

# **Algorithms & Data Structures Solutions - SoSe 24**

Igor Dimitrov

2024-04-22

# Table of contents

<b>Preface</b>	<b>3</b>
<b>1 Blatt 1</b>	<b>4</b>
1.1 Aufgabe 2 . . . . .	4
1.2 Aufgabe 3 . . . . .	5
a) . . . . .	5
b) . . . . .	6
c) <b>Falsch:</b> . . . . .	6
d) <b>Falsch:</b> . . . . .	6
e) <b>Falsch:</b> . . . . .	6
f) . . . . .	7
1.3 Auafgabe 4 . . . . .	7
a) . . . . .	7
b) . . . . .	7
c) . . . . .	8
d) . . . . .	9
<b>2 Blatt 2</b>	<b>11</b>
2.1 Aufgabe 1 . . . . .	11
1 . . . . .	11
2 . . . . .	11
3 . . . . .	12
2.2 Aufgabe 2 . . . . .	13
a) . . . . .	13
b) . . . . .	13
c) . . . . .	13
d) . . . . .	13
2.3 Aufgabe 3 . . . . .	14
1 Doubly Linked List . . . . .	14
2 Array . . . . .	15
3 . . . . .	17
4 . . . . .	17

# Preface

This is a Quarto book.

To learn more about Quarto books visit <https://quarto.org/docs/books>.

# 1 Blatt 1

## 1.1 Aufgabe 2

a)

$$\log(n!) = \log\left(\prod_{i=1}^n i\right) \quad (\text{Def } n!)$$

$$= \sum_{i=1}^n \log(i) \quad (\text{Eig } \log(\bullet))$$

$$\leq \sum_{i=1}^n \log(n) \quad (\text{Eig } \log(\bullet))$$

$$= n \log(n)$$

Wähle nun  $c_0 := 1$  und  $n_0 := 1$ . Es folgt somit:

$$\begin{aligned} \log(n!) &\leq 1 \cdot n \log(n), \quad \forall n \geq 1 \\ \Leftrightarrow \log(n!) &\in \mathcal{O}(n \log(n)) \end{aligned} \quad (\text{Def } \mathcal{O})$$

b)

Zuerst bemerken wir die folgende Eigenschaft

$$\begin{aligned} n \log(n) &\leq c \log(n!) \\ \Leftrightarrow c &\geq \frac{n \log(n)}{\log(n!)} \\ &= \frac{n \log(n)}{\sum_{i=1}^n \log(i)} \end{aligned}$$

Wir definieren die Folge:

$$\begin{aligned}
c(n) &:= \frac{n \log(n)}{\sum_{i=1}^n \log(i)} \\
&= \frac{\overbrace{\log(n) + \dots + \log(n)}^{n\text{-mal}}}{\log(1) + \dots + \log(n)}
\end{aligned}$$

Wir behaupten ohne Beweis, dass  $c(n)$  eine monoton fallende Folge ist. D.h. es gilt:

$$c(n) \leq c(m), \quad \forall n \geq m$$

Setze nun  $n_0 := 10, c_0 := c(10) = \frac{10 \log(10)}{\sum_{i=1}^{10} \log(i)}$ . Somit folgt:

$$\begin{aligned}
n \log(n) &\leq \left( \frac{n \log(n)}{\sum_{i=1}^n \log(i)} \right) \log(n!) \\
&= c(n) \log(n!) \\
&\leq c_0 \log(n!), \quad \forall n \geq n_0 = 10
\end{aligned}
\tag{c(n) monoton fallend}$$

## 1.2 Aufgabe 3

a)

Da  $f_1 \in \mathcal{O}(g_1)$  und  $f_2 \in \mathcal{O}(g_2)$  existieren  $n_1, n_2, c_1, c_2$  s.d:

$$\begin{aligned}
f_1(n) &\leq c_1 g_1(n), \quad \forall n \geq n_1 \\
f_2(n) &\leq c_2 g_2(n), \quad \forall n \geq n_2
\end{aligned}$$

Setze  $c_0 := \max\{c_1, c_2\}, n_0 := \max\{n_1, n_2\}$ . Dann gilt

$$\begin{aligned}
(f_1 + f_2)(n) &= f_1(n) + f_2(n) \\
&\leq c_1 g_1(n) + c_2 g_2(n), \quad \forall n \geq n_0 & (n \geq n_0 \Rightarrow n \geq n_1 \wedge n \geq n_2) \\
&\leq c_0 g_1(n) + c_0 g_2(n), \quad \forall n \geq n_0 \\
&= c_0 (g_1 + g_2)(n) \quad \forall n \geq n_0 \\
\iff f_1 + f_2 &\in \mathcal{O}(g_1 + g_2) & (\text{Def } \mathcal{O})
\end{aligned}$$

**b)**

mit  $f_1 \in \Theta(g_1)$ ,  $f_2 \in \Theta(g_2)$  existieren  $a_1, a_2, b_1, b_2, n_1, n_2$ , s.d.:

$$\begin{aligned}a_1 f_1(n) &\leq g_1(n) \leq a_2 f_1(n), \forall n \geq n_1 \\b_1 f_2(n) &\leq g_2(n) \leq b_2 f_2(n), \forall n \geq n_2\end{aligned}$$

Setze  $n_0 := \max\{n_1, n_2\}$ ,  $c_1 := a_1 b_1$ ,  $c_2 := a_2 b_2$ . Dann gilt:

$$c_1(f_1 f_2)(n) = a_1 f_1(n) b_1 f_2(n) \leq (g_1 g_2)(n) \leq a_2 f_1(n) b_2 f_2(n) = c_2(f_1 f_2)(n), \quad \forall n \geq n_0$$

Somit  $f_1 f_2 \in \Theta(g_1 g_2)$ .

**c) Falsch:**

Betrachte  $f(n) := n$  und  $g(n) := 10n$ . Offensichtlich gilt  $f \in \Omega(g)$  mit  $c_0 := 1/10, n_0 := 1$ . Aber  $2^n \notin \Omega(2^{10n})$ , da  $2^n$  langsamer als  $2^{10n}$  wächst. (Setze z.B.  $2^n := x$ . Dann  $2^{10n} = (2^n)^{10} = x^{10}$ , und  $x^{10}$  ist offensichtlich schneller als  $x$ )

**d) Falsch:**

Sei  $g(n) := 2^n$ . Dann  $f(n) = g(2n) = 2^{2n} = (2^n)^2$ .  $(2^n)^2$  ist offensichtlich schneller als  $2^n$

**e) Falsch:**

Seien  $f(n) := n^2$ ,  $f_1(n) := n^3$ ,  $f_2(n) := n$ . Es gilt:

$$\begin{array}{ll}f \in \mathcal{O}(f_1) & (n^2 \in \mathcal{O}(n^3)) \\f_1 \in \Omega(f_2) & (n^3 \in \Omega(n))\end{array}$$

aber

$$f \notin \mathcal{O}(f_2) \quad (n^2 \notin \mathcal{O}(n))$$

**f)**

Es gilt:

$$\begin{aligned}\lim_{n \rightarrow \infty} \frac{f(n)}{f_2(n)} &= \lim_{n \rightarrow \infty} \left( \frac{f(n)}{f_1(n)} \cdot \frac{f_1(n)}{f_2(n)} \right) \\ &= \lim_{n \rightarrow \infty} \left( \frac{f(n)}{f_1(n)} \right) \cdot \lim_{n \rightarrow \infty} \left( \frac{f_1(n)}{f_2(n)} \right) \\ &= 0 \cdot c, \text{ fuer ein } c \quad (f \in o(f_1), f_1 \in \mathcal{O}(f_2)) \\ &= 0 \\ \Leftrightarrow f &\in o(f_2) \quad (\text{Def } o)\end{aligned}$$

Wobei wir die alternativen Definitionen von  $o(\bullet)$  und  $\mathcal{O}(\bullet)$  benutzt haben.

## 1.3 Aufgabe 4

**a)**

$\mathcal{O}(n^2 \log(n))$ :

```
read(n) //input
for i := 1 to n :
  for j := 1 to n:
    k := 1
    // O(log(n))
    while (k < n) :
      k := 2 * k
```

**b)**

$\mathcal{O}((\log(n))^2)$ :

```
read(n) //input
i := 1
while (i < n) :
  j := 1
  while (j < n) :
    j := 2 * j
  i := 2 * i
```

c)

Wir 'simulieren' Exponentiation durch einzelne Additionsoperationen. Somit ist  $n^n$  in  $n^n$  Additionen berechnet - Python Implementierung:

```
def add(n, m) :  
    if m == 0 : return n  
    return 1 + add(n, m - 1)  
  
def mult(n, m) :  
    if m == 0 : return 0  
    return add(n, mult(n, m - 1))  
  
def exp(n, m) :  
    if m == 0 : return 1  
    return mult(n, exp(n, m - 1))  
  
def f(n) : return exp(n, n)
```

Wir testen diese Funktion fuer einige Werte:

```
for i in range(5) :  
    print(f(i))
```

```
1  
1  
4  
27  
256
```

Alternativ betrachte folgende rekursive Funktionsdefinition:

```
function recursiveLoops(n : Nat, m : Nat) :  
    if m > 0 then :  
        for i = 1 ... n do :  
            recursiveLoops(n, m - 1)
```

Dann erzeugt der Aufruf `recursiveLoops(n, n)` eine Anzahl von  $\mathcal{O}(n^n)$  rekursiven Aufrufe.



**d)**

$\Theta 2^n$  - Wir 'simulieren' binaeres Zaehlen:

```
read(n)
base := 0
count := 0
k := 1
// invariant: k == 2^b, count < k
while (base < n) :
    k := 2 * k
    base := base + 1
    while (count < k) :
        count := count + 1
    // post-condition: count == k
//post-condition b == n => count == 2^n
```

Python Implementierung:

```
def binary_count(n) :
    base = count = 0
    k = 1
    while (base < n) :
        k = 2 * k
        base = base + 1
        while (count < k) :
            count = count + 1
    return count
```

Wir testen diese Funktion fuer einige Werte. Das Ergebniss ist die Anzahl der Schritte fuer die jeweilige Eingabe:

```
for i in range(11) :
    print(binary_count(i))
```

0  
2  
4  
8  
16  
32

64  
128  
256  
512  
1024

Alternativ:

```
function f(n) :  
    if n == 0 : return 1  
    return f(n - 1) + f(n - 1)
```

Diese rekursive Funktion ruft sich selbst zweimal fuer jeden Wert von  $n$  auf, was zu einer Laufzeit von  $2^n$  fuehrt.

## 2 Blatt 2

### 2.1 Aufgabe 1

1

Rekursionsbaum des Aufrufs `sum(<1, 2, 3, 4, 5, 6, 7, 8>)`

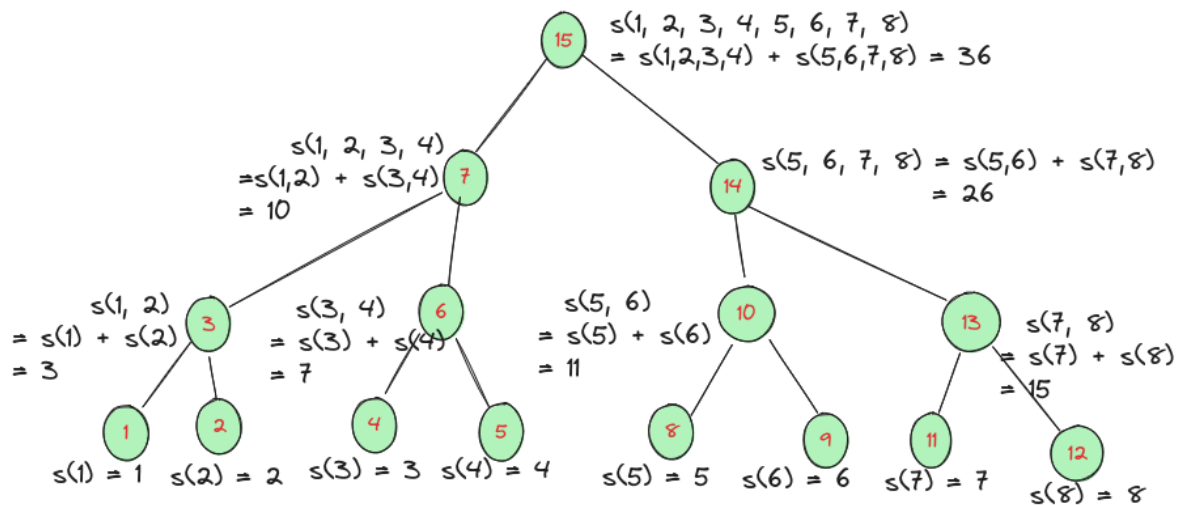


Figure 2.1: Rekursionsbaum

Die Nummerierung der Knoten entspricht der Berechnungsreihenfolge.

2

Eine nicht-konstante Laufzeit entsteht, falls uebergebene arrays auf den Stack des Funktionsaufrufs kopiert werden muessen.

Wenn eine gegebene Implementierung der Programmiersprache folgende zwei Eigenschaften aufweist, kann dies vermieden werden:

- Die Groesse eines Arrays ist immer als zusaetzliche Information beinhaltet.

- Die Funktionsaufrufe werden per-default als **call by reference** realisiert statt **call by value**.

So wuerde fuer einen existierenden Array  $A : \text{Array}[0..n-1]$  of  $\mathbb{N}$  der allgemeiner Ausdruck  $A[l..k]$  einen Array liefern, dessen Anfang-position im Speicher und Groesse durch Pointerarithmetik, bzw durch den Ausdruck  $k-l+1$  bestimmt werden koennen. Das sind nur zwei Grundoperationen, und somit  $\mathcal{O}(1)$

Da die Uebergabe der Arrays per Referenz stattfindet, wuerden die Aufrufe  $\text{sum}(A[0..m-1])$  und  $\text{sum}(A[m..n-1])$  nur konstante Zeit bei der Initialisierungen auf ihren Function call-stacks benoetigen.

### 3

- a) Die Laufzeit erfuehlt die Rekurrenzgleichung:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 1 + 2 \cdot T\left(\frac{n}{2}\right) \quad (\text{fuer } n = 2^k > 1) \end{aligned}$$

- b) Da, die Eingabe bei jedem Aufruf halbiert wird ist die Tiefe des Rekurrenzbaums (Figure 2.1)  $k = \log_2(n)$ . Dieser Baum ist vollstaendig binaer, deshalb enthaelt jede Tiefe  $i$  genau  $2^i$  Knoten, fuer  $i = 0 \dots k$ . Somit betraegt die Gesamtzahl der Knoten:

$$\begin{aligned} N &= \sum_{i=0}^{\log_2(n)} 2^i \\ &= 2^{\log_2(n)+1} - 1 \\ &= 2n - 1 \end{aligned} \quad (\text{Geom Reihe})$$

Bei jedem Knoten wird eine konstante Anzahl von Additions- & Zuweisungsoperationen durchgefuehrt, und das Ergebnis zur aufrufenden Funktion zurueckgegeben. Somit ist die Laufzeit proportional zur Anzahl der Knoten, die wir in der vorangehenden Diskussion berechnet haben, d.h.  $T(n) = c_1 n + c_2$ . Dann gilt offensichtlich  $T(n) = \Theta(n)$

## 2.2 Aufgabe 2

a)

$$\begin{aligned}a &= 1, \\c &= \tilde{c}, \\d &= 1 < 2 = b \\ \Rightarrow T(n) &\in \Theta(n) \qquad \qquad \qquad (\text{Fall } d < b \text{ des MT})\end{aligned}$$

b)

$$\begin{aligned}a &= 1, \\c &= 4, \\d &= 9 > 3 = b \\ \Rightarrow T(n) &\in \Theta(n^{\log_3(9)}) = \Theta(n^2) \qquad \qquad \qquad (\text{Fall } d > b \text{ des MT})\end{aligned}$$

c)

Der Ausdruck  $C(n/4) + n + 6$  kann asymptotisch als  $C(n/4) + n$  vereinfacht werden, da Addition mit konstante vernachlässigt werden kann. Somit:

$$\begin{aligned}a &= 1, \\c &= 1, \\d &= 1 < 4 = b \\ \Rightarrow T(n) &\in \Theta(n) \qquad \qquad \qquad (\text{Fall } d < b \text{ des MT})\end{aligned}$$

d)

In c) wurde gezeigt, dass  $C(n) \in \Theta(n)$ . Somit kann  $C(n)$  für asymptotische Zwecke durch  $c \cdot n$  ersetzt werden. Dann gilt:

$$T(n) = c \cdot n + 4D\left(\frac{n}{4}\right)$$

und somit:

$$\begin{aligned}
 a &= 1, \\
 d &= 4 = 4 = b \\
 \Rightarrow T(n) &\in \Theta(n \log n)
 \end{aligned}
 \quad (\text{Fall } d = b \text{ des MT})$$

## 2.3 Aufgabe 3

### 1 Doubly Linked List

Wir gehen von einer Implementierung aus, die das **Dummy-element** verwendet, wie in der VL beschrieben.

**Idee:** Tausche fuer jedes List-Item die Pointer **next** und **prev** aus. Illustration:

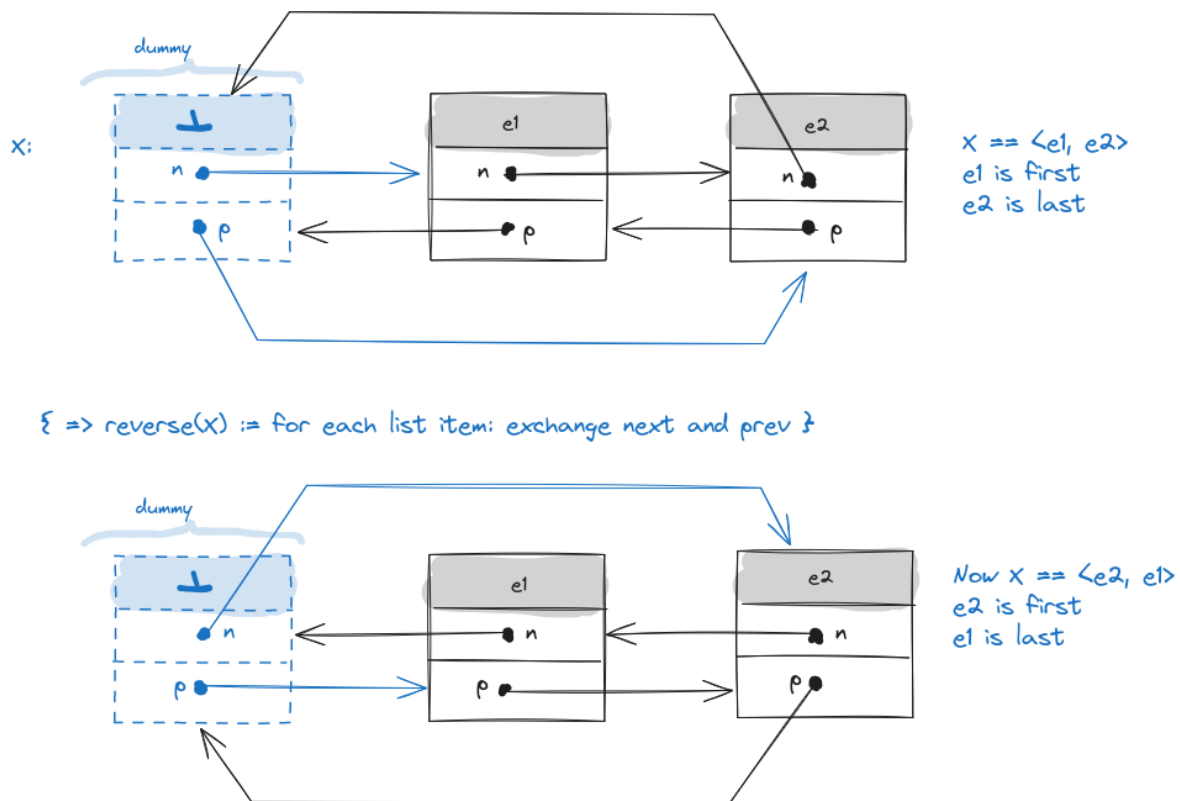


Figure 2.2: Reverse(X)

a) Pseudocode implementierung:

```

procedure reverse(X : List<T>)
    // let e1 := initial first element
    // let e_n := initial last element

    // define ip initially to point to e1
    // exchange last and first, e1 is now last
    ip := X.first() : *Item<T>
    X.first() := X.last()
    X.last() := ip

    // invariant: reversed from e1 to *ip
    while (ip->next != &dummy)
        //exchange next and prev of the item pointed by ip
        ip_next := ip->next : *Item<T>
        ip->next := ip->prev
        ip->prev := ip_next
        ip = ip_next //increment to next item
    //*ip == e_n

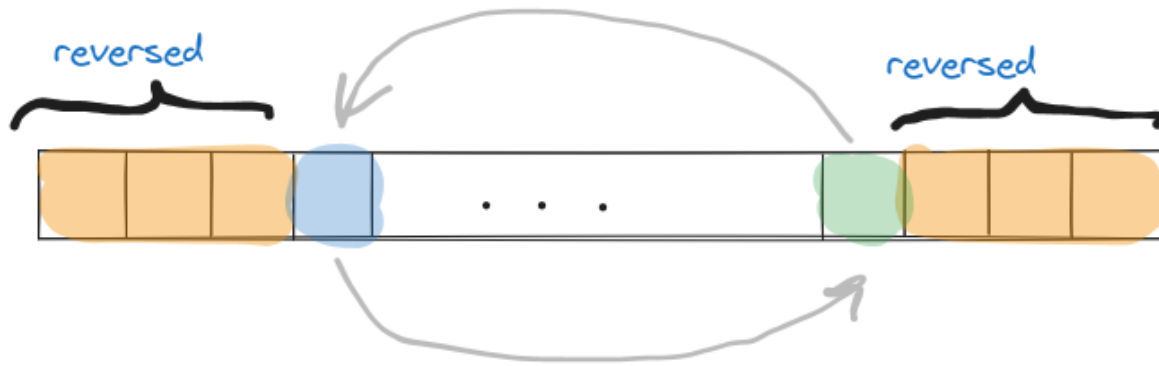
    // take care of e_n's pointers:
    ip.next = ip.prev
    ip.prev = &dummy

```

- b) Siehe Kommentare fuer den Beweis der Korrektheit
- c) Der Algorithmus benoetigt keine zusaetzliche Worte, da es keine neue Listenelemente abgelegt oder existierende Elemente kopiert werden. Es werden einfach nur Pointer ausgetauscht.
- d) Die Listenelemente werden sequentiell durchgelaufen und fuer jedes Element werden eine konstante Anzahl von Grundoperationen durchgefuehrt  $\Rightarrow \mathcal{O}(n)$ .

## 2 Array

**Idee:** Tausche die ‘auesersten’ noch nicht ausgetauschten Elementen aus, und inkrementiere zu den inneren. Siehe das Bild:



=> {swap blue and green}

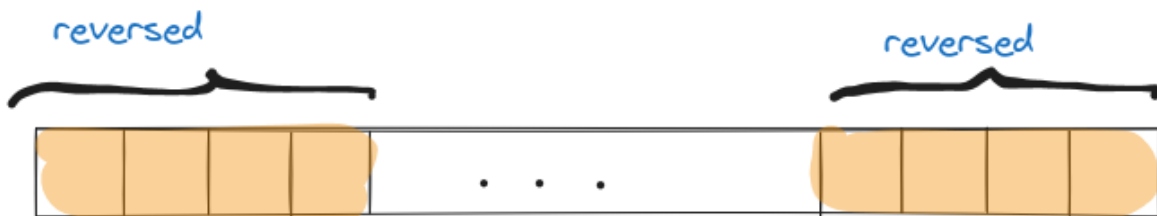


Figure 2.3: Reverse Array

a) Pseudocode:

```

procedure reverse(X: Array[0..n-1] of Nat)
  i := 0 : Nat
  // invariant: the X[0..i] and X[(n-1) - i .. n-1]
  // portions of X are reversed
  while (i < n/2)
    temp := X[i] : Nat
    X[i] := X[(n-1) - i]
    X[(n-1) - i] := temp
    i := i + 1
  // i == ceiling(n/2)

```

Python Beispiel:



```

X = [1, 2, 3, 4, 5]
i = 0
n = len(X)
while (i < n/2) :
    temp = X[i]
    X[i] = X[(n-1) - i]
    X[(n-1) - i] = temp
    i = i + 1
print(X)

```

[5, 4, 3, 2, 1]

- b) Siehe die Kommentare im Pseudocode fuer den Beweis der Korrektheit
- c) Der Algorithmus verwendet keine neue Worte, da die Einträge des Arrays “in-place” ausgetauscht werden. D.h. der vorhandene Array wird ueberschrieben
- d) Der Algorithmus besteht aus einer **while**-schleife mit  $n/2$  iterationen  $\Rightarrow \Theta(n)$ .

**3**

**4**