

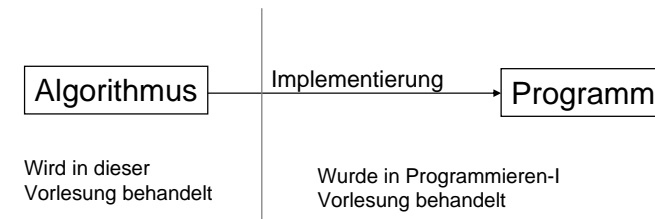
## 4. Algorithmen

1. Informatik: Eine Übersicht
2. Objektorientierte Modellierung
3. Logik (Einführung / Grundlagen)
4. Algorithmen
  - 4.1 Einführung
  - 4.2 Eigenschaften
  - 4.3 Komplexität
5. Entwurfsmethoden für Algorithmen
6. Sortieralgorithmen
7. Dynamische Datenstrukturen

## 4.1 Einführung

**Definition von Algorithmus** (zunächst nur eine vage Charakterisierung später genauere Beschreibung)

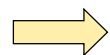
„Ein **Algorithmus** ist ein exaktes Verfahren zur Lösung eines Problems“



## 4.1 Einführung

### Bestandteile eines Algorithmus:

- **Objekte:** Darauf soll eine Wirkung ausgeübt werden. Sie können abstrakter oder auch konkreter Natur sein.
- **Handlungen:** Wirken – in einer bestimmten Reihenfolge ausgeführt - in gewünschter Weise auf die Objekte ein.



Zustandsänderungen an Objekten

## 4.1 Einführung

### Beispiel: Auswechseln eines Reifens am Auto

1. Löse die Radmuttern.
2. Hebe den Wagen an.
3. Schraube die Radmuttern ab.
4. Tausche das Rad gegen ein anderes Rad aus.
5. Schraube die Radmuttern an.
6. Setze den Wagen ab.
7. Ziehe die Radmuttern fest.

Die Abarbeitung der Schritte (1) und (2) könnte auch vertauscht werden.

## 4.1 Einführung

### Beobachtungen:

- Anwender des Algorithmus muss jeden einzelnen Schritt verstehen und ausführen können
- Verfahren enthält Folge von Handlungen in fester Reihenfolge ODER über Reihenfolge wird erst während der Ausführung des Algorithmus entschieden (notwendig: entsprechende Sprachkonstrukte)
- Abarbeitungsreihenfolge mancher Einzelschritte nicht von Bedeutung.
- Ausführungsorgan / Prozessor muss über gewisse elementare Fähigkeiten (**Elementaroperationen**) verfügen

## 4.1 Einführung

### Begriff des Algorithmus in der Informatik

- Einzelne Handlungen werden „computerverständlich“ formuliert.
  - (Halb)-formale Darstellung für Entwurf und Bearbeitung von Algorithmen
- Beteiligte Objekte werden „computerverständlich“ formuliert, indem man ihre Eigenschaften durch Daten beschreibt.
  - Beispiel: Eine einzelne **Person** wird durch **Größen** wie Name, Alter, Wohnort, ... beschrieben.

## 4. Algorithmen

1. Informatik: Eine Übersicht
2. Objektorientierte Modellierung
3. Logik (Einführung / Grundlagen)
4. Algorithmen
  - 4.1 Einführung
  - 4.2 Eigenschaften
  - 4.3 Komplexität
5. Entwurfsmethoden für Algorithmen
6. Sortieralgorithmen
7. Dynamische Datenstrukturen

## 4.2 Eigenschaften

### Denkbare Klassifizierung

- **Problemunabhängige Eigenschaften:** beziehen sich ausschließlich auf den betrachteten Algorithmus, nicht jedoch auf das Problem oder dessen Spezifikation (z.B. Endlichkeit, Determiniertheit, Rekursivität)
- **Problembezogene Eigenschaften:** sind dagegen für einen Algorithmus in Bezug auf die entsprechende Spezifikation erklärt. (z.B. Korrektheit, Effizienz)

## 4.2 Eigenschaften

### Endlichkeit

**Erste Endlichkeitsbedingung (E1):** Ein Algorithmus muss endlich beschreibbar sein, d.h. durch einen endlichen Text formulierbar.

- unmittelbar einleuchtende Forderung für jeden Algorithmus
- bezieht sich auf **Beschreibung**, nicht auf Ausführung
- Beschreibung besteht aus **endlich** vielen elementaren Operationen; zur Ausführungszeit können jedoch beliebig viele neue Operationssequenzen durchlaufen werden (z.B. durch rekursive Prozeduraufrufe), die eventuell dazu führen, dass der Algorithmus nicht terminiert
- endliche Beschreibbarkeit hängt von der zur Verfügung stehenden Sprache ab, also von den **Ablaufstrukturen** und **Elementaroperationen**

## 4.2 Eigenschaften

### ■ Beispiel

Eine rationale Zahl  $a$  soll mit einer Zahl  $n \in \mathbb{IN}_0$  multipliziert werden

**Eingabe:**  $a$  rational,  $n \in \mathbb{IN}_0$

**Ausgabe:**  $x = n * a$ .

**Rahmenbedingungen:**

- **Elementaroperationen:** Zuweisung ( $:=$ ), Addition ( $+$ ), Subtraktion ( $-$ ), Gleichheit ( $=$ )

**Ablaufstrukturen:** Sequenz, Schleife (WHILE)

➔ Algorithmus:

```
BEGIN
  Eingabe: a, n;
  x := 0;
  WHILE n ≠ 0 DO
    x := x + a;
    n := n - 1
  END (* WHILE *);
  Ausgabe: x;
END.
```

## 4.2 Eigenschaften

**Elementaroperationen:** wie zuvor

**Ablaufstrukturen:** Sequenz, Verzweigung (IF-THEN-ELSE)

```
BEGIN
  Eingabe: a, n;
  x := 0;
  IF n = 0 THEN
    Ausgabe: x
  ELSE n := n - 1;
    x := x + a;
    IF n = 0 THEN
      Ausgabe: x
    ELSE n := n - 1;
      x := x + a;
      IF n = 0 THEN
        ...
      
```



Endliche Beschreibung mit vorhandenen Ablaufstrukturen nicht möglich!

## 4.2 Eigenschaften

Für den praktischen Umgang mit Algorithmen ist Forderung **(E1)** nicht ausreichend.

**Zweite Endlichkeitsbedingung (E2):** Ein Algorithmus soll in endlicher Zeit ausführbar sein. D.h. für jede erlaubte Eingabe terminiert der Algorithmus nach endlicher Ausführungszeit.

## 4.2 Eigenschaften

### ■ Beispiel Gegebener Algorithmus

```
BEGIN
  Eingabe: n;
  REPEAT
    n := n + 1
  UNTIL n = 50;
  Ausgabe: "fertig";
END.
```

Algorithmus terminiert nur für die Eingaben 1, 2, ..., 49, dagegen nicht für größere Zahlen.

➔ Forderung (E2) nicht erfüllt.

13

## 4.2 Eigenschaften

Folgerungen aus (E2):

➔ Jede Elementaroperation muss in endlicher Zeit ausführbar sein

### ■ Beispiel nicht zulässige Elementaroperation

$\sqrt{n}$  (nicht für jedes n in endlicher Zeit ausführbar)

➔ Beteiligte Größen müssen endlich beschreibbar sein

### ■ Beispiel nicht zulässige Darstellung einer Größe

$1/3 = 0.333333333\dots$

- entweder: hinreichend genaue Näherung
- oder: Zahlenpaar (1, 3) wird benutzt (Interpretation als Bruch)

14

## 4.2 Eigenschaften

**Determiniertheit** (Globale Eindeutigkeit):

Ein Algorithmus heißt **determiniert**, falls er eine eindeutige Abhängigkeit der Ausgabedaten von den Eingabedaten garantiert.

**Determinismus** (Lokale Eindeutigkeit):

Ein Algorithmus heißt **deterministisch**, falls die Wirkung bzw. das Ergebnis jeder einzelnen Anweisung eindeutig ist und an jeder einzelnen Stelle des Ablaufs festliegt, welcher Schritt als nächster auszuführen ist.

➔ Jeder deterministische Algorithmus ist determiniert

15

## 4.2 Eigenschaften

Ursachen für nicht-deterministisches Verhalten von Algorithmen:

(1) Nicht-Determinismus in den **Elementaroperationen**,

d.h. die Wirkung bzw. das Ergebnis einer Operation ist nicht eindeutig festgelegt.

(2) Nicht-Determinismus in den **Ablaufstrukturen**,

d.h. die Ausführungsreihenfolge von Anweisungen ist nicht eindeutig festgelegt.

16

## 4.2 Eigenschaften

### ■ Beispiel Binäres Suchen in einer sortierten Folge (nicht-deterministisch)

Mitte der betrachteten Teilfolge wird nicht berechnet, sondern beliebig aus  $\{Li+1, \dots, Re-1\}$  ausgewählt

```
ALGORITHMUS bin_suche_ndet;
BEGIN
  Eingabe:  $k_1, \dots, k_n, k; k \in \{k_1, \dots, k_n\}$ 
  und  $n >= 1$ ;

   $Li := 0$ ;
   $Re := n + 1$ ;
  WHILE  $Li < Re - 1$  DO
    "Wähle  $M \in \{Li+1, \dots, Re-1\}$ ";
    IF  $k < k_M$  THEN
       $Re := M$ 
    ELSE  $Li := M$ 
    END (* IF *)
  END (* WHILE *);
  Ausgabe:  $Re$ 
END;
```

- Algorithmus ist **nicht-deterministisch** („Auswahl“ ist elementare Operation ohne eindeutig bestimmtes Ergebnis)
- Algorithmus ist **determiniert** (Ausgabe hängt in eindeutiger Weise von Eingabe ab)

17

## 4.2 Eigenschaften

Determiniertheit als Grundeigenschaft eines Algorithmus oft gefordert; allerdings gibt die Spezifikation manchmal einen gewissen Spielraum, z. B.

- ➔ **Gegeben** sei der Radius  $r$  eines Kreises.
- ➔ **Gesucht** ist der Flächeninhalt des Kreises mit einer Abweichung von höchstens  $\pm 0.00025$

d. h. Näherungslösungen möglich

18

## 4.2 Eigenschaften

### Rekursivität

### Beispiele

- Rückkopplung bei Schallübertragung/-verstärkung
- Spiegelbild zwischen zwei Spiegeln
- "Bild vom Bild" (Kamera, die Monitor mit eigenem Bild aufnimmt)
- Fakultätsfunktion:
  - $0! = 1$
  - $n! = n * (n-1)!$
- Fibonacci-Folge  $fib(0), fib(1), fib(2), \dots$ :

$$fib(0) = 0$$

$$fib(1) = 1$$

$$fib(n) = fib(n-1) + fib(n-2)$$

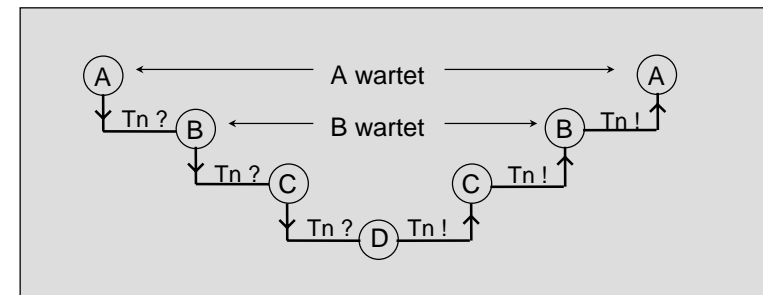
19

## 4.2 Eigenschaften

### "Erfragen einer Telefonnummer"



- A ruft B an und fragt nach Telefonnummer  $T_n$ ;
- B kennt  $T_n$  nicht, weiß aber dass C sie kennt;
- B ruft C an und fragt nach  $T_n$ ;
- C hat sein Notizbuch bei D vergessen;
- C ruft D an und fragt nach  $T_n$ ;
- D teilt C, C teilt B und B teilt A die Telefonnummer mit.



20

## 4.2 Eigenschaften

Definition:

Ein Algorithmus heißt **rekursiv**, wenn er sich selbst wieder benutzt.

- **direkte Rekursion:**

Algorithmus ruft sich selbst wieder auf

- **indirekte Rekursion:**

Man hat mehrere Algorithmen  $A_1, \dots, A_n$  ( $n > 1$ ):

$A_1$  ruft  $A_2$  auf,

$A_2$  ruft  $A_3$  auf

...

$A_{n-1}$  ruft  $A_n$  auf,

$A_n$  ruft  $A_1$  auf.

## 4.2 Eigenschaften

Schema eines direkt rekursiven Algorithmus A:

EP: Eingabeparameter von A

```
IF "Abbruchbedingung erfüllt" THEN
  (* z.B. EP minimal *)

  "direkte Lösung"
  (* ggf. mit weiterem Algorithmus B *)

ELSE "stelle augenblickliche Aufgabe zurück";

  "führe A für modifizierte Parameter EP aus";

  "berechne endgültige Lösung"

END (* IF *)
```

## 4.2 Eigenschaften

■ Beispiel

**Direkte Rekursion: Binäres Suchen (rekursiv)**

```
ALGORITHMUS bin_suche_rek (Li, Re);
(* durchsucht  $k_{Li+1}, \dots, k_{Re-1}$  nach Schlüsselwert  $k$  *)

BEGIN
  IF (Li < Re - 1) THEN
    M := (Li + Re) DIV 2;
    IF (k <  $k_M$ ) THEN
      RETURN bin_suche_rek (Li, M)
    ELSE RETURN bin_suche_rek (M, Re)
  END (* IF *)
  ELSE RETURN Re - 1
  END (* IF *)
END.
```

**Erster Aufruf:** bin\_suche\_rek (0, n+1)

## 4.2 Eigenschaften

■ Beispiel

**Indirekte Rekursion:  
Test, ob Zahl gerade oder ungerade ist**

```
ALGORITHMUS ungerade (n);

BEGIN
  IF n = 0 THEN
    RETURN FALSE
  ELSE RETURN gerade(n-1)
  END (* IF *)
END;

ALGORITHMUS gerade (n);

BEGIN
  IF n = 0 THEN
    RETURN TRUE
  ELSE RETURN ungerade(n-1)
  END (* IF *)
END;
```

## 4.2 Eigenschaften

### ■ Beispiel Fibonacci-Zahlen

a) rekursiv

```

ALGORITHMUS fibo (n);
BEGIN
  IF n = 0 THEN
    RETURN 0
  ELSE IF n = 1 THEN
    RETURN 1
  ELSE RETURN fibo(n-1) + fibo(n-2)
  END (* IF *)
END (* IF *)
END;
    
```

25

## 4.2 Eigenschaften

b) iterativ

```

ALGORITHMUS fibo2 (n);
BEGIN
  x := 0; y := 1;
  FOR i := 2 TO n DO
    hilf := y;
    y := x + y;
    x := hilf;
  END (* FOR *);
  IF n = 0 THEN
    RETURN x
  ELSE RETURN y
  END (* IF *);
END;
    
```

- rekursiver Algorithmus oft kürzer als nicht-rekursiver
- rekursiver Algorithmus **nicht** zwangsläufig effizienter

26

## 4.2 Eigenschaften

### Parallelität

- **ein Prozessor:** streng sequentielle Ausführung
- **mehrere Prozessoren:** gleichzeitige Ausführung verschiedener Teile eines Algorithmus eventuell möglich

sequentiell	parallel	
<i>Prozessor 1:</i> x := 3; y := 4;	<i>Prozessor 1:</i> x := 3;	<i>Prozessor 2:</i> y := 4;

27

## 4.2 Eigenschaften

- ### ■ Beispiel sequentielle und parallele Verarbeitung einer Anweisungsfolge mit unterschiedlichen Ergebnissen

sequentiell	parallel	
<i>Prozessor 1:</i> x := 3; y := x;	<i>Prozessor 1:</i> x := 3;	<i>Prozessor 2:</i> y := x;

Ein Algorithmus heißt **parallel**, wenn er so in Teilaufgaben aufgeteilt ist, dass diese gleichzeitig durch verschiedene Prozessoren ausgeführt werden können.

28

## 4.2 Eigenschaften

Einschränkung der Parallelverarbeitung durch:

- **Datenabhängigkeit:**

Anweisung  $a_j$  kann erst nach Anweisung  $a_i$  ausgeführt werden, da  $a_j$  Daten benötigt, die von  $a_i$  berechnet oder verändert werden.

- **Prozedurale Abhängigkeit:**

Es muss erst entschieden werden, wohin sich ein Programm verzweigt, bevor die Anweisungen, die auf die Verzweigungen folgen, den zur Verfügung stehenden Prozessoren zugeordnet und von ihnen ausgeführt werden können.

## 4.2 Eigenschaften

Vorgehensweisen, um parallel ausführbare Teile zu finden und festzulegen:

- Auffinden von Datenabhängigkeiten und parallel auszuführenden Teilen durch:

- **Datenflussgraphen**

(Knoten entsprechen Operatoren, Pfeile stellen Datenfluss dar)

- **Präzedenzgraphen**

(mögliche Ausführungsreihenfolgen der Operationen)

- Programmierer gibt parallel auszuführende Teile explizit an  
→ Programmiersprache muss entsprechende **Schlüsselwörter** zur Verfügung stellen, z. B.

PARBEGIN

$a_1; a_2; \dots; a_n$

PAREND;

## 4.2 Eigenschaften

- **Beispiel**

**Problem beim gleichzeitigen Zugriff auf dieselbe Größe**

Verschiedene Buchungen einer Bank können gleichzeitig auf verschiedenen Prozessoren durchgeführt werden.

**Algorithmus** für einzelne Buchung:

- (V1) "Lese vom Beleg Kontonummer  $K$  und Betrag  $B$ ";
- (V2) "Lese Kontostand  $S$  von Konto  $K$  und weise ihn  $X$  zu";
- (V3)  $X := X + B$ ;
- (V4) "Trage  $X$  als neuen Kontostand von  $K$  ein".

Bei zwei verschiedenen Einzahlungen von 1000,- und 5000,- für das Konto 3089:

Prozessor 1:	Prozessor 2:
$K=3089; B=1000$ (V1)	$K=3089; B=5000$ (V1)
$X := S$ ; (V2)	$X := S$ ; (V2)
$X := X + B$ ; (V3)	$X := X + B$ ; (V3)
$S := X$ ; (V4)	$S := X$ ; (V4)

Was ist der resultierende Kontostand?

## 4.2 Eigenschaften

Unter **Universalität** versteht man die Forderung, dass ein Algorithmus nicht nur eine konkrete Ausprägung eines Problems, sondern eine möglichst allgemeine **Problemklasse** löst.

- **Beispiel**

**Sortieralgorithmus**

- soll nicht nur konkrete Zahlenfolge, z. B. 5, 17, 2, 45, 67, 30, 12, 9, 3, sondern beliebige Zahlenfolgen endlicher Länge sortieren
- **universeller:** nicht nur Zahlenfolgen, sondern beliebige endliche Folgen (totale Ordnung vorausgesetzt)  
→ Wiederverwendbarkeit

## 4. Algorithmen

1. Informatik: Eine Übersicht
2. Objektorientierte Modellierung
3. Logik (Einführung / Grundlagen)
4. Algorithmen
  - 4.1 Einführung
  - 4.2 Eigenschaften
  - 4.3 Komplexität
5. Entwurfsmethoden für Algorithmen
6. Sortieralgorithmen
7. Dynamische Datenstrukturen

## 4.3 Komplexität

Wichtiges Merkmal von Algorithmen:

➔ **Komplexität**, d.h.

- Wieviel Zeit
- Wieviel Speicherplatz

benötigt ein Algorithmus zur Ausführung?

**Ziel** bei der Entwicklung von Algorithmen:

- möglichst effiziente Algorithmen,  
d.h. geringer Zeit- und Speicherplatzverbrauch

*Im Folgenden betrachtet: Zeitkomplexität / Verbrauch an Rechenzeit*

## 4.3 Komplexität

**Einflussfaktoren:**

- (a) Anzahl und Größe der Eingabedaten ("Problemgröße")
- (b) der Algorithmus selbst (insbesondere Art und Zusammensetzung seiner Ablaufstrukturen)
- (c) Eigenschaften der zur Ausführung eingesetzten Rechanlage (z.B. Schnelligkeit, Befehlssatz )

## 4.3 Komplexität

■ **Beispiele** zu (a) **Anzahl und Größe der Eingabedaten**

Sortieren von Zahlenfolgen:  
Die Größe der zu sortierenden Folge  
(**Problemgröße**) hat großen Einfluss.

➔ *Sortieren einer Folge von 5 Zahlen ist offensichtlich  
schneller als Sortieren einer Folge von 1000 Zahlen !*

Problemgröße kann sein:

- Länge einer Folge (z.B. Shortest Path, Job-Scheduling)
- Größe einer Zahl (z.B. Primzahlfaktorisation)

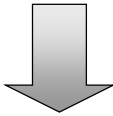
## 4.3 Komplexität

### zu (b) Algorithmus selbst

betrachte folgende Teile von Algorithmen:

```
Eingabe: n;  
i := 0;  
FOR k:=1 TO n DO  
  i:= i + 1  
END (* FOR *);  
Ausgabe: i;
```

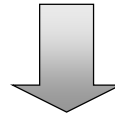
*FOR-Schleife wird  
n-mal durchlaufen.*



Anweisung  $i:=i+1$  wird  
n-mal ausgeführt.

```
Eingabe: n;  
i := 0;  
FOR k:=1 TO n DO  
  FOR j:=1 TO k DO  
    i:= i + 1  
  END (* FOR *);  
END (* FOR *);  
Ausgabe: i;
```

*Verschachtelung zweier  
FOR-Schleifen.*



Anweisung  $i:=i+1$  wird  
 $n*(n+1)/2$  - mal ausgeführt.

## 4.3 Komplexität

### zu (c) Art der Rechenanlage

- Eigenschaften der eingesetzten Rechenanlage nehmen Einfluss auf die Ausführungszeit
- Bei Komplexitätsberechnungen von Algorithmen wird jedoch von diesem Parameter abstrahiert
- Betrachtung einer abstrakten Maschine, die die Operationen von modernen Rechnern simulieren und prinzipiell die gleichen Berechnungen wie diese durchführen können.  
z.B. - **Random-Access-Maschine (RAM)**  
- **Turing-Maschine** (vgl. Grundlagen II)
- Abstrakte Maschinen haben einen einfachen Befehlssatz.
- Messung der Komplexität von Algorithmen:



*Zählen der ausgeführten elementaren Rechenschritte*

## 4.3 Komplexität

### Idealisierende Annahmen:

- Jede Elementaroperation (Zuweisung, Addition, Subtraktion, Multiplikation, Vergleich zweier Zahlen, etc.) benötigt genau einen Rechenschritt
- Aufwendige Operationen (z.B. Matrizenmultiplikation) nicht verfügbar
- Einheitliche Größe der betrachteten Daten, Darstellung in jeweils einer Speicherzelle

## 4.3 Komplexität

### Fragestellungen:

Welchen Zeitbedarf hat ein **konkreter** Algorithmus  $A$  zur Lösung eines Problems  $P$  in Abhängigkeit von der Größe  $n$  der Eingabedaten?



*Anzahl der Elementaroperationen (**ELOP**), die nötig sind, um mit  $A$  eine Problemausprägung der Größe  $n$  zu lösen*

Welchen Zeitbedarf hat ein **optimaler** Algorithmus zur Lösung von  $P$ ?  
Beispiel: „Suchen“: sequentielles Suchen, binäres Suchen



*Frage meist nur schwer zu beantworten (oft ist zu Algorithmus  $A$ , der  $P$  löst, kein besserer Algorithmus bekannt, aber auch nicht bewiesen, dass es keinen besseren gibt.)*

*Wir beschäftigen uns im Folgenden mit dem konkreten Zeitbedarf eines Algorithmus!*

## 4.3 Komplexität

Geg.: Algorithmus A

Eingabe e für spezielle Problemausprägung der Größe n  
Die Menge aller möglichen Eingaben zu n:  $E_n$

$t(A; n, e)$  bezeichne die **Anzahl der** durchzuführenden **Elementaroperationen** bei der Ausführung von A bezüglich der Eingabe e.

Ferner seien:

$$T_{\min}(A; n) ::= \min\{t(A; n, e) \mid e \in E_n\}, \quad \text{"best-case"}$$

$$T_{\text{avg}}(A; n) ::= \text{avg}\{t(A; n, e) \mid e \in E_n\} \quad \text{"average-case"}$$

$$T_{\max}(A; n) ::= \max\{t(A; n, e) \mid e \in E_n\} \quad \text{"worst-case"}$$

➔ Alle drei Funktionen sind Maße für die **Zeitkomplexität** von A

## 4.3 Komplexität

Bemerkungen:

- **best-case-** bzw. **worst-case-Komplexität** beschreiben Zeitbedarf im besten bzw. schlechtesten Fall
- **average-case-Komplexität** mittelt den Zeitbedarf über alle möglichen Eingaben der Größe n

average-case-Komplexität ist nur schwer zu bestimmen (nur realistisch, wenn alle Problemausprägungen gleich wahrscheinlich)

- Analog **Speicherkomplexität S(A; n)**

## 4.3 Komplexität

■ Beispiel **Berechnung der Summe s von n Zahlen**

**Eingabe:**  $x_1, x_2, \dots, x_n$ , mit  $n \in \mathbb{IN}$  und  $x_i \in \mathbb{IN}$  für alle  $i \in \{1, \dots, n\}$

**Ausgabe:**  $s = \sum_{i=1}^n x_i$  **Elementaroperationen:** Zuweisung (:=), Vergleich (<), Addition (+), ...

Für jede Zahlenfolge der Länge n lässt sich die Anzahl der ELOPs wie folgt bestimmen:

Eingabe: $x_1, \dots, x_n$		
$s := 0; j := 0;$	...2	➔
while $j < n$	...n + 1	
do		
$j := j + 1;$	...2n	
$s := s + x_j;$	...2n	
Ausgabe: s;		

$$\begin{aligned} T_{\min}(A; n) &= T_{\max}(A; n) \\ &= T_{\text{avg}}(A; n) \\ &= 5n + 3 \end{aligned}$$

## 4.3 Komplexität

Für größere Beispiele ist es zu aufwendig, für jede mögliche Eingabe die Anzahl der ELOPs zu bestimmen.

➔ Angabe des Zeitbedarfs i.a. nur **größenordnungsmäßig** (in Abhängigkeit von der Größe der Eingabe)

➔ Beschreibung des Wachstums des Zeitbedarfs, sogenannte **"Groß-Oh-Notation"**

## 4.3 Komplexität

### Definition:

Es sei  $P$  die Menge aller Funktionen  $f: \mathbb{N} \rightarrow \mathbb{R}$ , und es sei  $g \in P$ .  
Dann wird festgelegt:

$$(a) \quad O(g) ::= \{ f \in P \mid \exists n_0 \in \mathbb{N} \exists c > 0 \forall n > n_0: |f(n)| \leq c \cdot |g(n)| \}$$

Für  $f \in O(g)$  sagt man: "f wächst höchstens so stark wie g", bzw.  
"f hat höchstens die Ordnung g", bzw.  
"f ist aus Groß-Oh von g".

$$(b) \quad \Omega(g) ::= \{ f \in P \mid \exists n_0 \in \mathbb{N} \exists c > 0 \forall n > n_0: |f(n)| \geq c \cdot |g(n)| \}$$

Für  $f \in \Omega(g)$  sagt man: "f wächst mindestens so stark wie g", bzw.  
"f hat mindestens die Ordnung g".

## 4.3 Komplexität

$$(c) \quad \Theta(g) ::= O(g) \cap \Omega(g).$$

Für  $f \in \Theta(g)$  sagt man: "f wächst genauso stark wie g", bzw.  
"f hat genau die Ordnung g".

In diesem Fall gilt auch die Umkehrung  $g \in \Theta(f)$ .

➔  $O(g)$ ,  $\Omega(g)$  und  $\Theta(g)$  beschreiben also **Mengen von Funktionen**, die für große  $n$  höchstens bzw. mindestens bzw. genauso stark wachsen wie die Funktion  $g$ .

## 4.3 Komplexität

### ■ Beispiel

(a) Funktion von Beispiel: Summe von  $n$  Zahlen:  $f(n) = 5n + 3$

Es gilt:  $f \in O(g)$ , mit  $g(n) = n$

Schreibweise:  $f \in O(n)$

(b)  $f(n) = n^2 + 1000n \in O(n^2)$

denn: wähle etwa  $c = 2$ ,  $n_0 = 1000$

$$n^2 + 1000n \leq c \cdot n^2 \quad \forall n \geq n_0$$

## 4.3 Komplexität

(c) Allgemeiner: sei für  $i \in \{0, \dots, k\}$ :  $a_i \in \mathbb{R}$ ,  
für  $i \in \{0, \dots, j\}$ :  $b_i \in \mathbb{R}$ ,  
ferner  $a_k \neq 0$  und  $b_j \neq 0$ ,  
 $k, j \in \mathbb{N}_0$

Dann gilt (ohne Beweis):

$$(i) \quad \sum_{i=0}^k a_i n^i \in O\left(\sum_{i=0}^j b_i n^i\right) \Leftrightarrow k \leq j.$$

$$(ii) \quad \sum_{i=0}^k a_i n^i \in \Omega\left(\sum_{i=0}^j b_i n^i\right) \Leftrightarrow k \geq j.$$

$$(iii) \quad \sum_{i=0}^k a_i n^i \in \Theta\left(\sum_{i=0}^j b_i n^i\right) \Leftrightarrow k = j.$$

## 4.3 Komplexität

### Bemerkung:

$\Theta(\log_2 n) = \Theta(\log_b n)$ , für  $b \in \mathbb{R}^+$  bel.

denn:  $\log_b n = \log_b 2 * \log_2 n$   
 $\log_b n = c * \log_2 n$

➔ Für Logarithmen ist die Betrachtung der Basiszahl im Zusammenhang mit Wachstumsklassen irrelevant.

## 4.3 Komplexität

Welche Operation als Elementaroperationen gesehen werden, hängt vom Anwendungsfall ab.

Beim Entwurf von Schaltkreisen ist die Granularität höher (Bitoperationen als Elementaroperationen) als beim Entwurf von Anwendungssoftware (Addition als Elementaroperation)

Wir einigen uns auf folgende Elementaroperationen:

- Zuweisung
- Addition, Subtraktion, Multiplikation, Division
- Vergleich ( =, <, >, <=, >= )
- Logisches „oder“
- Logisches „und“

## 4.3 Komplexität

### ■ Beispiel Komplexität des sequentiellen Suchens

```
ALGORITHMUS seq_suche;  
BEGIN  
  Eingabe:  $k_1, \dots, k_n, k$ ;  
            $k \in \{k_1, \dots, k_n\}$ ;  
           Liste ist sortiert.  
   $i := 0$ ;  
  REPEAT  
     $i := i + 1$   
  UNTIL ( $k < k_i$ ) oder ( $i = n + 1$ );  
  Ausgabe:  $i$   
END;
```

### Eingabe:

$e ::= (k_1, k_2, \dots, k_n, k)$   
konkrete Eingabe;  
Folge der Länge  $n$

### Ausgabe:

$i ::= i(e)$ , mit  $i \in \{1, \dots, n + 1\}$

Die REPEAT-Schleife wird  $i$ -mal durchlaufen, wobei  $i$ =Position des Elementes:

außerhalb der Schleife: 1 ELOP,  
innerhalb dagegen: 6 ELOPs

➔  $t(\text{seq\_Suche}; n, e) = 6i + 1$

## 4.3 Komplexität

### Komplexität:

$$\begin{aligned} T_{\min}(\text{seq\_suche}; n) &= 7 \\ T_{\max}(\text{seq\_suche}; n) &= 6(n + 1) + 1 \\ &= 6n + 7 \\ &\approx 6n \\ T_{\text{avg}}(\text{seq\_suche}; n) &= \frac{1}{(n+1)} \sum_{i=1}^{n+1} (6i+1) \\ &= 3n + 7 \\ &\approx 3n \end{aligned}$$

Annahme für  $T_{\text{avg}}$ :

Gleichverteilung für die Position von  $k$  in der Folge  $k_1, k_2, \dots, k_n$

➔ Zeitkomplexität im Durchschnitt und im schlechtesten Fall:  $O(n)$ , d. h. Zeitbedarf hängt linear von  $n$  ab

## 4.3 Komplexität

### ■ Beispiel Komplexität des binären Suchens

```

ALGORITHMUS bin_suche;
BEGIN
  Eingabe:  $k_1, \dots, k_n, k$ ;
            $k \in \{k_1, \dots, k_n\}$ ;
           Liste ist sortiert.

  Li := 0;
  Re := n + 1;
  WHILE Li < Re - 1 DO
    M := (Li + Re) DIV 2;
    IF  $k < k_M$ 
      THEN Re := M
    ELSE Li := M
    END (* IF *)
  END (* WHILE *);
  Ausgabe: Re
END;
    
```

$$T_{\max}(\text{bin\_suche}, n) \in O(\log_2 n)$$

außerhalb der Schleife 3 ELOPs  
 Innerhalb der Schleife 7 ELOPs  
 Schleife wird höchstens  
 $\log(n)+2$  mal durchlaufen

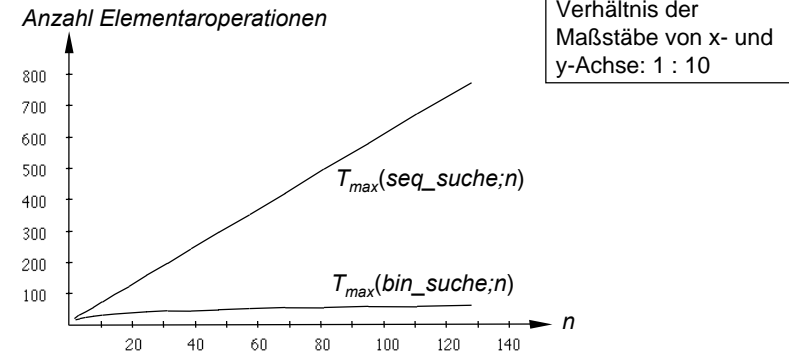
$$T_{\max}(\text{bin\_suche}; n) = 7(\log(n)+2)+3$$

#### Komplexität einer If-Anweisung:

$$T_{\max}(\text{If-Anweisung}) = T_{\max}(\text{COND}) + \max(T_{\max}(\text{THEN-Teil}), T_{\max}(\text{ELSE-Teil}))$$

## 4.3 Komplexität

### Vergleich der Komplexität von binärem und sequentiellem Suchen:



➔ Binäres Suchen effizienter als sequentielles

## 4.3 Komplexität

$n$	$T_{\max}(\text{bin\_suche}; n)$	$T_{\max}(\text{seq\_suche}; n)$
2	24	19
4	31	31
8	38	55
16	45	103
32	52	192

## 4.3 Komplexität

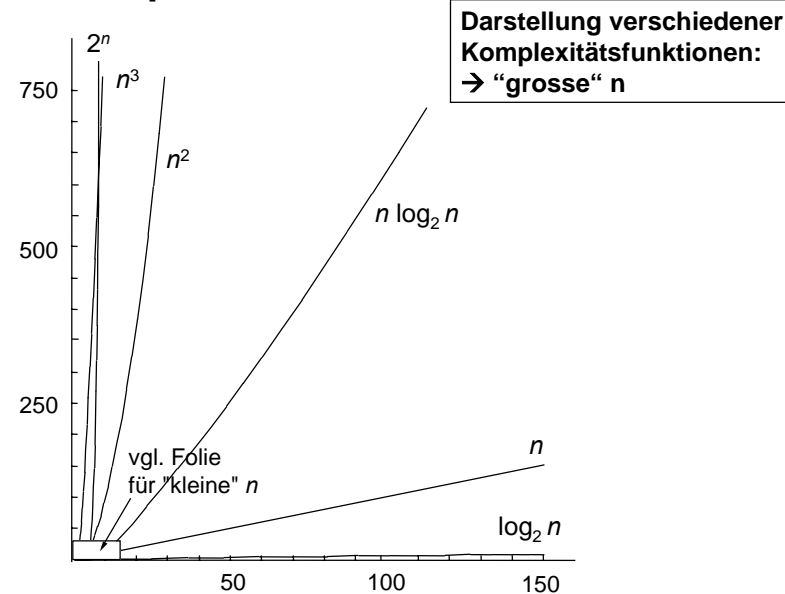
Man unterscheidet verschiedene typische Komplexitätsfunktionen für Algorithmen (**Klassen**):

- **$O(\log_2 n)$ : Logarithmische Komplexität:**  
 Laufzeit des Algorithmus wächst wesentlich schwächer als  $n$ .  
 Verdopplung von  $n$  bedeutet Anstieg der Laufzeit um additive Konstante  $\log_2 2 = 1$
- **$O(n)$ : Lineare Komplexität:**  
 Laufzeit des Algorithmus wächst proportional zu  $n$ .  
 Verdopplung von  $n$  hat (größenordnungsmäßig) Verdopplung der Laufzeit zur Folge.
- **$O(n \log_2 n)$ : Leicht überlineare Komplexität:**  
 Laufzeit des Algorithmus wächst etwas stärker als  $n$ .  
 Verdopplung von  $n$  hat etwas mehr als Verdopplung der Laufzeit zur Folge  
 (Genauer: Verdopplung von  $n$  bedeutet Multiplikation mit dem Faktor  $2 + 2 / \log_2(n)$ , der für große  $n$  gegen 2 konvergiert)

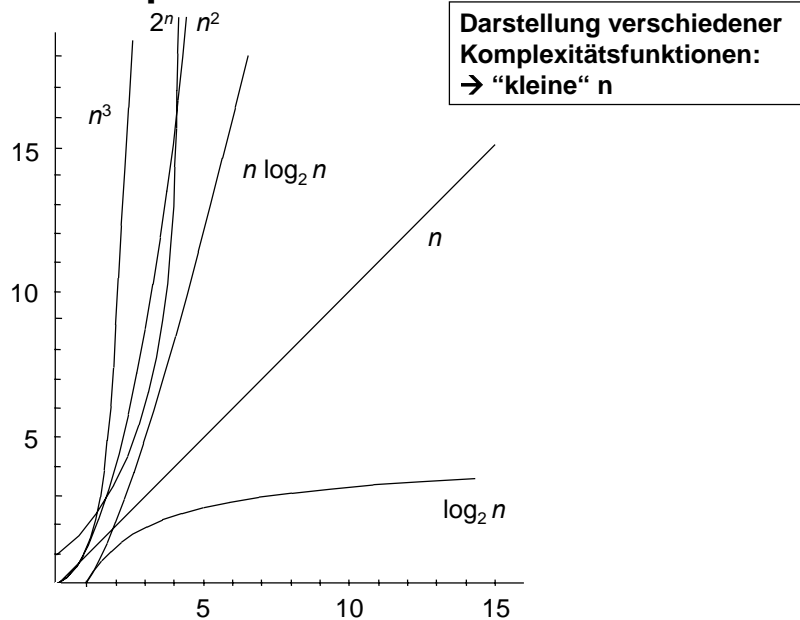
## 4.3 Komplexität

- $O(n^2)$ : **Quadratische Komplexität:**  
 Laufzeit des Algorithmus wächst wesentlich schneller als  $n$ .  
 Verdopplung von  $n$  bewirkt Vervielfachung der Laufzeit.
- $O(n^3)$ : **Kubische Komplexität:**  
 Verdopplung von  $n$  bedeutet Verachtfachung der Laufzeit  
 (für große  $n$  nicht akzeptabel).
- $O(2^n)$ : **Exponentielle Komplexität:**  
 Verdopplung von  $n$  bedeutet Quadrierung der Laufzeit  
 (für große  $n$  katastrophal).

## 4.3 Komplexität



## 4.3 Komplexität



## 4.3 Komplexität

Werte verschiedener Komplexitätsfunktionen:

	10	50	100	300	1000
$\log_2 n$	3	6	7	8	10
$n$	10	50	100	300	1000
$n \log_2 n$	33	282	656	2469	9966
$n^2$	100	2500	10000	90000	$10^6$
$n^3$	1000	125000	$10^6$	$27 * 10^6$	$10^9$
$2^n$	1024	16-stellige Zahl	31-stellige Zahl	91-stellige Zahl	302-stellige Zahl

**Zum Vergleich:**

Anzahl der Protonen im Weltall: eine ca. 126-stellige Zahl.

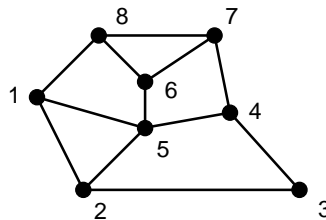


## 4.3 Komplexität

### ■ Beispiel Hamiltonscher Kreis (Hamiltonscher Zyklus)

Ein Reisender will verschiedene Städte besuchen. Zwischen einigen dieser Städte gibt es Verkehrsverbindungen. Der Reisende will sich eine Rundreise durch die Städte derart zurechtlegen, dass jede Stadt exakt einmal bereist wird. Zum Schluss will er wieder in der Ausgangsstadt ankommen.

#### Problemausprägung:



Gültige Rundreise für vorliegende Städteverbindung (Knoten bedeuten Städte, Kanten bedeuten Verkehrsverbindungen): z.B. 1, 2, 3, 4, 5, 6, 7, 8, 1

## 4.3 Komplexität

### Problemspezifikation (allgemein):

#### Eingabe:

Ein ungerichteter Graph  $G$  bestehend aus einer nichtleeren, endlichen Knotenmenge  $E$  und einer Kantenmenge  $K$ :

$$G = (E, K)$$

$$E = \{e_1, e_2, \dots, e_n\}, K \subseteq E \times E, \text{ symmetrisch.}$$

#### Ausgabe:

"Ja", falls es eine Folge  $F = e_{i_1}, e_{i_2}, \dots, e_{i_n}$  gibt, für die gilt:

- (i) Jeder Knoten ist genau einmal enthalten.
- (ii)  $(e_{i_1}, e_{i_2}), (e_{i_2}, e_{i_3}), \dots, (e_{i_{n-1}}, e_{i_n}), (e_{i_n}, e_{i_1}) \in K$

➔ Folge  $F$  in einem Graphen, die obiger Spezifikation entspricht, heißt auch **Hamiltonscher Kreis (Zyklus)**.

## 4.3 Komplexität

Folgender grobe und naive Algorithmus stellt fest, ob ein Graph einen derartigen Kreis enthält:

```
ALGORITHMUS HZ;
BEGIN
  Eingabe: E, K; (* n ::= Anzahl der Knoten *)
  gefunden := false;
  REPEAT
    "Berechne Permutation  $e_{i_1}, e_{i_2}, \dots, e_{i_n}$  der Knoten";
    IF  $\{(e_{i_1}, e_{i_2}), (e_{i_2}, e_{i_3}), \dots, (e_{i_n}, e_{i_1})\} \subseteq K$ 
      THEN gefunden := TRUE;
      Ausgabe: ja
    END (* IF *)
  UNTIL gefunden oder "alle Permutationen ausprobiert";
END.
```

## 4.3 Komplexität

- Da es zu einer Folge der Länge  $n$  genau  $n!$  Permutationen gibt, ist  $T_{\max}(HZ; n) \in O(n!)$
- Nach heutigem Kenntnisstand:  
Algorithmus lässt sich nur unwesentlich verbessern, d. h. bislang ist kein Algorithmus bekannt, der jeden Hamiltonschen Kreis in einem Graph findet und dabei mit polynomiellem Zeitaufwand auskommt.