhaltsbezogener Zugriff.
- Schlüssel: Für den assoziativen Zugriff herangezogenes Merkmal.



- Nur das zuletzt eingefügte Element ist zugänglich: Kellerspeicher.
- Nur das älteste noch vorhandene Element ist zugänglich: Schlange.



Generische Schnittstelle

```

interface Stack<T> {
    invariant:  $\forall a : T$ :
        this.equals (push(a).pop())  $\wedge$  push(a).top() = a  $\wedge$  push(a).empty() =
        False  $\wedge$  new Stack<T>().empty() = True  $\wedge$  (empty()  $\rightarrow$  pop() = fehler)
         $\wedge$  (empty()  $\rightarrow$  top() = fehler)

    public Stack<T> push(T a);
        { }
        { Erg.top() = a }

    public boolean empty();
        { }
        { Erg = (this.equals (new Stack<T>())) }

    public Stack<T> pop();
        {  $\neg$ this.empty() }
        { this.equals (Erg.push(this.top())) }

    public T top();
        {  $\neg$ this.empty() }
        {  $\exists$  Stack<T> k, T a : this.equals (k.push(a))  $\wedge$  Erg = a }
}
    
```



Aufwand

- Die Größe der Menge der Eingabedaten für einen Algorithmus kann stark variieren.
- Das Verhalten eines Algorithmus sollte durch die Größe der Eingabedaten nur gering beeinflusst werden:
Zum Beispiel soll ein Sortieralgorithmus 10, 100, 1000, 10000, ... Datenelemente sortieren können, ohne bei mehr Elementen bedeutend mehr Ressourcen (Zeit, Speicher) zu benötigen.

Bei der Implementierung eines Algorithmus sollte man also danach streben, dass sich der Zeit- und Speicheraufwand unterproportional zur Größe der Eingabedaten verhält.



Sortieren natürlicher Zahlen $x[1], \dots, x[n]$

```
// Maximum suchen
```

```
for (max=0, i=1; i<=n; i++)
```

n Durchläufe `if (x[i] > max) max=x[i];`

```
// Neues Feld der Länge max+1 initialisieren
```

```
s = new int[max+1];
```

```
for (i=0; i<=max; i++) s[i]=0;
```

max Durchläufe

```
// Zähle, wie oft welche Zahl in x vorkommt
```

```
for (i=1; i<n; i++) s[x[i]]++;
```

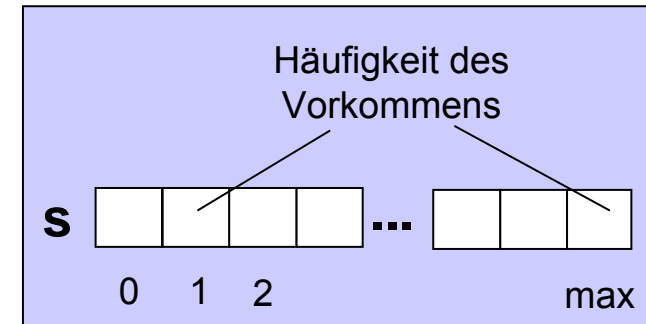
n Durchläufe

```
// x neu auffüllen
```

```
for (i=0, j=0; i<=max; i++)
```

mind. max Durchläufe

```
while (s[i]>0) {x[j++] = i; s[i]--};
```



Insgesamt $2 \cdot n + 2 \cdot \max$ Schleifendurchläufe zum Sortieren von n natürlichen Zahlen (Duplikate möglich).

Taugt der Algorithmus etwas? Unter welchen Umständen ist er sinnvoll?

Algorithmen

- verbrauchen Rechenzeit:
 - Ausführungszeit des Programms selbst
 - In der Praxis auch: Zeit für Ein-/Ausgabe, Zeit für System,...
- verbrauchen Speicher für Programm und Datenstrukturen:
 - Platz für das Programm selbst
 - Platz für seine statischen Datenstrukturen
 - Platz für seine dynamischen Datenstrukturen

Fragestellungen:

- Ist Algorithmus A schneller/sparsamer als Algorithmus B?
- Kann ein Algorithmus signifikant verbessert werden?



Umfang und Aufwand

- Umfang n : Anzahl der Eingabewerte, z.B. Länge einer Liste
- Aufwand $T(n)$: Anzahl der Zeit- bzw. Speichereinheiten, die der Algorithmus für ein Problem mit Umfang n benötigt

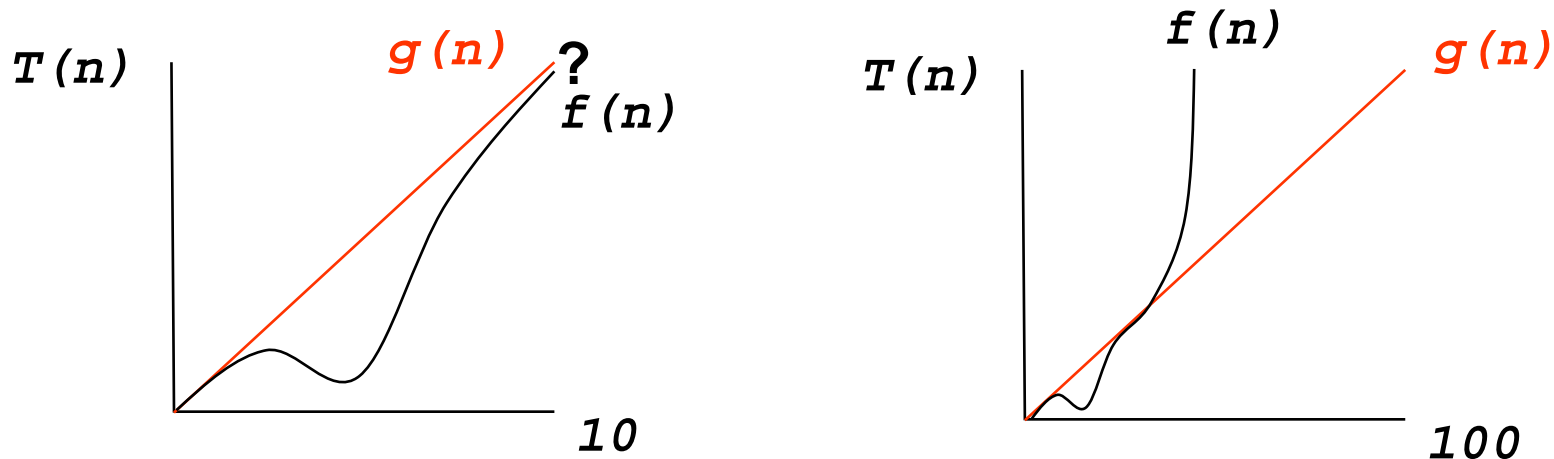
Hängt der Aufwand nicht nur vom Umfang ab, sondern auch von den tatsächlichen Eingabewerten, dann interessiert ferner:

- ungünstigster Aufwand (worst-case)
- mittlerer Aufwand (average-case)
- Aufwand im besten Fall (best-case)

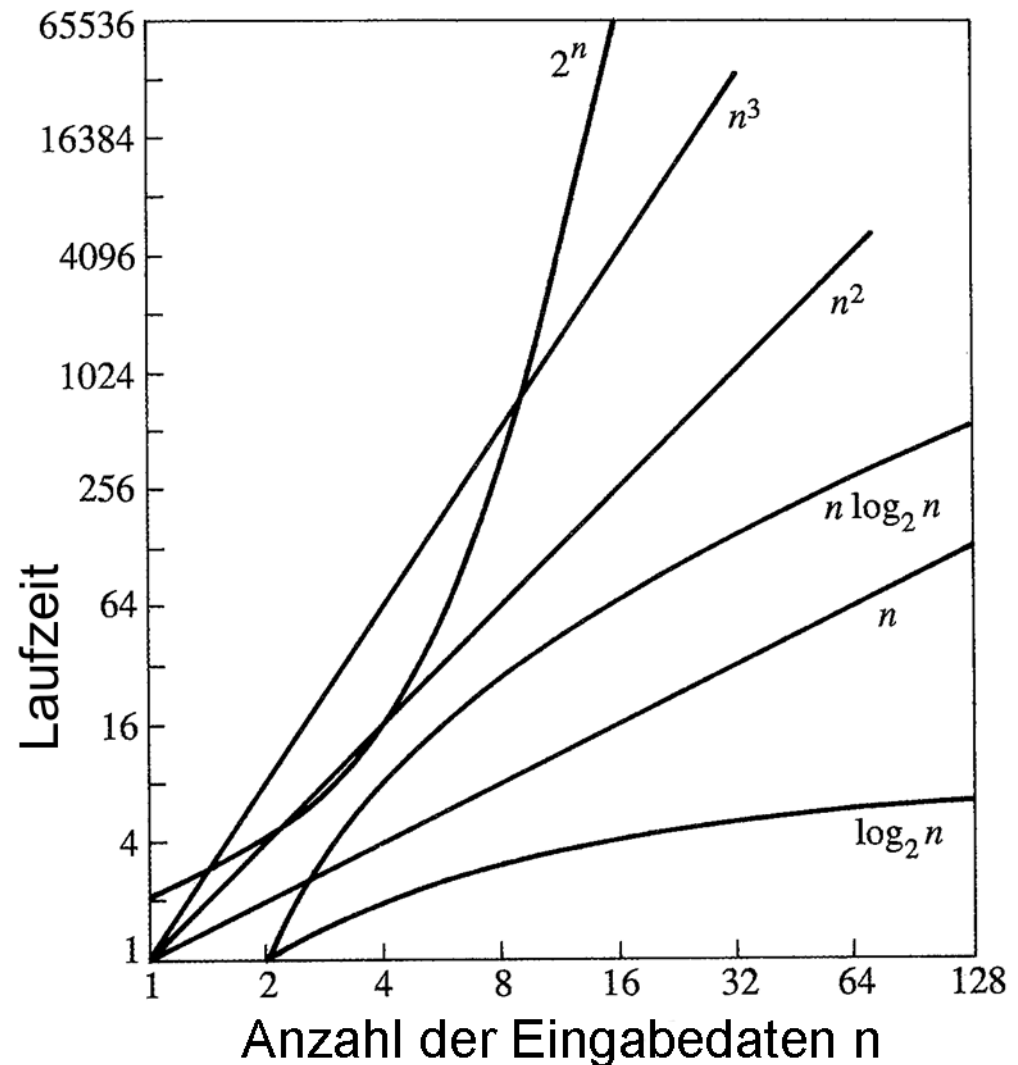


- Der exakte Aufwand eines Algorithmus etwa in Form der Anzahl der Rechenschritte bis zur Terminierung oder der verbrauchten Zeit ist kaum berechenbar.
- Lineare Faktoren sind für die Theorie uninteressant: verschiedene Rechner sind „um Faktor“ unterschiedlich schnell.
- Wichtiger ist die ungefähre Größenordnung, mit der der Aufwand in Abhängigkeit der Größe der Eingabe wächst.

Hierbei interessieren vor allem der Aufwand für sehr große Umfänge n :



- Um die Größenordnung des Aufwandswachstums festzulegen, wird das asymptotische Verhalten der Aufwandsfunktion zumeist mit einem Repräsentanten einer bestimmten Funktionsklasse verglichen.
- Der O-Kalkül erlaubt die Beschreibung solcher Vergleiche zwischen Funktionen.



- Gegeben eine irgendwie berechnete Aufwandsfunktion $f(n)$ für einen Algorithmus.
- Wir suchen einen Repräsentanten $g(n)$ so, dass $f(n)$ in einer Menge $O(g(n))$ von Funktionen (Ordnung von $g(n)$) liegt derart dass $(c, n, n_0 \in \mathbb{N})$

$$O(g(n)) = \{ f(n) \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n) \}$$

Ab Umstiegspunkt n_0 ist $f(n)$ kleiner als $g(n)$ multipliziert mit einer festzulegenden Konstante.

Ab Umstiegspunkt n_0 wächst $f(n)$ höchstens so schnell wie $g(n)$.

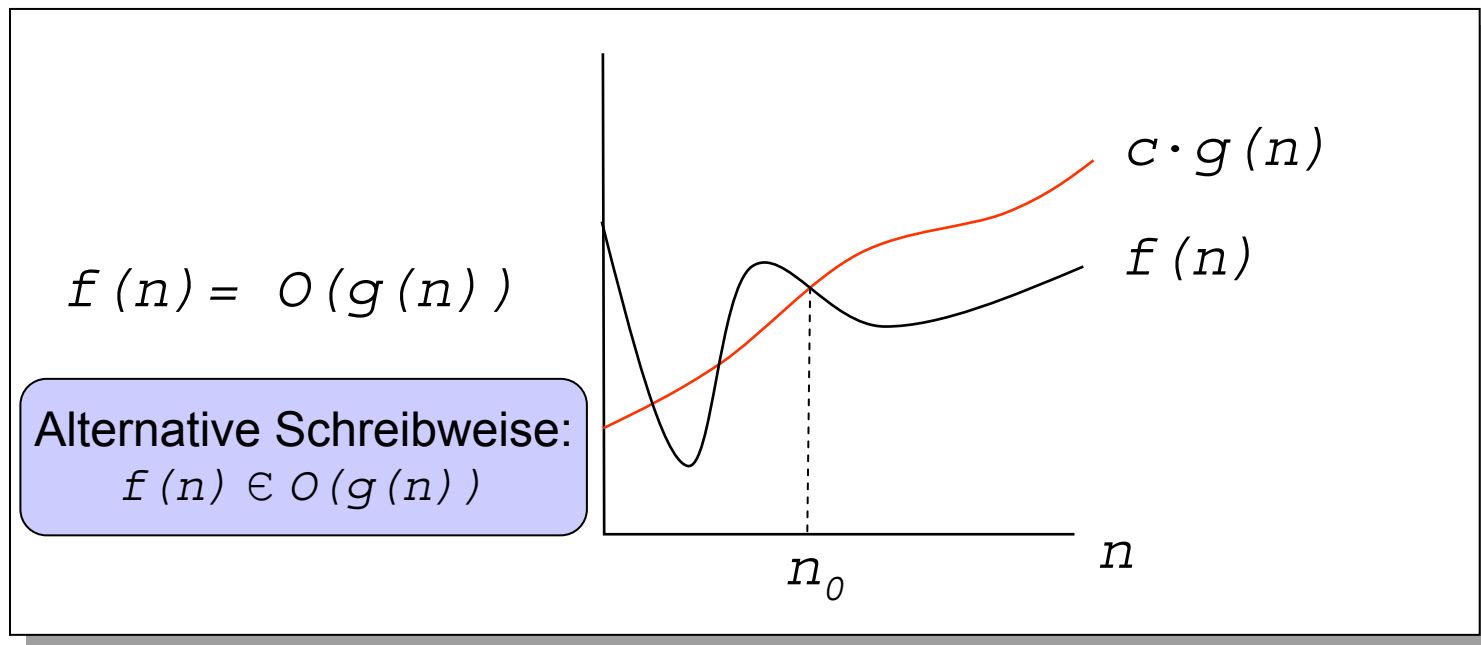


(asymptotisch obere Schranke)

- $f(n)$ wächst höchstens so schnell wie $g(n)$

Definition:

$$O(g(n)) = \{ f(n) \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n) \}$$



Beispiel: $n^2 + n = O(n^2)$

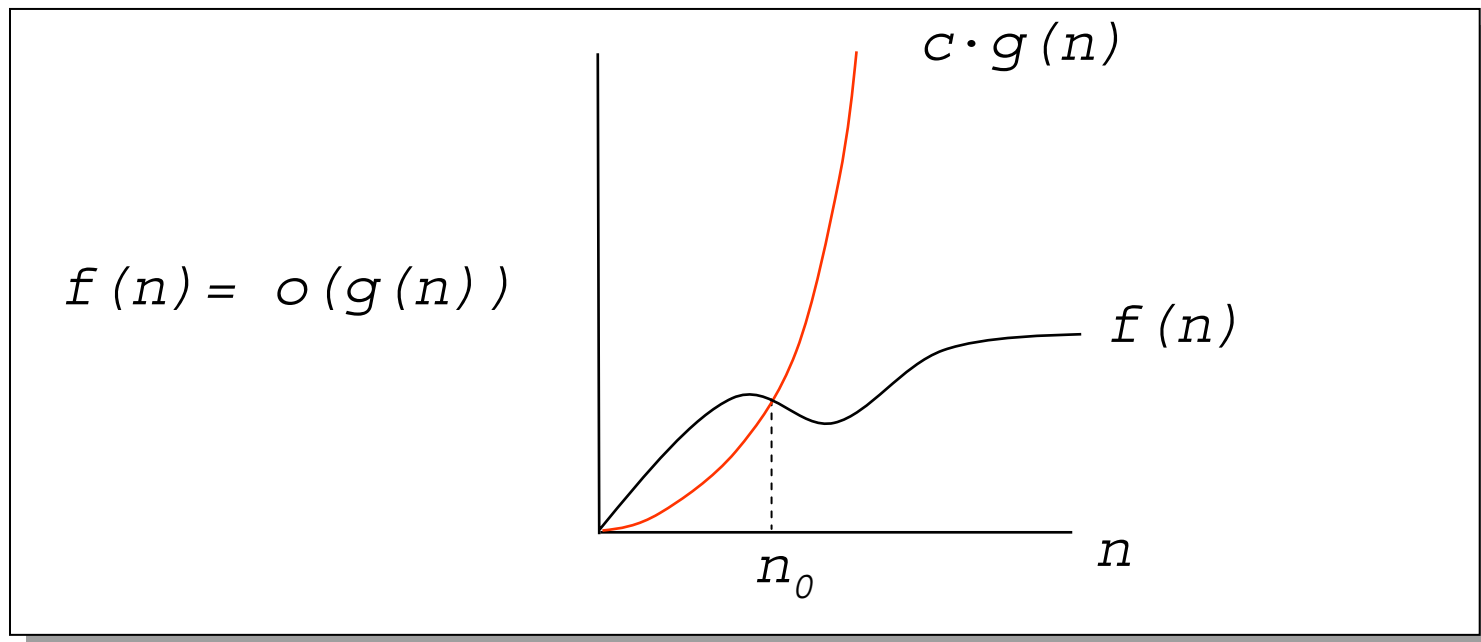
(obere Schranke)

- $f(n)$ wächst deutlich langsamer als $g(n)$

Definition:

$$o(g(n)) = \{ f(n) \mid \forall c > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq f(n) < c \cdot g(n) \}$$

Jetzt nicht wie bei O „es gibt irgend eine Konstante“, sondern „egal für welche Konstante“.



Es gilt: $\lim_{n \rightarrow \infty} f(n) / g(n) = 0$

Beispiele: $2 \cdot n = o(n^2)$ $2 \cdot n^2 \neq o(n^2)$ $2 \cdot n^2 = o(n^3)$

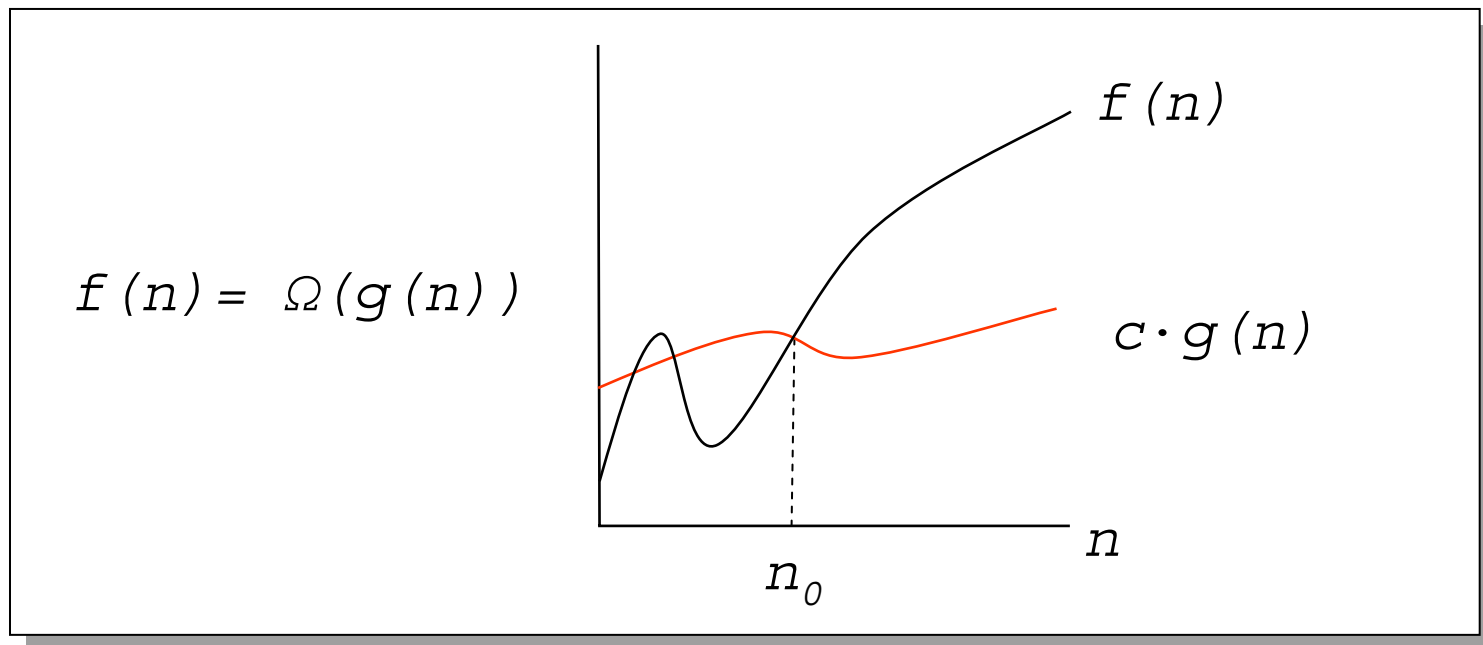


(asymptotisch untere Schranke)

- $f(n)$ wächst mindestens so schnell wie $g(n)$

Definition:

$$\Omega(g(n)) = \{ f(n) \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n) \}$$



Beispiele: $5 \cdot n^2 + 42 \cdot n + 2 = \Omega(n^2)$

$2^n + 5 \cdot n = \Omega(2^n)$

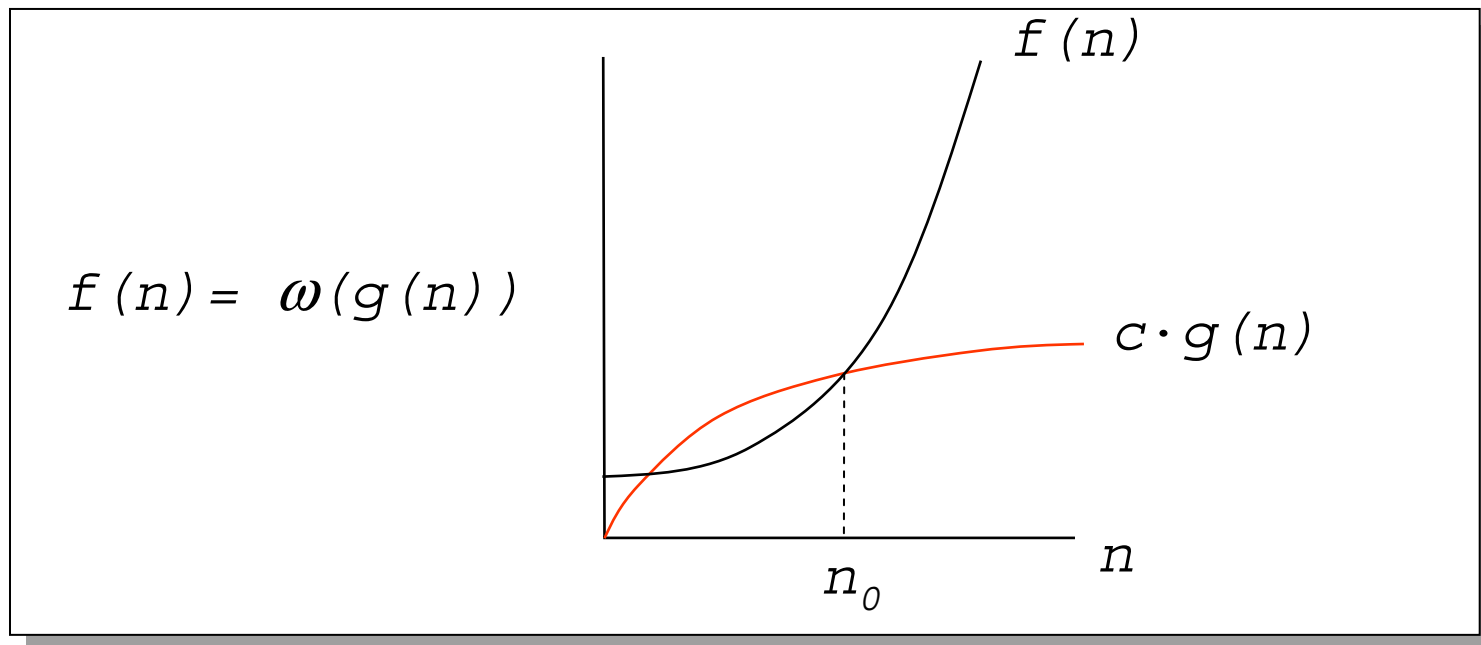


(untere Schranke)

- $f(n)$ wächst deutlich schneller als $g(n)$

Definition:

$$\omega(g(n)) = \{ f(n) \mid \forall c > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq c \cdot g(n) < f(n) \}$$



Es gilt: $\lim_{n \rightarrow \infty} f(n) / g(n) = \infty$

Beispiele: $n^2/2 = \omega(n)$ $n^2/2 \neq \omega(n^2)$



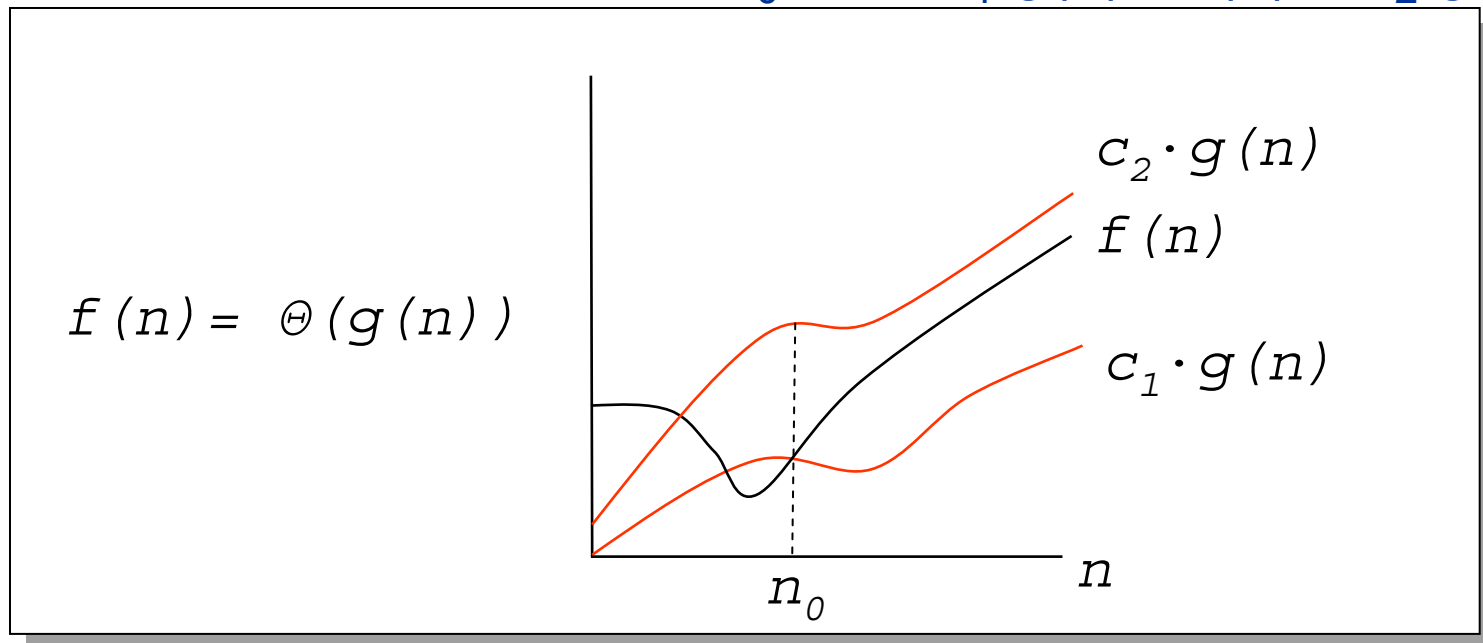
(asymptotisch gebunden)

- $f(n)$ wächst ebenso schnell wie $g(n)$

Definition:

$$\Theta(g(n)) = \{ f(n) \mid \exists c_1 > 0 \exists c_2 > 0 \exists n_0 > 0$$

$$\forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \}$$



Beispiele: $a \cdot n^2 + b \cdot n = \Theta(n^2)$

$a \cdot n^2 \neq \Theta(n)$



- Sei $f(n) = O(r(n))$ und $g(n) = O(s(n))$.

Dann:

$$f(n) + g(n) = O(r(n) + s(n))$$

$$f(n) \cdot g(n) = O(r(n) \cdot s(n))$$

$f(n) = \Theta(g(n))$ genau dann wenn $f(n) = O(g(n))$ und $f(n) = \Omega(g(n))$

$f(n) = O(g(n))$ genau dann wenn $g(n) = \Omega(f(n))$

$f(n) = o(g(n))$ genau dann wenn $g(n) = \omega(f(n))$

Reflexivität:

- $f(n) = \Theta(f(n))$, $f(n) = O(f(n))$, $f(n) = \Omega(f(n))$

Symmetrie:

- $f(n) = \Theta(g(n))$ genau dann wenn $g(n) = \Theta(f(n))$



Eigentlich interessiert man sich für den Aufwand für ein Problem

$f: A \rightarrow B$, denn

- verschiedene Algorithmen zur Berechnung von f können bei gleicher Eingabe unterschiedlichen Aufwand haben.

Daher zwei Aufgaben:

- Bestimme kleinsten möglichen Aufwand für das Problem (Komplexität des Problems) \Rightarrow später, in 8.4!
- Bestimme Algorithmus mit kleinst möglichem Aufwand \Rightarrow
 - Dauerbeschäftigung des Algorithmikers,
 - auch für uns in weiten Teilen der Vorlesung!

Auf den tatsächlichen Aufwand haben auch Einfluss:

- Programmiersprache
- Rechnerorganisation
- Prozessorgeschwindigkeit



- Aufgabe: Finde in einem Zahlenvektor den Abschnitt, dessen Elemente addiert, die größte Summe ergeben.
- Hierfür gibt es einen Algorithmus mit $O(n^3)$ und einen mit $O(n)$.

n	Alpha 21164A (533 MHz), C-Programm, $O(n^3)$ -Algorithmus	Radio Shack TRS-80 (2,03 MHz), BASIC-Programm, $O(n)$ -Algorithmus
10^1	0,58µs	195ms
10^2	0,58ms	1,95s
10^3	0,58s	19,5s
10^4	9min 40s	3min 15s
10^5	6d 16h 6min 40s	32min 30s
10^6	18y 142d 23h 6min 40s	5h 25min

Umschlagpunkt bei ca. $n=5800$ und $t=1$ min 53 s



- Wenn die Rechner 10 mal schneller werden, um wie viel kann der Umfang k zunehmen, so dass der Aufwand gleich bleibt?
- d.h. es soll gelten $10 \cdot f(k) \approx f(g(k))$

$f(n)$	$g(k)=$
$\log_2 n$	$1000 \cdot k$
n	$10 \cdot k$
$n \log_2 n$	$9-10 \cdot k$ (für große k)
n^2	$3 \cdot k$
2^n	$k + 3$
$n!$	k (für $k \gg 10$)



- Wenn ein Algorithmus sich selbst rekursiv aufruft, kann seine Laufzeit oft mit einer Rekurrenz beschrieben werden.
- Eine Rekurrenz ist eine Gleichung oder Ungleichung, bei der der Funktionswert in Form von Funktionswerten für kleinere Eingabewerte beschrieben wird.
- Einen allgemeinen Algorithmus zur Lösung einer Rekurrenz gibt es nicht.

Beispiel:

Sortieren durch Mischen

$$T(1) = \Theta(1)$$

$$T(n) = 2 \cdot T(n/2) + \Theta(n)$$

Liste in zwei
Hälften aufteilen

Ergebnis
zusammenbauen



Substitutionsmethode

- Eine Rekurrenz lässt sich lösen, indem eine Lösung „erraten“ wird.
- Berechne $T(n)$ für einige n .
- Schätze geschlossene Form für $T(n)$ (oder ggf. nur Schranke).
- Beweise (Induktion?), dass das geschätzte $T(n)$ korrekt ist.

Beispiel:

$$T(n) = 2 \cdot T(n/2) + 4 \quad \text{und} \quad T(1) = 1$$

$T(1)$	$=$	1	$=$	1	$=$	$5 \cdot 1 - 4$
$T(2)$	$=$	$4 + 2 \cdot 1$	$=$	6	$=$	$5 \cdot 2 - 4$
$T(4)$	$=$	$4 + 2 \cdot 6$	$=$	16	$=$	$5 \cdot 4 - 4$
$T(8)$	$=$	$4 + 2 \cdot 16$	$=$	36	$=$	$5 \cdot 8 - 4$
$T(16)$	$=$	$4 + 2 \cdot 36$	$=$	76	$=$	$5 \cdot 16 - 4$

$T(n)$	$=$		$=$	$5 \cdot n - 4$
	\Rightarrow	$T(n) = O(n)$		



Master Method

- Eine Rekurrenz lässt sich lösen, indem allgemeine Regeln verwendet werden.

Hauptsatz über Rekurrenzen

- Sei $a \geq 1$ und $b \geq 1$, sei $f(n)$ eine Funktion und $T(n)$ gegeben durch die Rekurrenz

$$T(n) = a \cdot T(n/b) + f(n)$$

- Dann ist $T(n)$ asymptotisch beschränkt durch:

Wenn $f(n) = O(n^{\log_b a - \varepsilon})$ für einige $\varepsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$

Wenn $f(n) = \Theta(n^{\log_b a})$, dann $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$

Wenn $f(n) = \Omega(n^{\log_b a + \varepsilon})$ für einige $\varepsilon > 0$ und wenn

$a \cdot f(n/b) \leq c \cdot f(n)$ für $c < 1$ und genügend große n ,
dann $T(n) = \Theta(f(n))$.



Beispiele für Master Method: $T(n) = a \cdot T(n/b) + f(n)$

- Wenn $f(n) = O(n^{\log_b a - \epsilon})$ für ein $\epsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$.
- Wenn $f(n) = \Theta(n^{\log_b a})$, dann $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$
- Wenn $f(n) = \Omega(n^{\log_b a + \epsilon})$ für ein $\epsilon > 0$ und
wenn $a \cdot f(n/b) \leq c \cdot f(n)$ für $c < 1$ und genügend große n , dann
 $T(n) = \Theta(f(n))$.

$$T(n) = 4 \cdot T(n/2) + n$$

Somit $a=4$, $b=2$, $f(n) = n$ und $n^{\log_b a} = n^{\log_2 4} = n^2$.

Da $f(n) = O(n^{\log_2 4 - \epsilon})$ mit z.B. $\epsilon=1$ wird der erste Fall erfüllt und daher ergibt sich $T(n) = \Theta(n^2)$.



Beispiele für Master Method: $T(n) = a \cdot T(n/b) + f(n)$

- Wenn $f(n) = O(n^{\log_b a - \epsilon})$ für ein $\epsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$.
- Wenn $f(n) = \Theta(n^{\log_b a})$, dann $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$
- Wenn $f(n) = \Omega(n^{\log_b a + \epsilon})$ für ein $\epsilon > 0$ und wenn $a \cdot f(n/b) \leq c \cdot f(n)$ für $c < 1$ und genügend große n , dann $T(n) = \Theta(f(n))$.

$$T(n) = 3 \cdot T(n/4) + n \log n$$

Somit $a=3$, $b=4$, $f(n) = n \log n$ und $n^{\log_b a} = n^{\log_4 3} = n^{0,793}$. Da $f(n) = \Omega(n^{\log_4 3 + \epsilon})$ mit z.B. $\epsilon \approx 0,2$ und für große n gilt:
 $a \cdot f(n/b) = 3(n/4) \log(n/4) \leq (3/4) n \log n = c \cdot f(n)$ mit $c=3/4$, daher wird der dritte Fall erfüllt und es ergibt sich $T(n) = \Theta(n \log n)$.



Beispiele für Master Method: $T(n) = a \cdot T(n/b) + f(n)$

- Wenn $f(n) = O(n^{\log_b a - \epsilon})$ für ein $\epsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$.
- Wenn $f(n) = \Theta(n^{\log_b a})$, dann $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$
- Wenn $f(n) = \Omega(n^{\log_b a + \epsilon})$ für ein $\epsilon > 0$ und
wenn $a \cdot f(n/b) \leq c \cdot f(n)$ für $c < 1$ und genügend große n , dann
 $T(n) = \Theta(f(n))$.

$$T(n) = T(2n/3) + 1$$

Somit $a=1$, $b=3/2$, $f(n) = 1$ und $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$.

Da $f(n) = \Theta(n^{\log_{3/2} 1}) = \Theta(1)$ wird der zweite Fall erfüllt und daher ergibt sich $T(n) = \Theta(\log n)$.

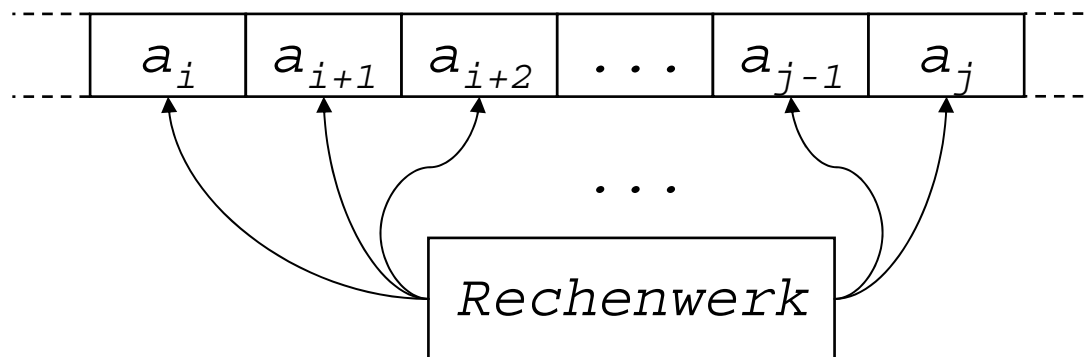


Einfluß des Maschinenmodells auf den Aufwand

- Der Aufwand eines Algorithmus hängt von der zugrundeliegenden Rechnerarchitektur ab.

Random Access Machine (RAM) dient in der Regel als Maschinenmodell für Aufwandsberechnungen :

- ein Prozessor
- Hauptspeicher unbeschränkt groß
- Speicherzelle unbeschränkt lang
- eine Zeiteinheit je elementarer Operation



Gegeben: Zeichenfolge $A = a_1, a_2, \dots, a_n$

Aufgabe: Stelle fest, ob A ein Palindrom ist, d.h.
 $\forall i \in \{1 \dots n\}: a_i = a_{n+1-i}$

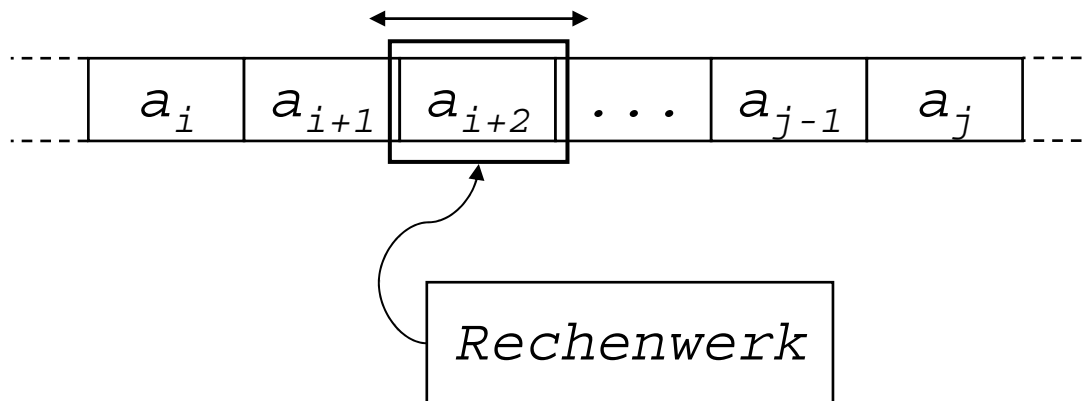
Algorithmus für Random Access Machine:

```
for (int i=1; i<=n; i++) {  
    if (a[i] != a[n+1-i]) {  
        return false;  
    }  
}  
return true;
```

Zeitaufwand ist $O(n)$.

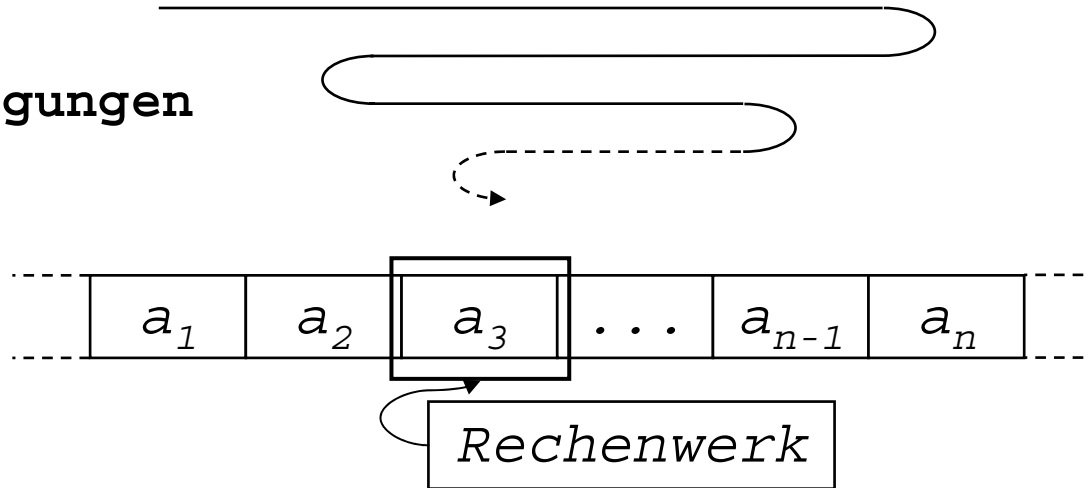


- (Potenziell) unendliches Band mit Feldern
- Genau ein Zeichen je Feld
- Ein Schreib-/Lesekopf bewegt sich über das Band.
- Nur das Zeichen, auf dem der Lesekopf steht, kann im nächsten Rechenschritt verändert werden.
- Der Kopf kann in einem Rechenschritt ein Feld nach links oder ein Feld nach rechts bewegt werden.



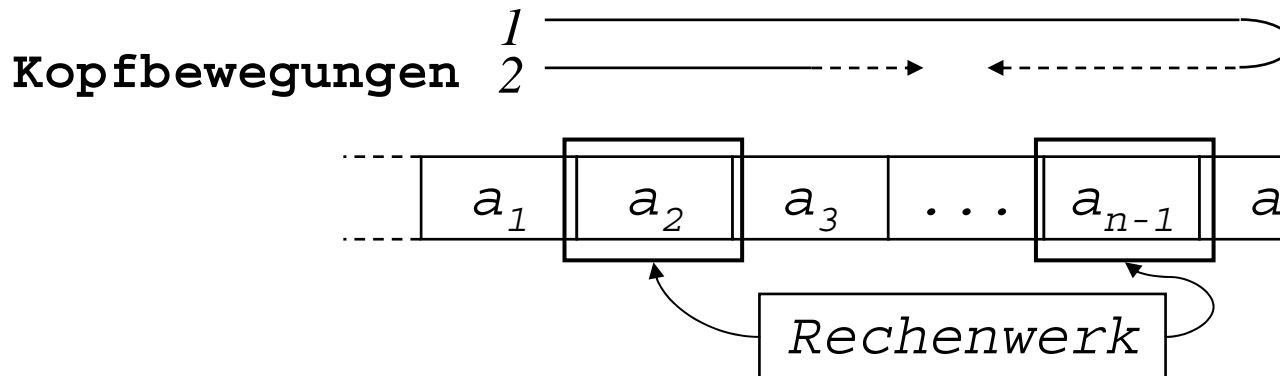
Palindrom-Algorithmus für Turing-Maschine:

Kopfbewegungen



Aufwand: $O(n^2)$

Palindrom-Algorithmus für 2-Kopf-Turing-Maschine:



Aufwand: $O(n)$

Zur Erinnerung:

- Wir haben die Aufgabe, für ein gegebenes Problem den kleinst möglichen Aufwand zu bestimmen.
- Dabei sind Zeit- und Speicheraufwand zu betrachten.



Beispiel: Wie berechnet man $n!$ am schnellsten?

	Zeit	Speicher
■ Multiplizieren: $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$	$O(n)$	$O(1)$
■ Rekursiv: $n! = n \cdot (n-1)!$	$O(n)$	$O(n)$
■ $n!$ sowieso nur darstellbar für kleine n (z.B. $n < 200$) $n! = \text{Tabelle}[n]$	$O(1)$	$O(n)$

Kann dem 3. Ansatz folgend jedes Problem in $O(1)$ gelöst werden?

- In der Tat, wenn man sich auf eine maximale Größe der Eingabe beschränkt und jedes Ergebnis vorberechnet.
- Aber selbst dann reicht oft der Speicher nicht aus.



Weiteres Beispiel: Mit welchem Aufwand lässt sich eine Liste mit n Elementen sortieren?

Sortieralgorithmen mit Vergleich der Elemente:

- Sortieren durch Einfügen $\Theta(n^2)$ im schlechtesten Fall
- Sortieren durch Mischen $\Theta(n \log_2 n)$ im schlechtesten Fall

Geht es noch besser oder ist $\Theta(n \log_2 n)$ untere Schranke?

Sortieralgorithmen für natürliche Zahlen:

- Counting Sort: $\Theta(n)$ im schlechtesten Fall
- Es geht also noch besser, aber nur unter bestimmten Randbedingungen.



Komplexität eines Problems

- Die Komplexität eines Problems entspricht dem Aufwand des „billigsten“ Algorithmus, der es löst.
- Oft wird dabei Komplexität als Tupel angesehen: (Zeitkomplexität, Platzkomplexität, Hardwarekomplexität, ...)

Komplexitätsklassen:

- Superfeine Einteilung: $T(n)$
- Feine Einteilung: $O(T(n))$
- Grobe Einteilung: $O(\text{POLY}(n))$
oder $O(\text{EXP}(n))$

mit $\text{POLY}(n) := \bigcup_{p>0} O(n^p)$ und $\text{EXP}(n) := \bigcup_{p>0} O(p^n)$



- Eine grobe Einteilung bedarf eines entsprechenden Abstraktionsniveaus.
- Dazu Formulierung als *Entscheidungsproblem*: Als Ausgabe wird entweder 1 (ja) oder 0 (nein) erwartet.
- Die meisten Probleme lassen sich auf ein oder mehrere Entscheidungsprobleme zurückführen.

Beispiel:

- Ursprüngliches Problem: Gesucht $\max(a,b)$
- Formulierung als Entscheidungsproblem: Ist $a = \max(a,b)$?



Entscheidungsprobleme als formale Sprachen

- Ein Entscheidungsproblem wird vollständig durch die Eingaben charakterisiert, die 1 als Ergebnis liefern.
- Diese Eingaben lassen sich als Worte einer formalen Sprache darstellen.
- Bei Entscheidungsproblemen üblich: Eingabedaten in Form von Binärdaten.

Beispiel:

- Entscheidungsproblem Palindrom:
Ist $Z = b_1b_2...b_n$ Palindrom?
- Eingaben, die 1 liefern:

$$L(Palindrom) = \{0, 1, 00, 11, 000, 010, 101, 111, \\ 0000, 0110, 1001, \dots\}$$



Problemreduktion als Sprachübersetzung:

- Bei der Reduktion wird ein Problem in eines mit einem bereits bekannten Lösungsalgorithmus überführt.
- In unserem Fall wird Problem X durch die Sprache $L(X)$ dargestellt.
 \Rightarrow Problemreduktion auf Y entspricht daher der Abbildung $L(X) \rightarrow L(Y)$.

Beispiel:

Problem X : ist a_1 Maximum von $A = a_1 a_2 \dots a_n$?

$L(X) = \{0,1,00,10,11,000,100,101,110,111,0000,\dots\}$

Problem Y : ist b_1 Minimum von $B = b_1 b_2 \dots b_n$?

$L(Y) = \{0,1,00,01,11,000,001,010,011,111,0000,\dots\}$

Abbildung $f: L(X) \rightarrow L(Y)$: $f(a_1 \dots a_n) = (1-a_1 \dots 1-a_n)$

Offensichtlich: $A \in L(X) \Leftrightarrow B = f(A) \in L(Y)$



Problemreduktion (Forts.): Man kann auch eine Sprache auf sich selbst reduzieren.

Beispiel:

Problem X : ist $A = a_1 a_2 \dots a_n$ teilbar durch 3?

$$L(X) = \{11, 110, 1001, 1100, 1111, 10010, 10101, \dots\}$$

Abbildung $f: L(X) \rightarrow L(X)$: $f(a_1 \dots a_n) = \text{Quersumme}(a_1 \dots a_n)$

Offensichtlich: $A \in L(X) \Leftrightarrow f(A) \in L(X)$

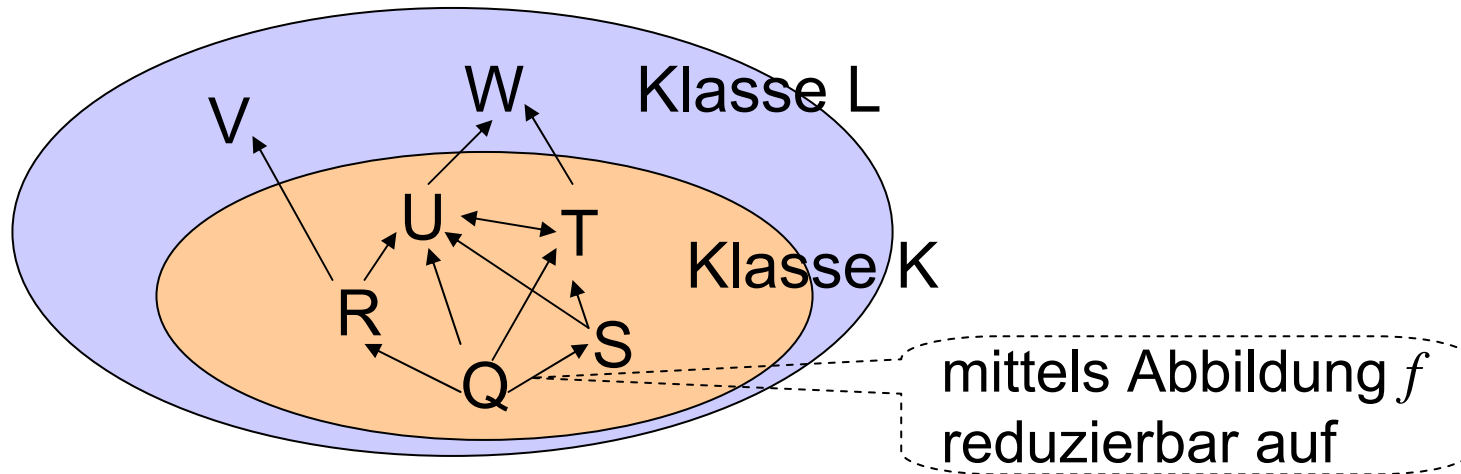


K-hart und K-vollständig

- Wenn Problem A mit „geringem“ Aufwand auf Problem B reduzierbar ist, dann gehört A mindestens zur Komplexitätsklasse von B.
 - B kann mindestens A, evtl. aber noch mehr.
 - Wir nennen dann das Problem B härter.
-
- K-hart (K-hard): Ein Problem, auf das sich alle Probleme einer Klasse K reduzieren lassen.
 - K-vollständig (K-complete): Ein K-hartes Problem, das selbst auch in K ist.



K-hart und K-vollständig



Problem	Q	R	S	T	U	V	W
K-hart	-	-	-	✓	✓	-	✓
K-vollständig	-	-	-	✓	✓	-	-

Polynomial reduzierbar und polynomial äquivalent

- Was heißt nun „mit geringem Aufwand reduzierbar“?

Definitionen:

- Wenn es eine in polynomialer Zeit berechenbare Abbildung f von L_1 auf L_2 gibt, dann heißt L_1 polynomial reduzierbar auf L_2 .
- Wenn sowohl L_1 auf L_2 als auch L_2 auf L_1 polynomial reduzierbar ist, dann heißen L_1 und L_2 polynomial äquivalent.

Folgerung:

- Leicht zu sehen, dass „polynomial reduzierbar“ transitiv und reflexiv ist.



Die Komplexitätsklasse P

- Zur Klasse P gehören die Probleme, die mit Hilfe eines deterministischen Algorithmus mit Aufwand $O(\text{POLY}(n))$ gelöst werden können.
- Algorithmen, die in $O(\text{POLY}(n))$ laufen, heißen effizient (efficient).
- Probleme, für die es einen effizienten Algorithmus gibt, heißen effizient lösbar oder praktikabel (tractable).
- Wenn ein Problem X in der Komplexitätsklasse P liegt, und wenn Y auf das Problem X polynomial reduzierbar ist, dann liegt Y auch in P.

Beispiel:

Das Problem Palindrom gehört wegen $O(n)$ zur Klasse P.



Nichtpolynomiale Probleme

- Es gibt unzählige Probleme, für die (bisher) kein Algorithmus mit polynomialem Aufwand bekannt ist.
- Daher kann man nur nach Näherungslösungen für das Problem (Heuristiken) suchen.
- Liegt nun ein Problem vor, für das man keinen effizienten Algorithmus findet, so würde man sich gerne erst einmal davon überzeugen, dass es auch keinen gibt.
- Dies geschieht, indem man das Problem auf eines reduziert, für das man (bisher) keine effiziente Lösung gefunden hat.



Modell zur Beschreibung nicht effizient lösbarer Probleme

- „Nichtdeterministischer“ Algorithmus
- Prüft Akzeptanz von Eingabe x , d.h. ob $x \in L$.
- Verwendet „normale“ Anweisungen
- Zusätzlich gibt es eine Verzweigungsanweisung:
 - Jede Verzweigung teilt einen Programmpfad in mehrere auf.
 - Jeder Programmpfad liefert als Ergebnis 1 oder 0.
- An der Verzweigung angekommen, wird einer der Pfade gewählt.
- Der Algorithmus terminiert, sobald Eingabe x akzeptiert.
 - Das ist der Fall, wenn es eine Folge von Verzweigungsentscheidungen gibt mit Ergebnis 1.
 - Indeterminismus: Unterschiedliche Anordnung der Verzweigungen.
- Es wird nicht verlangt, dass bei $x \notin L$ ein Ergebnis zustande kommt.



Unterschied:

- Ein deterministischer Algorithmus entscheidet für jedes beliebige Wort der Länge n nach spätestens $O(f(n))$ Schritten, ob es in der akzeptierten Sprache ist oder nicht.
- Beispiel : Gibt es für die Formel $(a \vee b) \wedge (\neg a \vee \neg b)$ eine erfüllende Belegung der Variablen?

```
for (int a=0;a<=1,a++)  
    for (int b=0;a<=1,a++)  
        if ((a || b) && (!a || !b))  
            return 1;  
return 0;
```

Aufwand für n
Variablen = $O(2^n)$

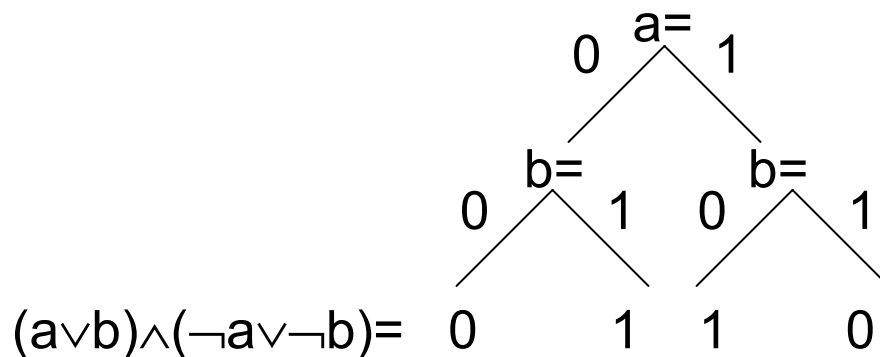
Konstruiert systematisch alle
Belegungen x bis $x \in L$



Unterschied (Forts.):

- Ein nichtdeterministischer Algorithmus stellt für jedes Wort der akzeptierten Sprache in $O(f(n))$ Schritten fest, dass es in der akzeptierten Sprache ist.

Beispiel : Gibt es für die Formel $(a \vee b) \wedge (\neg a \vee \neg b)$ eine erfüllende Belegung der Variablen?



Aufwand als Länge
eines Berechnungs-
pfades mit n Variablen
 $= O(n)$

Definition:

Laufzeit eines nichtdeterministischen Algorithmus für eine Eingabe $x \in L$ ist die Anzahl von Schritten, die die kürzeste akzeptierte Berechnung durchläuft.



Die Komplexitätsklasse NP

- Zur Klasse NP gehören die Probleme, die mit Hilfe eines nichtdeterministischen Algorithmus mit Aufwand $O(\text{POLY}(n))$ gelöst werden können.
- NP-hart (NP-hard): Ein Problem, auf das man jedes Problem aus NP reduzieren kann.
- NP-vollständig (NP-complete): Ein NP-hartes Problem, das selbst auch in NP ist.

Also:

Ist ein Lösungsvorschlag gegeben, kann mit einem deterministischen Algorithmus mit polynomialem Aufwand überprüft werden, ob dieser eine gültige Lösung ist.



Beispiel: Handlungsreisender (traveling-salesman problem)

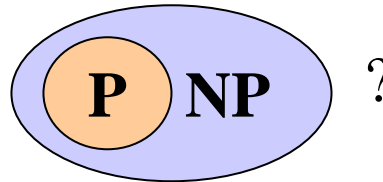
- Gegeben sei eine Karte mit n Städten.
- Ein Handlungsreisender benötigt eine Reiseroute durch alle n Städte, bei der er jede Stadt genau einmal besucht und die in der Stadt endet, in der sie begann.
- Aufgabe: Finde diejenige Route mit der kürzesten Gesamtstrecke.

Das Problem des Handlungsreisenden ist NP-vollständig.

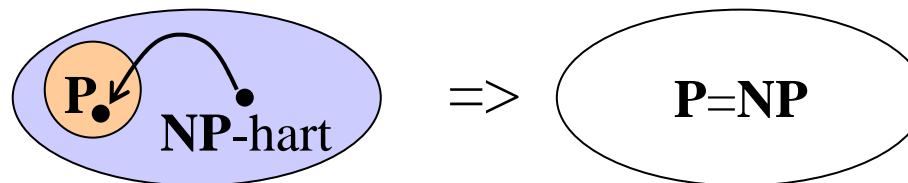


Die Komplexitätsklassen P und NP:

- Offensichtlich gilt: $P \subseteq NP$
- Die Frage, ob $P=NP$ gilt, ist eine der großen ungelösten Fragen in der theoretischen Informatik, seit sie 1971 aufgeworfen wurde.
- Ruhm und Ehre für den, der beweist, dass $P \subset NP$!



- Wenn jemals ein polynomialer Algorithmus für ein NP-vollständiges Problem gefunden wird, dann wäre das der Beweis für $NP=P$.



Beweis der NP-Vollständigkeit

- Bei (mindestens) einem Problem muss die NP-Vollständigkeit „von Hand“ nachgewiesen werden.
- Der Beweis der NP-Vollständigkeit anderer Probleme erfolgt durch Reduktion auf ein bekanntes NP-Problem.
⇒ Informatik III
- Angenommen es gibt ein NP-vollständiges Problem. Würde man nun dafür nachträglich eine effiziente Lösung finden, so gäbe es nur effizient lösbare Probleme ($NP=P$).

