

# Algorithmen und Datenstrukturen

Sommersemester 2021

Stefan Näher

Universität Trier

naeher@uni-trier.de

**Vorlesung 2**

8. April 2021

## 2. Binäre Suche

Wiederholung

Vorraussetzung:  $A[1..n]$  ist **aufsteigend sortiert**

d.h.  $A[i] \leq A[i + 1]$  für  $i = 1, \dots, n - 1$

Zunächst nehmen wir an, dass sogar  $A[i] < A[i + 1]$

d.h. es gibt keine Wiederholungen (allgemeiner Fall: Übung)

### Idee des Algorithmus:

Betrachte immer 2 Positionen  $\ell$  und  $r$  mit:

Falls  $x = A[i]$  für eine Position  $i$  dann gilt  $\ell \leq i \leq r$

↪ **Invariante**

## Initialisierung

$\ell \leftarrow 1$  und  $r \leftarrow n$ .

## Schritt des Algorithmus

sei  $m \leftarrow \lfloor (\ell + r)/2 \rfloor$  d.h.  $m$  ist mittlere Position in  $[\ell..r]$

Dann gibt es 3 mögliche Fälle:

- a)  $x = A[m]$  **fertig** ( $x$  gefunden)
- b)  $x < A[m]$  suche weiter im **linken** Teilfeld  $A[\ell..m-1]$  ( $r \leftarrow m-1$ )
- c)  $x > A[m]$  suche weiter im **rechten** Teilfeld  $A[m+1..r]$  ( $\ell \leftarrow m+1$ )

BINARY\_SEARCH ( $x$ )

1.  $\ell \leftarrow 1$ ;
2.  $r \leftarrow n$ ;
3. **while**  $\ell \leq r$  **do**
4.    $m \leftarrow \lfloor (\ell + r)/2 \rfloor$ ;
5.   **if**  $A[m] = x$  **then**
6.     **return**  $m$ ;
7.   **fi**
8.   **if**  $x < A[m]$
9.     **then**  $r \leftarrow m - 1$ ;
10.    **else**  $\ell \leftarrow m + 1$ ;
11.   **fi**
12. **od**
13. **return** 0;

## Korrektheit

1. Die Invariante ist stets erfüllt.

2. Termination

Die Länge des Teilfeldes  $A[\ell..r]$  wird in jedem Durchlauf der while-Schleife vermindert (sogar halbiert !)

## **Laufzeitanalyse 1** Anzahl der Iterationen

Wieviele Schritte führt BINARY\_SEARCH im schlimmsten Fall aus ?  
d.h. wenn  $x \notin A$

1. Die Länge des Intervalls  $[\ell..r]$  ist am Anfang  $n$ .
2. Bei jeder Ausführung der while-Schleife wird das Intervall mindestens halbiert.
3. Der Algorithmus terminiert spätestens, wenn dieses Intervall leer ist (Zeile 3).

### **Frage:**

Wie oft muss man eine Zahl  $n$  wiederholt durch 2 teilen bis das Resultat  $< 1$  ist.

### **Antwort:**

Dies ist eine mögliche Definition des **Logarithmus** zur Basis 2.

Also wird die while-Schleife maximal  $\log_2(n)$  mal ausgeführt.

## **Laufzeitanalyse 2** Gesamtkosten

Wir schätzen die worst-case Kosten einer Ausführung der Schleife ab:

Maximal 11 Operationen pro Schleife  
(bitte Nachzählen und evtl. korrigieren)

Die worst-case Laufzeit von BINARY\_SEARCH ist somit  $11 \cdot \log_2(n) + 3$

Falls  $\log_2(n) \geq 3$  d.h.  $n \geq 8$  ist die Laufzeit also  $12 \cdot \log_2(n)$   
d.h. sie wächst nur logarithmisch mit  $n$ .

**Eine Verdoppelung der Länge von A führt nur zu einer Erhöhung der Laufzeit um eine Konstante (nämlich 11).**

## Wie vergleicht man die Laufzeiten von Algorithmen ?

Sei z.B.

$T_1(n) = c_1 \cdot n + c_0$  die Laufzeit von Algorithmus  $A_1$  und

$T_2(n) = d_1 \cdot n^2 + d_0$  die Laufzeit von Algorithmus  $A_2$

Dann ist bei genügend großen Eingaben Algorithmus  $A_1$  irgendwann besser als  $A_2$ , unabhängig von den Konstanten  $c_0, c_1, d_0, d_1$ .

*Asymptotisch*, d.h. für genügend große  $n$ , wächst eine quadratische Funktion immer schneller als eine lineare Funktion.

~> Das Wachstum von Laufzeitfunktionen sollte unabhängig von multiplikativen und additiven Konstanten beschrieben werden.



## $\mathcal{O}$ -Notation

### Definition

Sei  $g : \mathbb{N} \longrightarrow \mathbb{R}^+$

(eine Funktion die natürliche Zahlen auf positive reelle Zahlen abbildet)

$$\mathcal{O}(g) := \{ f : \mathbb{N} \longrightarrow \mathbb{R}^+ \mid \exists c > 0, n_0 \in \mathbb{N} : \forall n \geq n_0 \quad f(n) \leq c \cdot g(n) \}$$

Schreibweise:  $f(n) \in \mathcal{O}(g)$  oder kurz:  $f = \mathcal{O}(g)$

Sprechweise:  $g$  ist eine **asymptotische obere Schranke** für  $f$

## Fortsetzung ( $\Omega$ und $\Theta$ )

$$\Omega(g) := \{ f : \mathbb{N} \longrightarrow \mathbb{R}^+ \mid \exists c > 0, n_0 \in \mathbb{N} : \forall n \geq n_0 \ f(n) \geq c \cdot g(n) \}$$

Schreibweise:  $f(n) \in \Omega(g)$  oder kurz:  $f = \Omega(g)$

Sprechweise:  $g$  ist eine **asymptotische untere Schranke** für  $f$

$$\Theta(g) := \{ f : \mathbb{N} \longrightarrow \mathbb{R}^+ \mid \exists c_1, c_2 > 0, n_0 \in \mathbb{N} : \forall n \geq n_0 \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \}$$

Schreibweise:  $f(n) \in \Theta(g)$  oder kurz:  $f = \Theta(g)$

Sprechweise:  $g$  ist eine **asymptotische scharfe Schranke** für  $f$

Leicht zu sehen:  $\Theta(g) = \mathcal{O}(g) \cap \Omega(g)$

## Einige Beispiele

$\mathcal{O}(1)$	konstanter Aufwand, unabhängig von $n$
$\mathcal{O}(n)$	linearer Aufwand (z.B. Einlesen von $n$ Zahlen)
$\mathcal{O}(n \log n)$	Aufwand guter Sortierverfahren (z.B. Quicksort)
$\mathcal{O}(n^2)$	quadratischer Aufwand
$\mathcal{O}(n^k)$	polynomialer Aufwand (bei festem $k$ )
$\mathcal{O}(2^n)$	exponentieller Aufwand
$\mathcal{O}(n!)$	Bestimmung aller Permutationen von $n$ Elementen

## Konkreter Vergleich

Annahme: 1 Schritt dauert 1  $\mu\text{s}$  = 0.000001 s

n =	10	20	30	40	50	60
n	10 $\mu\text{s}$	20 $\mu\text{s}$	30 $\mu\text{s}$	40 $\mu\text{s}$	50 $\mu\text{s}$	60 $\mu\text{s}$
$n^2$	100 $\mu\text{s}$	400 $\mu\text{s}$	900 $\mu\text{s}$	1.6 ms	2.5 ms	3.6 ms
$n^3$	1 ms	8 ms	27 ms	64 ms	125 ms	216 ms
$2^n$	1 ms	1 s	18 min	13 Tage	36 J	366 Jh
$3^n$	59 ms	58 min	6.5 J	3855 Jh	$10^8$ Jh	$10^{13}$ Jh
$n!$	3.62 s	771 Jh	$10^{16}$ Jh	$10^{32}$ Jh	$10^{49}$ Jh	$10^{66}$ Jh

## Weitere Beispiele

$$f(n) = 6n + 4 = \mathcal{O}(n)$$

**Beweis:**  $\forall n \geq 4$  gilt  $f(n) \leq 7n$

wähle  $c = 7$  und  $n_0 = 4$  in der Definition von  $\mathcal{O}(g)$ .

$$f(n) = 3n^2 + 6n + 4 = \mathcal{O}(n^2)$$

**Beweis:**  $\forall n \geq 4$  gilt  $f(n) \leq 3n^2 + 7n$  (siehe oben)

und  $\forall n \geq 7$  gilt  $3n^2 + 7n \leq 4n^2$ .

Also gilt für alle  $n \geq 7$  :  $f(n) \leq c \cdot n^2$  mit  $c = 4$ .

Verallgemeinerung: Polynom vom Grad  $d$

$$f(n) = a_d \cdot n^d + a_{d-1} \cdot n^{d-1} + \dots + a_1 \cdot n + a_0 = \mathcal{O}(n^d).$$

**Beweis:** Übungsaufgabe.

Logarithmus zu beliebiger Basis  $b$

$$f(n) = \log_b n = \mathcal{O}(\log n) \text{ (} \log n = \text{Logarithmus zur Basis 2)}.$$

**Beweis:** Übungsaufgabe.

$f(n) = c$  , wobei  $c$  eine Konstante (unabhängig von  $n$ )  
dann gilt:  $f(n) = \mathcal{O}(1)$

**Beweis:**  $f(n) \leq c \cdot 1$  für alle  $n \geq n_0 = 0$ .

## Weitere Eigenschaften

### Transitivität

Für  $op \in \{\mathcal{O}, \Omega, \Theta\}$  gilt:

Falls  $f \in op(g)$  und  $g \in op(h)$  dann gilt  $f \in op(h)$ .

### Reflexivität

Für  $op \in \{\mathcal{O}, \Omega, \Theta\}$  gilt:  $f \in op(f)$ .

### Symmetrie

$f = \Theta(g)$  genau dann, wenn  $g = \Theta(f)$ .

## Lineare und binäre Suche

LINEAR\_SEARCH auf einem Feld der Länge  $n$  hat Laufzeit  $\mathcal{O}(n)$ .

BINARY\_SEARCH auf einem Feld der Länge  $n$  hat Laufzeit  $\mathcal{O}(\log n)$ .



## Rekursive Algorithmen / Prozeduren

**Prozeduren** oder **Funktionen** kann man bei ihrem Aufruf **Parameter** übergeben und sie können bei ihrer Beendigung Werte (Resultate) zurückgeben.

Prozeduren und Algorithmen können “sich selbst” als Unterprogramm aufrufen. Eine solche Prozedur oder einen solchen Algorithmus nennt man **rekursiv**.

## **Rekursion** am Beispiel: Sortieren durch Mischen (Mergesort)

Gegeben Sei ein Feld  $A[1..n]$  von Zahlen. Wir möchten das Feld aufsteigend sortieren, d.h. die Elemente des Feldes so umordnen, dass gilt  $A[i] \leq A[i+1]$  für  $i = 1, \dots, n-1$ .

Anmerkung: In der Übung werden wir ein einfaches aber nicht effizientes Verfahren behandeln, das auf der Suche nach dem Maximum bzw. Minimum basiert.

Ein rekursiver Algorithmus muss im Allgemeinen in der Lage sein auf Teilproblemen des ursprünglichen Problems zu arbeiten.

MERGESORT( $\ell, r$ ) sortiert das Teilfeld  $A[\ell..r]$

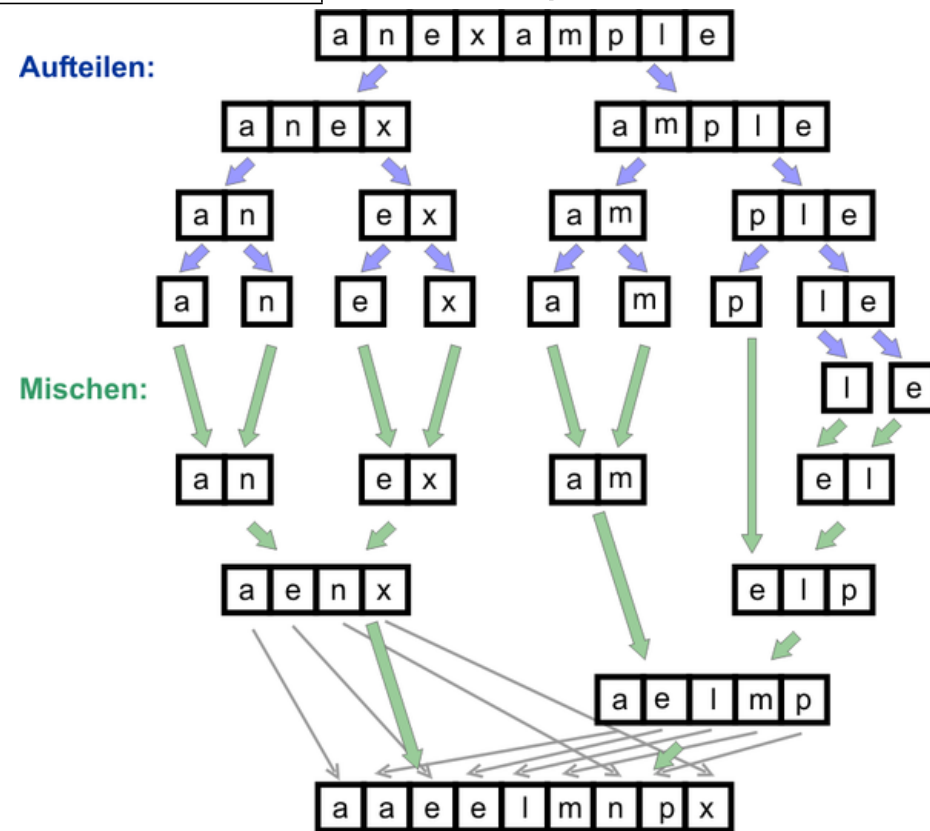
Dann löst der Aufruf MERGESORT(1,  $n$ ) das Gesamtproblem.

## Die rekursive Funktion MERGESORT

1. MERGESORT( $\ell, r$ )
2. **if**  $\ell \geq r$  **then return** ; // Verankerung: Feld der Länge  $\leq 1$  (nichts zu tun)
3.  $m \leftarrow \lfloor (\ell + r)/2 \rfloor$  //  $m$  = mittlere Position in  $A[\ell..r]$
4. MERGESORT( $\ell, m$ );
5. MERGESORT( $m + 1, r$ );
6. **MERGE**( $\ell, m, r$ );

Die eigentliche Arbeit steckt in der Prozedur **MERGE**. Sie ‘mischt’ die beiden sortierten Teilfelder  $A[\ell, m]$  und  $A[m + 1, r]$  zu einem sortierten Feld  $A[\ell, r]$ .

## Sortieren durch Mischen: Ein Beispiel



**Die Arbeitsweise von MERGE** [hier Skizze einfügen]

**Übungsaufgabe:** Ausformulierung des Algorithmus im PseudoCode.