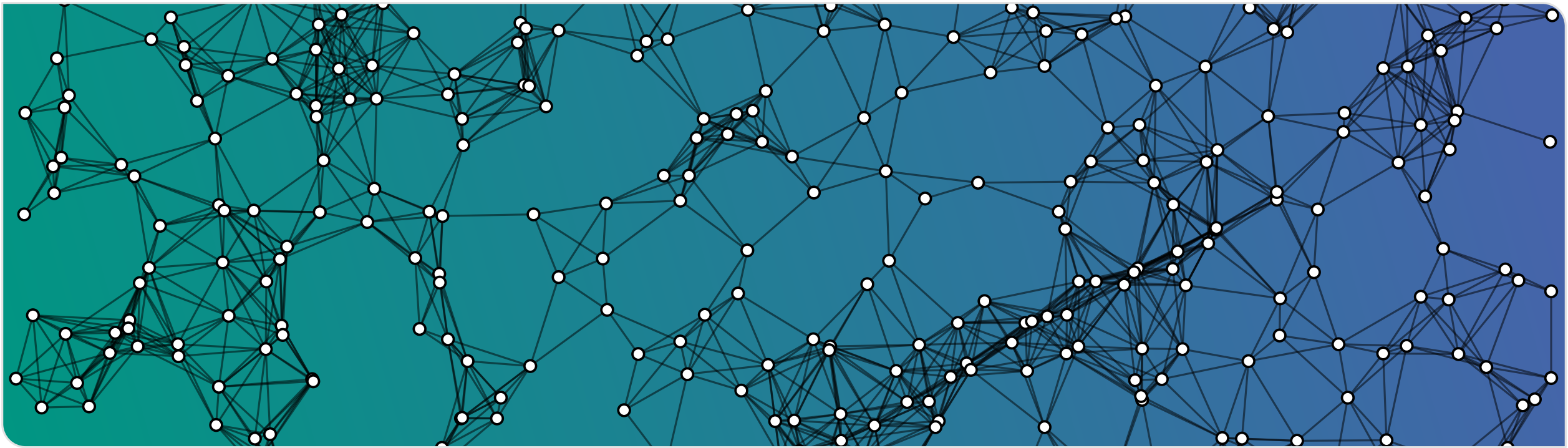
Karlsruher Institut für Technologie

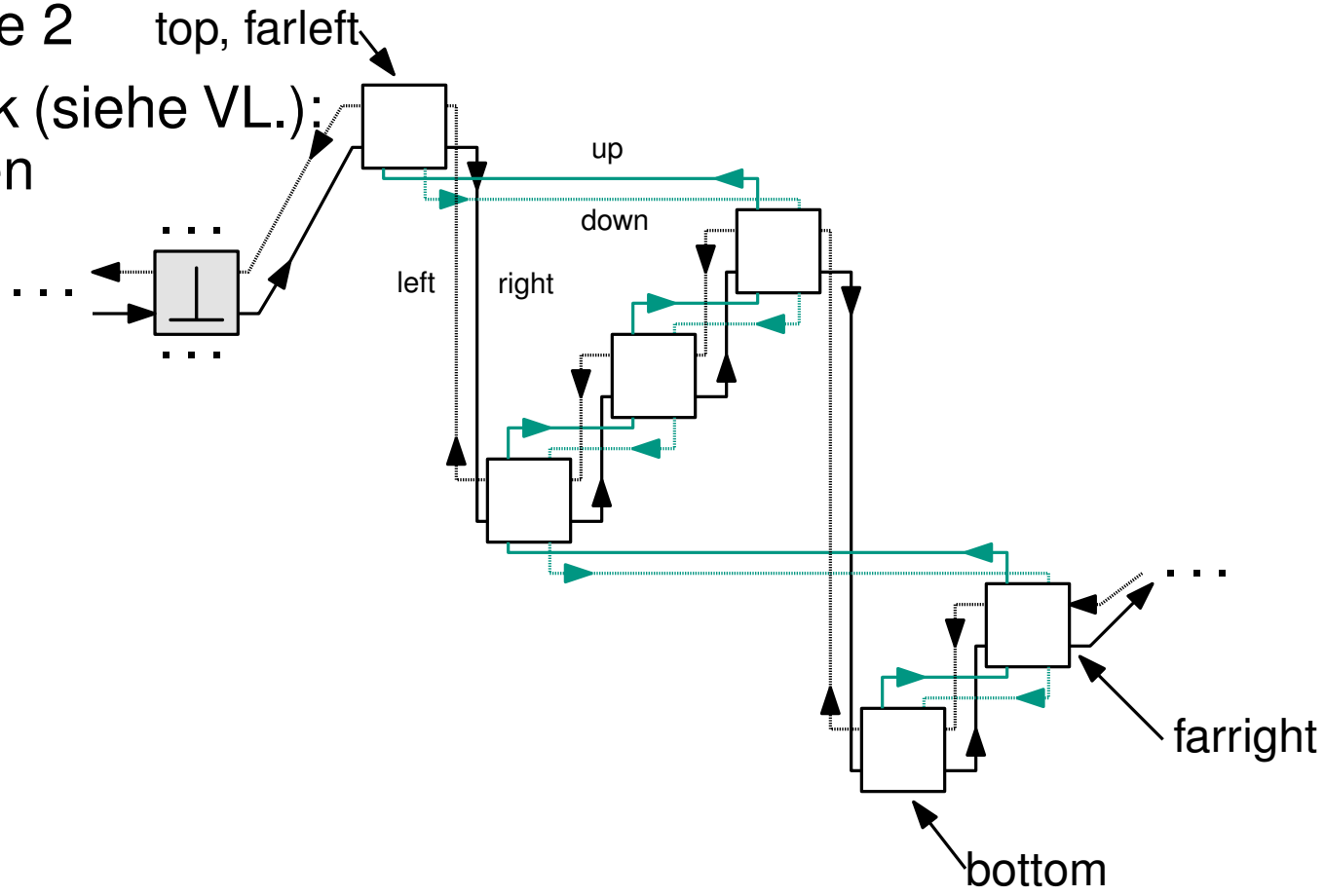
Algorithmen 1

Übung 2
Algorithmenentwurf, amortisierte Analyse



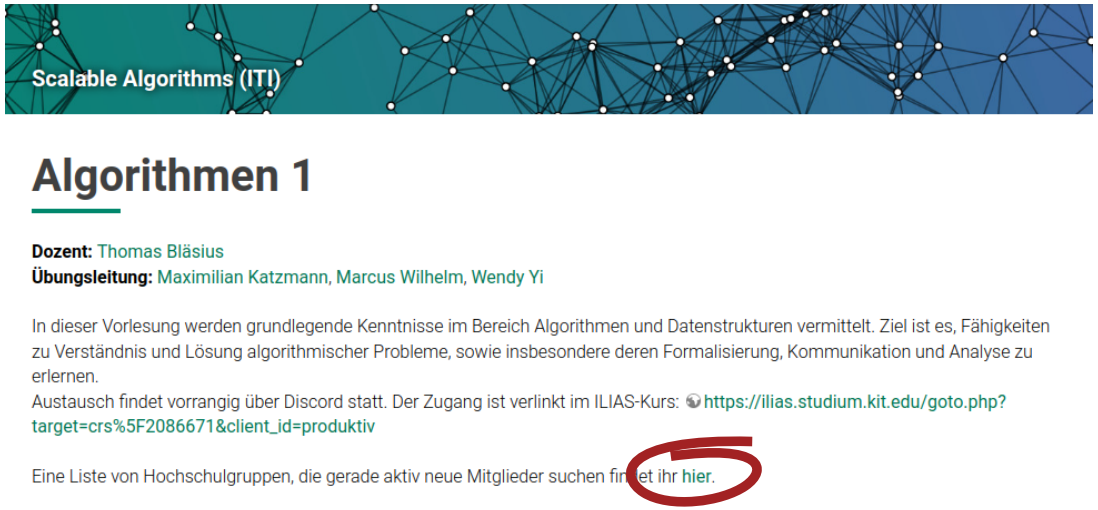
Hinweise / Organisatorisches

- Blatt 3, Aufgabe 2
  - möglicher Trick (siehe VL.): Dummy Knoten



Hinweise / Organisatorisches

- Blatt 3, Aufgabe 2
- Pseudocode vs. textuelle Beschreibung
  - „Gib Algorithmen nur in Pseudocode an anstatt sie in Worten zu beschreiben, falls dies explizit gefordert ist!“
  - ansonsten: 0 Punkte
- Hochschulgruppen

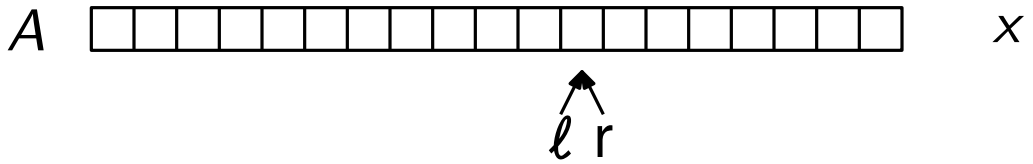


Pseudocode

Was macht der Pseudocode?

func$(A : [\mathbb{N}, n], x : \mathbb{N})$ // A aufst. sortiert

```
1  ℓ := 0
2  r := n − 1
3  while ℓ < r do
4      m := (ℓ + r)/2
5      if x < A[m] then r := m
6      if x > A[m] then ℓ := m
7      if x = A[m] then return „found x“
8  return „not found“
```



In welchen Zeilen befinden sich Fehler?

Klicker UZH

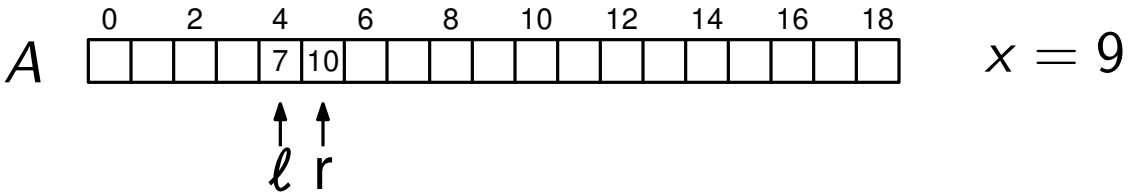
app.klicker.uzh.ch/join/algoueb

Pseudocode

Was macht der Pseudocode?

func$(A : [\mathbb{N}, n], x : \mathbb{N})$ // A aufst. sortiert

```
1  ℓ := 0
2  r := n − 1
3  while ℓ < r do
4      m := ⌊(ℓ + r)/2⌋
5      if x < A[m] then r := m
6      if x > A[m] then ℓ := m
7      if x = A[m] then return „found x“
8  return „not found“
```



In welchen Zeilen befinden sich Fehler?

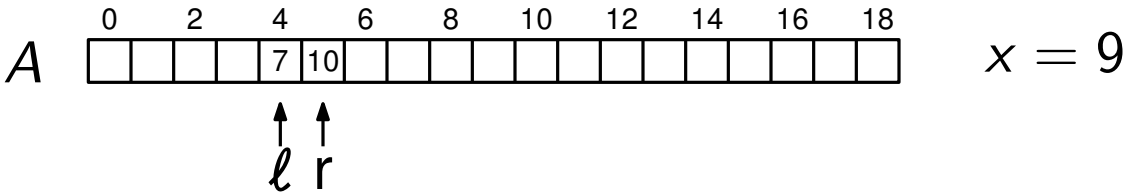
- betrachte $\ell = 4, r = 5$:
  - $m := (\ell + r)/2 = 4.5$
  - runden nötig, $m := 4$
- angenommen $A[4] = 7, A[5] = 10$ und $x = 9$
  - $9 = x > A[m] = A[4] = 7, x \neq A[m]$
  - Endlosschleife

Pseudocode

Was macht der Pseudocode?

func$(A : [\mathbb{N}, n], x : \mathbb{N})$ // A aufst. sortiert

```
1  ℓ := 0
2  r := n − 1
3  while ℓ < r do
4      m := ⌊(ℓ + r)/2⌋
5      if x < A[m] then r := m − 1
6      if x > A[m] then ℓ := m + 1
7      if x = A[m] then return „found x“
8  return „not found“
```



In welchen Zeilen befinden sich Fehler?

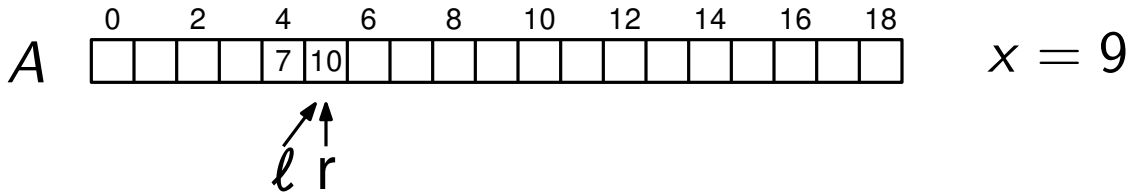
- betrachte $\ell = 4, r = 5$:
  - $m := (\ell + r)/2 = 4.5$
  - runden nötig, $m := 4$
- angenommen $A[4] = 7, A[5] = 10$ und $x = 9$

Pseudocode

Was macht der Pseudocode?

func$(A : [\mathbb{N}, n], x : \mathbb{N})$ // A aufst. sortiert

```
1  ℓ := 0
2  r := n − 1
3  while ℓ < r do
4      m := ⌊(ℓ + r)/2⌋
5      if x < A[m] then r := m − 1
6      if x > A[m] then ℓ := m + 1
7      if x = A[m] then return „found x“
8  return „not found“
```



In welchen Zeilen befinden sich Fehler?

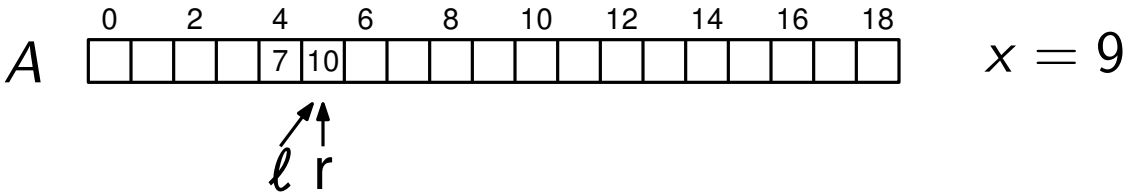
- betrachte $\ell = 4, r = 5$:
  - $m := (\ell + r)/2 = 4.5$
  - runden nötig, $m := 4$
- angenommen $A[4] = 7, A[5] = 10$ und $x = 9$
  - $x > A[m]$, also setze $\ell = 5$
  - Abbruch der Schleife

Pseudocode

Was macht der Pseudocode?

func$(A : [\mathbb{N}, n], x : \mathbb{N})$ // A aufst. sortiert

```
1  ℓ := 0
2  r := n − 1
3  while ℓ < r + 1 do
4      m := ⌊(ℓ + r)/2⌋
5      if x < A[m] then r := m − 1
6      if x > A[m] then ℓ := m + 1
7      if x = A[m] then return „found x“
8  return „not found“
```



In welchen Zeilen befinden sich Fehler?

- betrachte $\ell = 4, r = 5$:
  - $m := (\ell + r)/2 = 4.5$
  - runden nötig, $m := 4$
- angenommen $A[4] = 7, A[5] = 10$ und $x = 9$
  - $x > A[m]$, also setze $\ell = 5$
  - Abbruch der Schleife
    - nicht korrekt, falls $x = 10$

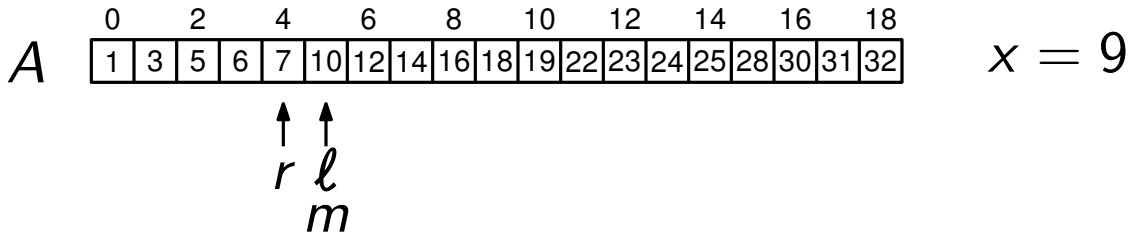
Pseudocode

Stimmt der Pseudocode nun?

```
func(A : [ℕ, n], x : ℕ) // A aufst. sortiert
    ℓ := 0
    r := n − 1
    while ℓ < r + 1 do
        m := ⌊(ℓ + r)/2⌋
        if x < A[m] then r := m − 1
        if x > A[m] then ℓ := m + 1
        if x = A[m] then return „found x“
    return „not found“
```

Frage: Was wenn $x = 10$?

- „found x“ sobald $m = 5$



Behauptung: Pseudocode ist korrekt.

- falls $x \in A$:
  - ℓ und r werden angepasst, bis $x = A[m]$
  - „found x“
- falls $x \notin A$:
  - ℓ und r werden angepasst, bis $\ell = r$
  - danach $r = \ell - 1$ oder $\ell = r + 1$
  - „not found“

Wie kommuniziert man Algorithmen?

Karlsruher Institut für Technologie

Abstrakte Beschreibung

Binäre Suche

- durch einen Vergleich: entscheide ob x in der linken oder rechten Hälfte von A liegt
- suche rekursiv in der relevanten Hälfte von A
- Abbruch: zu durchsuchende Folge ist nur noch konstant groß (hier: 2)

Pseudocode

```
func(A : [ℕ, n], x : ℕ) // A aufst. sortiert
    ℓ := 0
    r := n − 1
    while ℓ < r + 1 do
        m := ⌊(ℓ + r)/2⌋
        if x < A[m] then r := m − 1
        if x > A[m] then ℓ := m + 1
        if x := A[m] then return „found x“
    return „not found“
```

- Intuition einfacher zu erkennen
- Implementierungsdetails fehlen
- gut um Idee zu kommunizieren

- nah an Implementierung
- anfällig für kleine Fehler
- gut um Details für Implementierung zu kommunizieren

Beim Lösen von Problemen: **zuerst** Idee überlegen, **dann** Pseudocode schreiben.

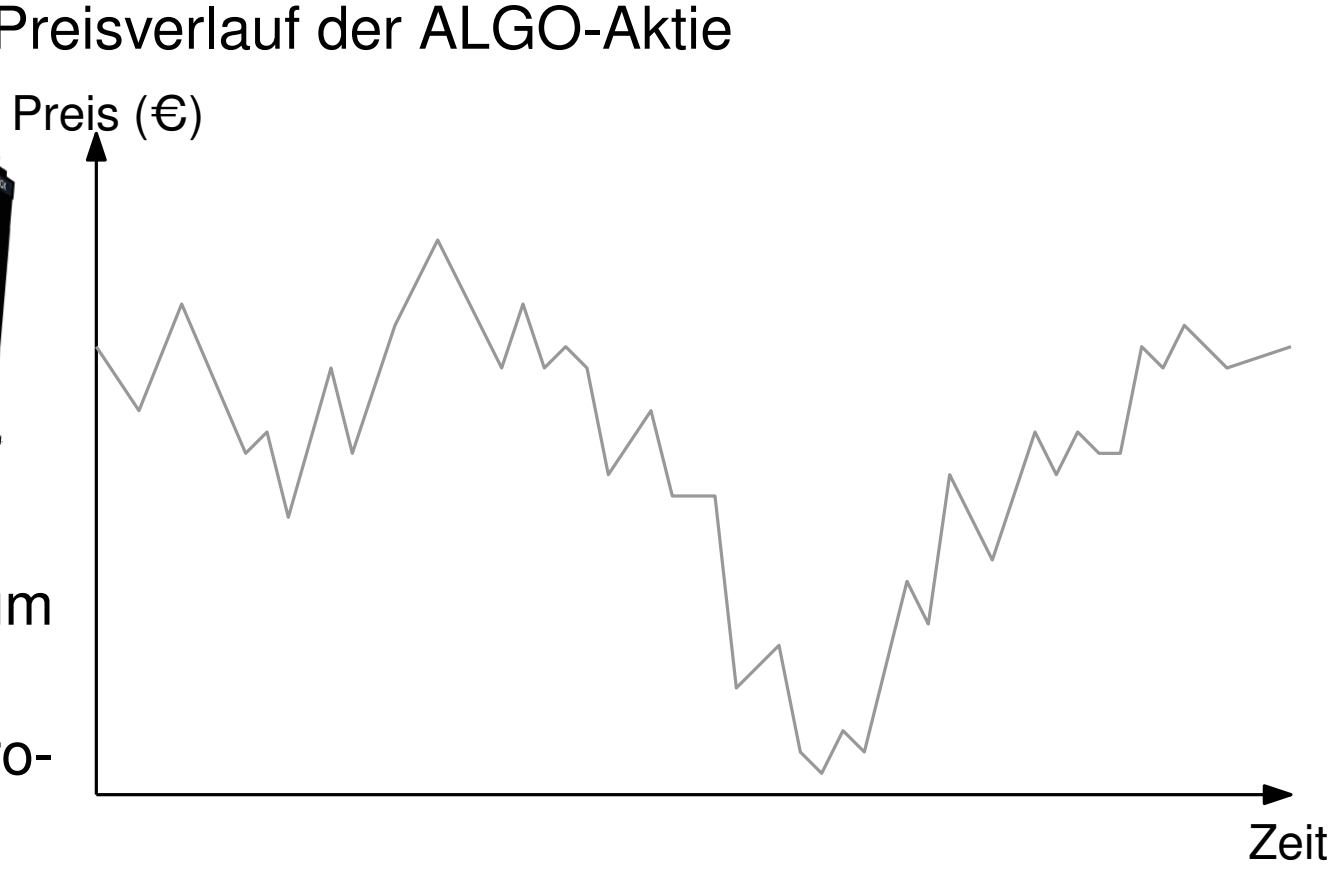
Ziel: viel Geld verdienen

Masterplan für großen Reichtum

- Schritt 1: vielversprechende Aktie suchen.
- Schritt 2: Zeitmaschine suchen. (single use)
- Schritt 3: zum richtigen Zeitpunkt kaufen und verkaufen...

Problem: wie finden wir Zeitpunkte um Gewinn zu maximieren?

Gesucht: Algorithmus für dieses Problem



Gewinnmaximierung

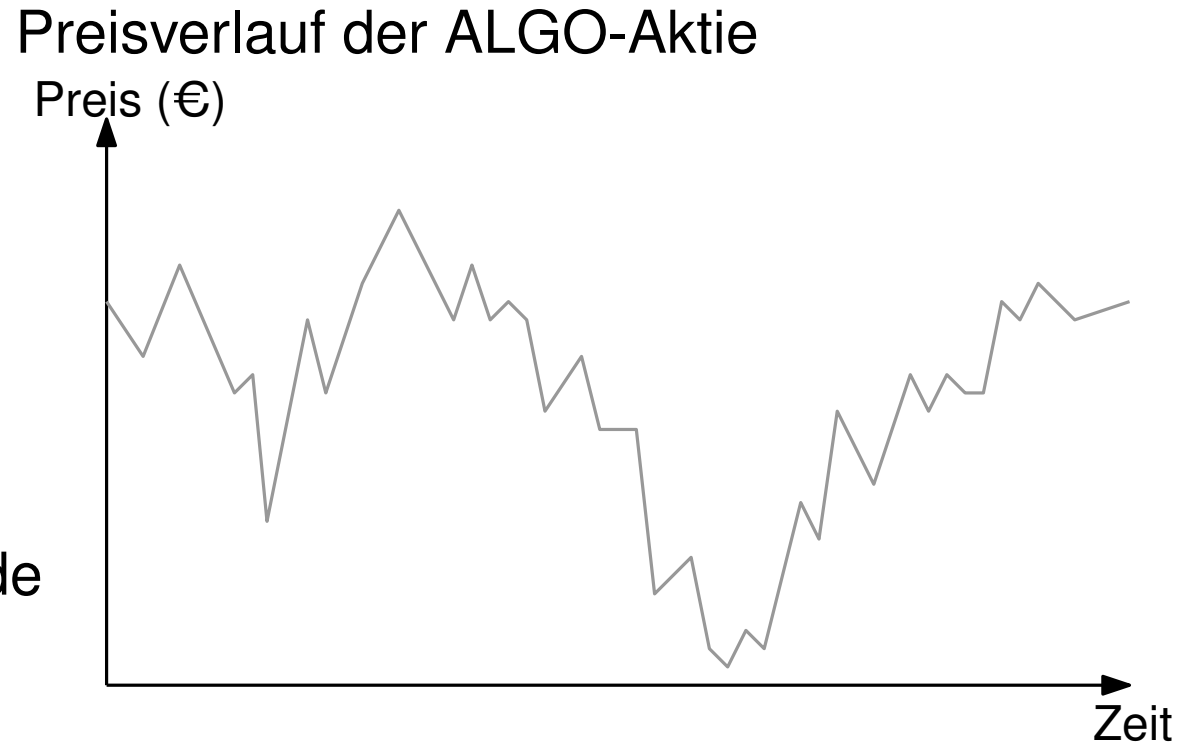
Problem: wie finden wir Zeitpunkte um Gewinn zu maximieren?

Gesucht: Algorithmus für dieses Problem

Fragen:

1. Formalisierung des Problems
   - Was sind Eingabe, Ausgabe?
2. Entwickeln eines Algorithmus
   - Algorithmische Idee finden
   - Beschreibung, evtl. Pseudocode
3. Überprüfen des Algorithmus
   - Analyse von Korrektheit und Laufzeit

ggf. Algo oder Formalisierung anpassen



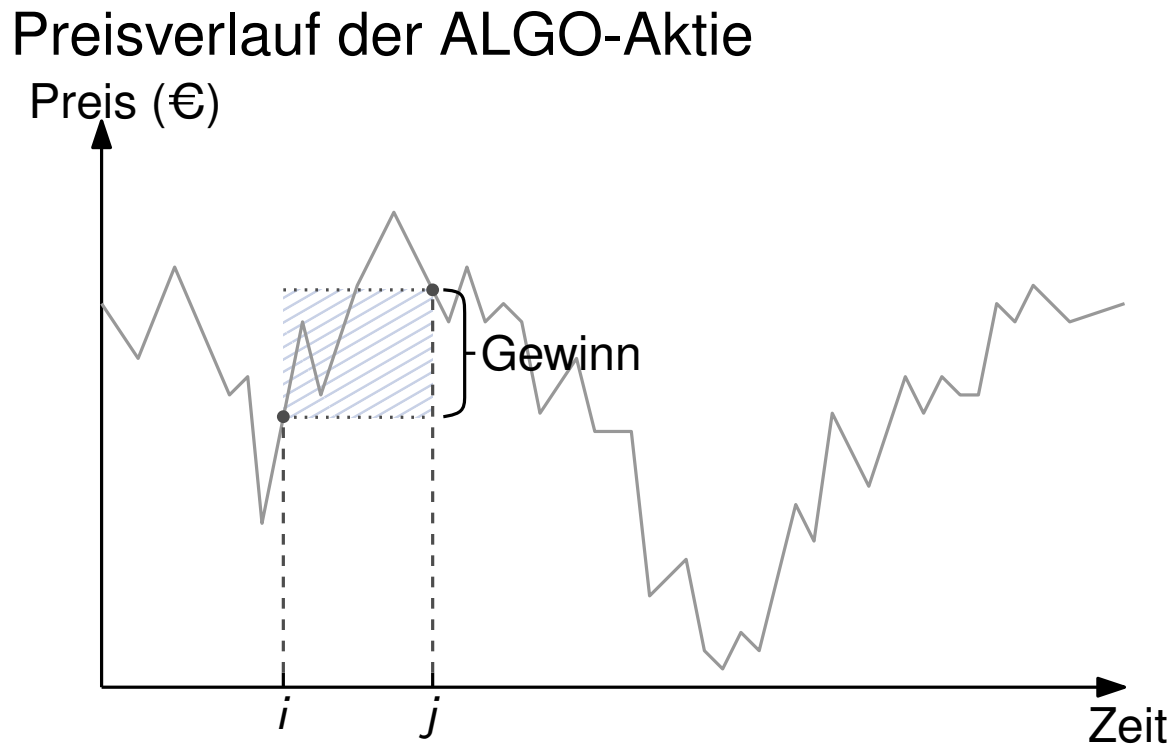
Formalisierung

Problem: wie finden wir Zeitpunkte um Gewinn zu maximieren?

Gesucht: Algorithmus für dieses Problem

Frage: Formalisierung des Problems

- Was sind Eingabe, Ausgabe?
- Mögliche Antwort:
  - **gegeben:** Array A mit Zahlen
  - **gesucht:** Indizes i, j mit $i < j$, sodass $A[j] - A[i]$ maximal ist.



Entwickeln eines Algorithmus

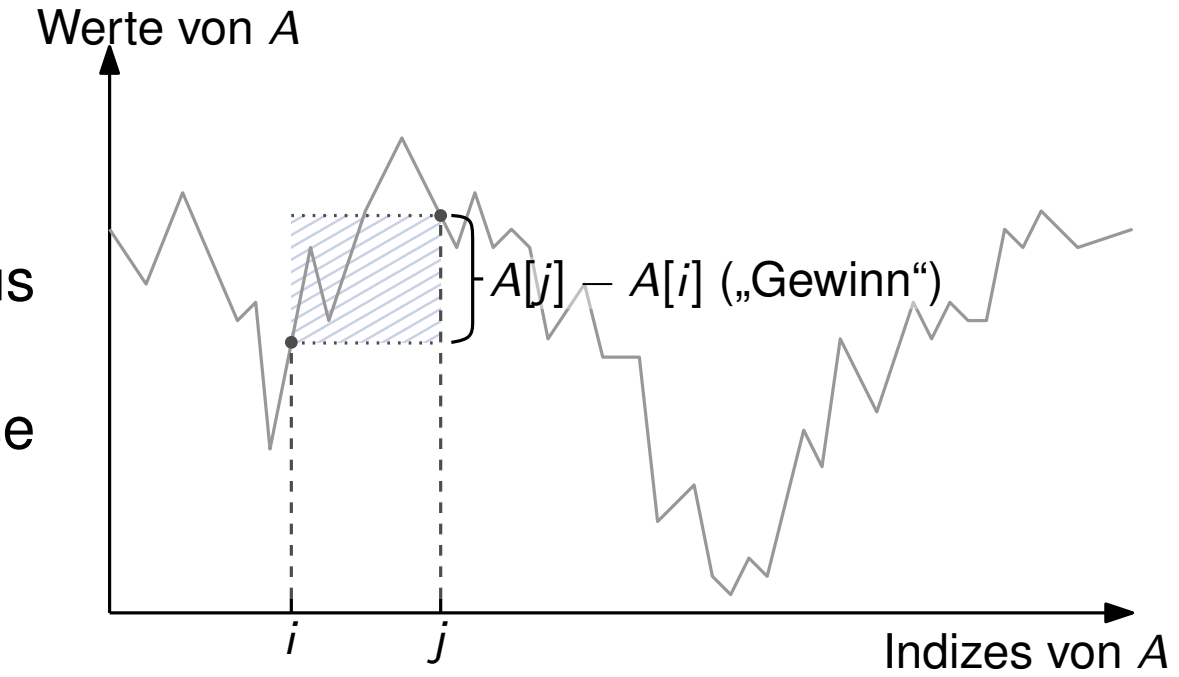
Gegeben: Array A mit Zahlen

Gesucht: Indizes i, j mit $i < j$, sodass $A[j] - A[i]$ maximal ist.

Frage: Wie könnte ein (einfacher) Algorithmus aussehen?

- Idee: vergleiche alle Werte von A paarweise und gib Paar mit größtem Gewinn aus
- Pseudocode:

```
profitmax(A : [ℕ, n])
    best_val := −1, best_i := −1, best_j := −1
    for i ∈ {0, ..., n − 1} do
        for j ∈ {i + 1, ..., n − 1} do
            if A[j] − A[i] > best_val then:
                best_val := A[j] − A[i], best_i := i, best_j := j
    return best_i, best_j
```



Analyse

- Laufzeit: $O(n^2)$
- Korrektheit: alle Paare werden überprüft

Frage: geht es schneller?

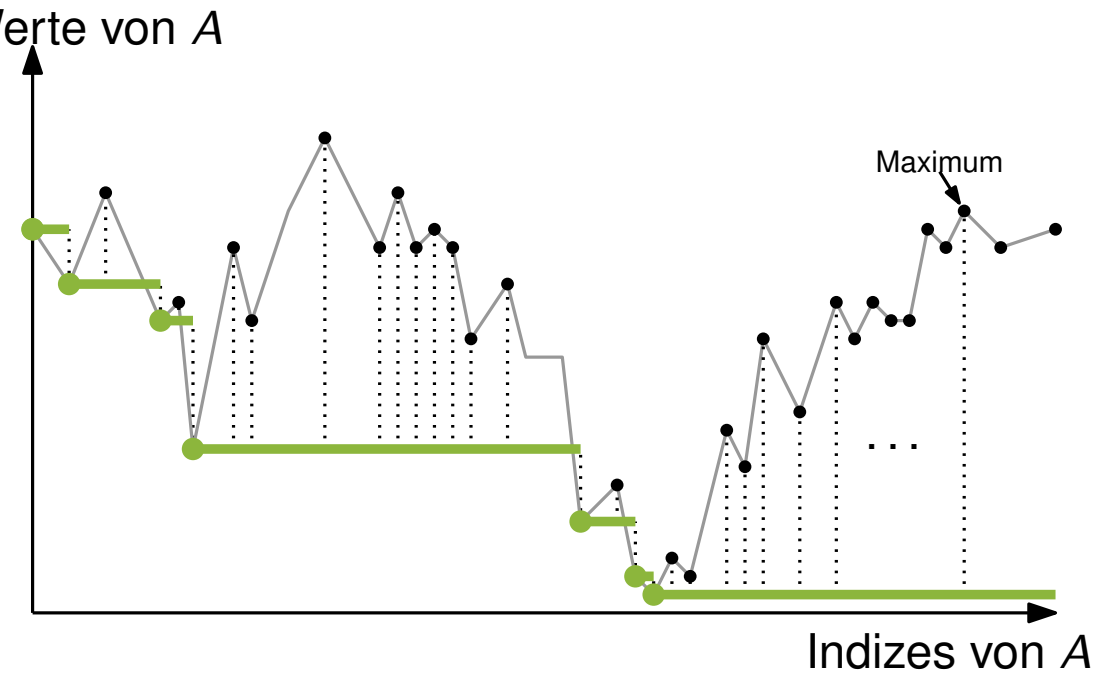
Entwickeln eines schnelleren Algorithmus

Gegeben: Array A mit Zahlen

Gesucht: Indizes i, j mit $i < j$ sodass $A[j] - A[i]$ maximal ist.

Frage: Müssen wirklich alle Paare betrachtet werden?

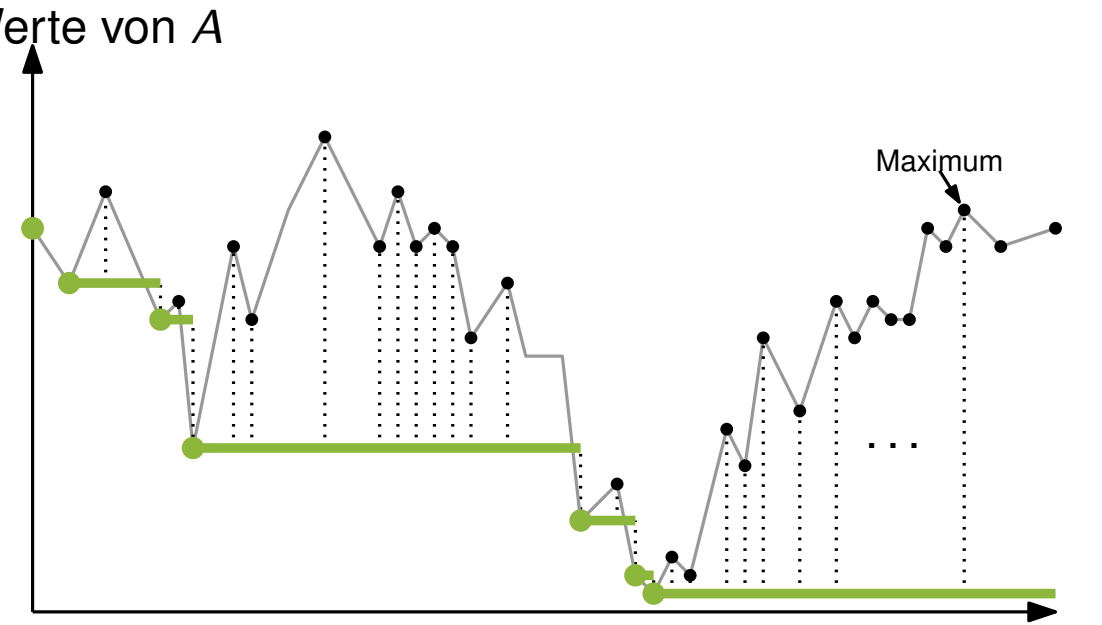
- nur lokale Minima / Maxima betrachten
- Kauf zum Zeitpunkt i nur falls für alle $k < i$: $A[k] > A[i]$
- Idee:
  - berechne für jedes $i \in \{0, \ldots, n-1\}$ den Wert $A'[i] = \min\{A[k] \mid 0 \leq k \leq i\}$
  - finde Maximum durch Prüfen von $A[i] - A'[i]$ für jedes $i \in \{0, n-1\}$



Entwickeln eines schnelleren Algorithmus

- Idee:
  - berechne für jedes $i \in \{0, n-1\}$ den Wert $A'[i] = \min\{A[k] \mid 0 \leq k \leq i\}$
  - finde Maximum durch Prüfen von $A[i] - A'[i]$ für jedes $i \in \{0, n-1\}$
- Pseudocode:

```
profitmax(A : [ℕ, n])
    min_val := ∞, min_i := −1
    max_profit := −1, profit_idx := (−1, −1)
    for i ∈ {0, ..., n − 1} do
        if A[i] < min_val then: min_val = A[i], min_i = i
            min_val := A[i], min_i = i
        if A[i] − min_val > max_profit then:
            max_profit := A[i]−min_val, profit_idx := (min_i, i)
    return profit_idx
```



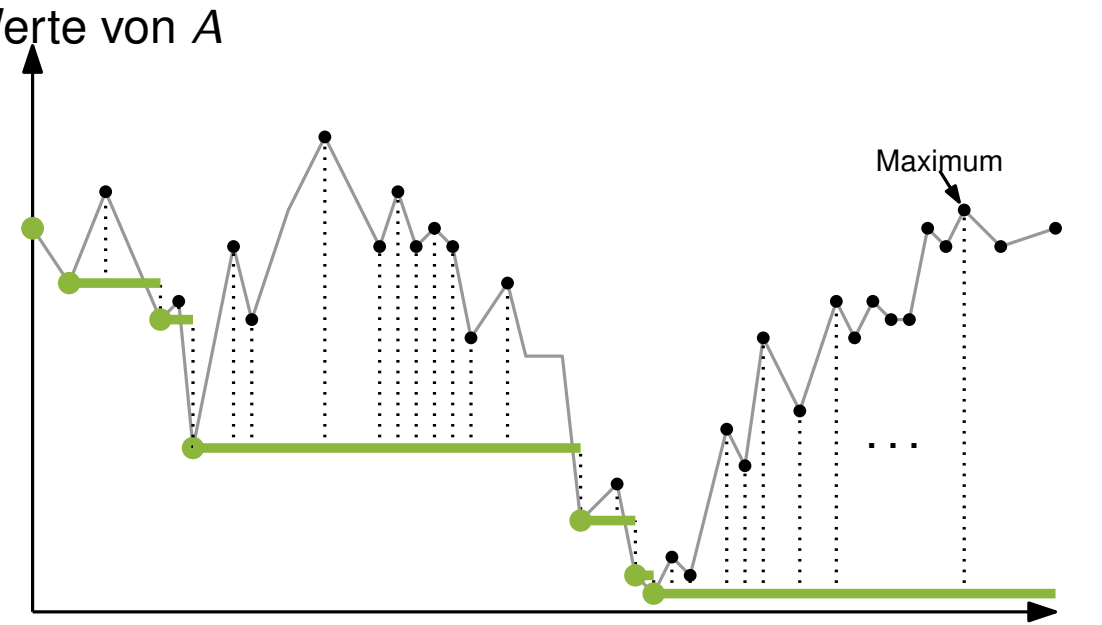
Analyse

- Laufzeit: $O(n)$
- Korrektheit zeigen:
  - Maximum hat Form $A[i] - A'[i]$
  - Algo berechnet A' und $A[i] - A'[i]$

Pause?

- Idee:
  - berechne für jedes $i \in \{0, n-1\}$ den Wert $A'[i] = \min\{A[k] \mid 0 \leq k \leq i\}$
  - finde Maximum durch Prüfen von $A[i] - A'[i]$ für jedes $i \in \{0, n-1\}$
- Pseudocode:

```
profitmax(A : [ℕ, n])
    min_val := ∞, min_i := −1
    max_profit := −1, profit_idx := (−1, −1)
    for i ∈ {0, ..., n − 1} do
        if A[i] < min_val then: min_val = A[i], min_i = i
            min_val := A[i], min_i = i
        if A[i] − min_val > max_profit then:
            max_profit := A[i]−min_val, profit_idx := (min_i, i)
    return profit_idx
```

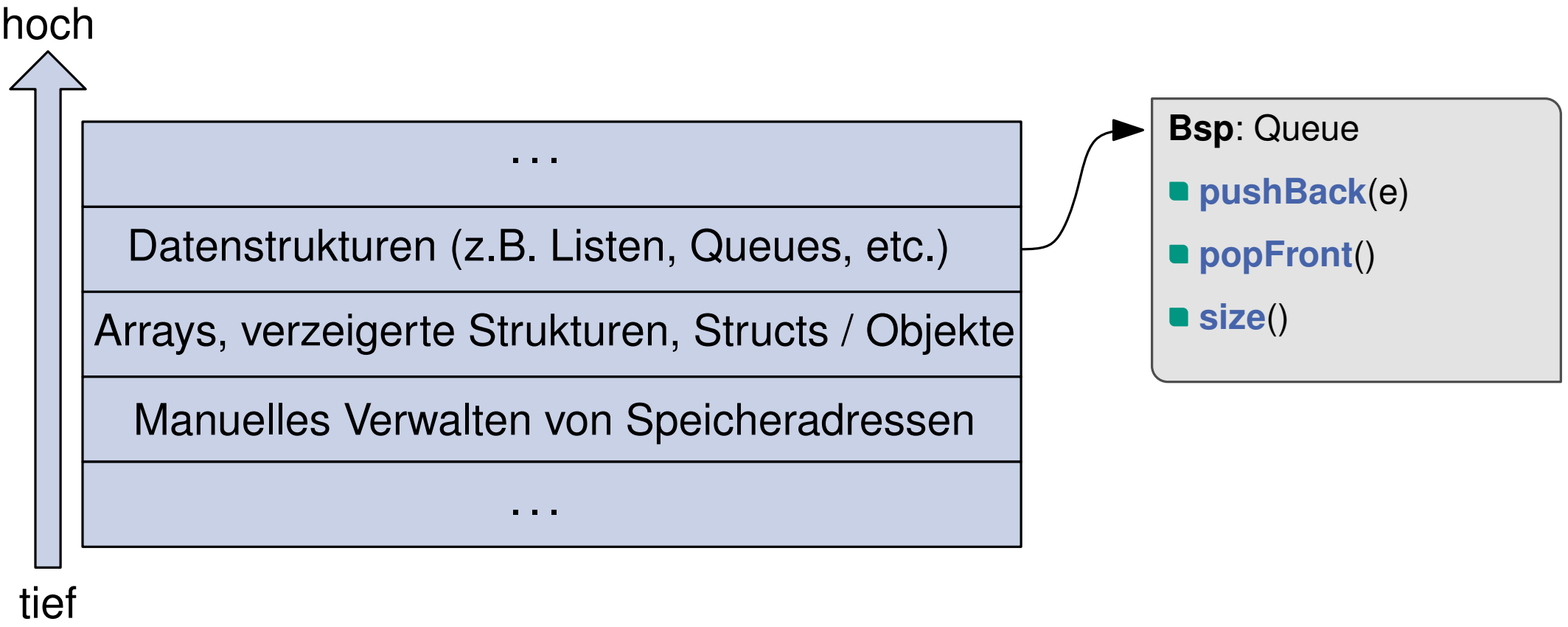


Analyse

- Laufzeit: $O(n)$
- Korrektheit zeigen:
  - Maximum hat Form $A[i] - A'[i]$
  - Algo berechnet A' und $A[i] - A'[i]$

Motivation: Datenstrukturen

Abstraktionsebenen auf dem Speicher



Motivation: Amortisierte Analyse

- Laufzeit von Operationen auf DS nicht immer gleich
  - z.B.: Operation meist günstig, seltener teurer

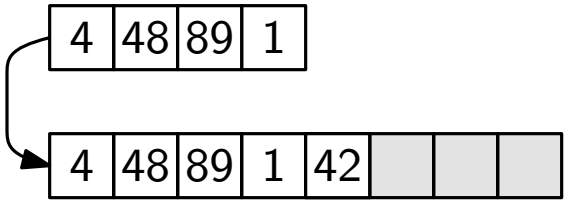
Idee

- **amortisierte Kosten**: durchschnittliche Kosten pro Operation in einer Folge von Operationen

Verschiedene Techniken zur Analyse

- Aggregation
- Charging
- Konto
- Potential

Beispiel: unbeschränktes Array



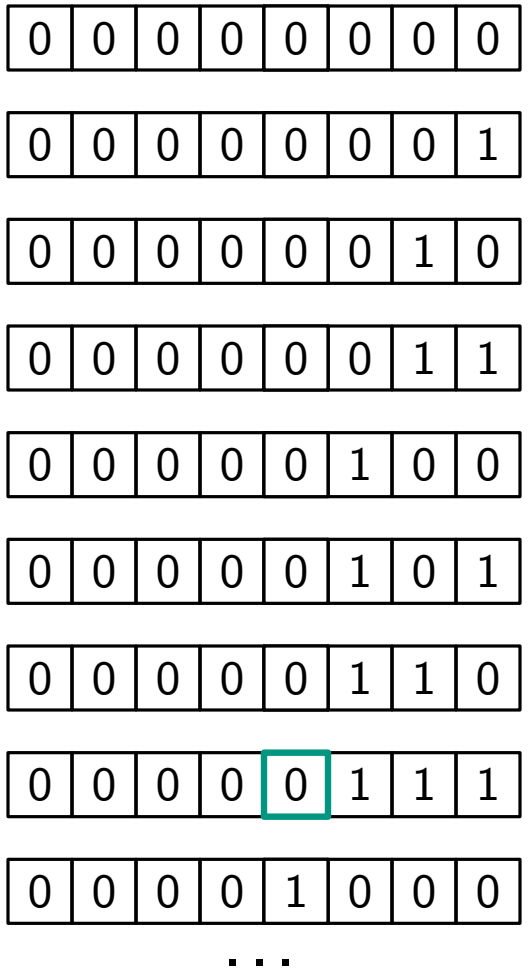
Beispiel: Binärzähler

Idee

- Array A verwaltet Bits
- Funktion **increment()** erhöht Zähler

Algorithmus

- suche Stelle i mit rechtester 0
- setze Stelle i auf 1
- setze Stellen rechts von i auf 0



Beispiel: Binärzähler

Idee

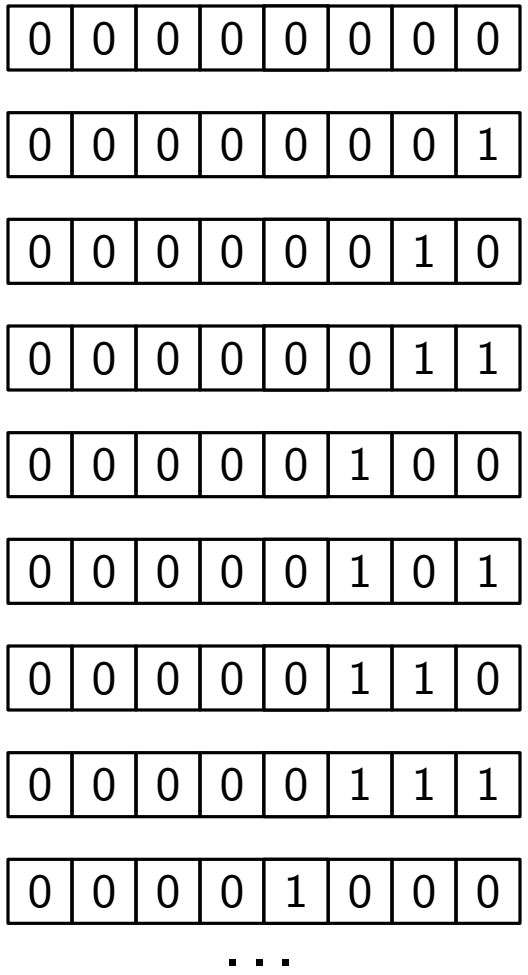
- Array A verwaltet Bits
- Funktion **increment()** erhöht Zähler

Algorithmus in Pseudocode

increment()

```
i := 0
while A[i] = 1 do
    A[i] := 0
    i := i + 1
A[i] := 1
```

Laufzeit?



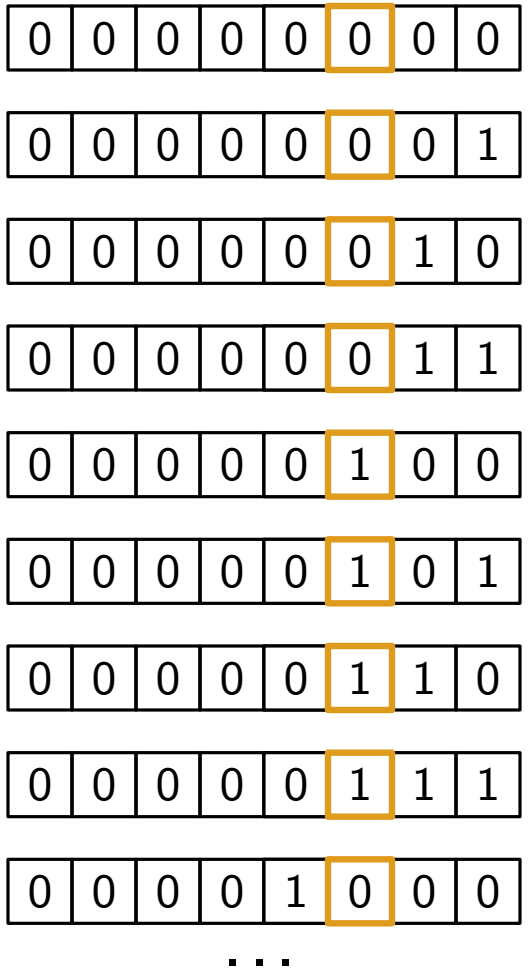
Analyse: Aggregatmethode

Idee

- berechne Gesamtkosten
- teile durch Anzahl von Operationen

Beobachtung

- nicht jedes Bit flipt bei jeder Operation
- $A[0]$ flipt jedes mal, $A[1]$ jedes zweite, etc.
- $A[i]$ flipt bei jedem 2^i-ten Inkrement
- Bei n Aufrufen: $A[i]$ flipt $\lfloor \frac{n}{2^i} \rfloor$ mal



Analyse: Aggregatmethode

Beobachtung

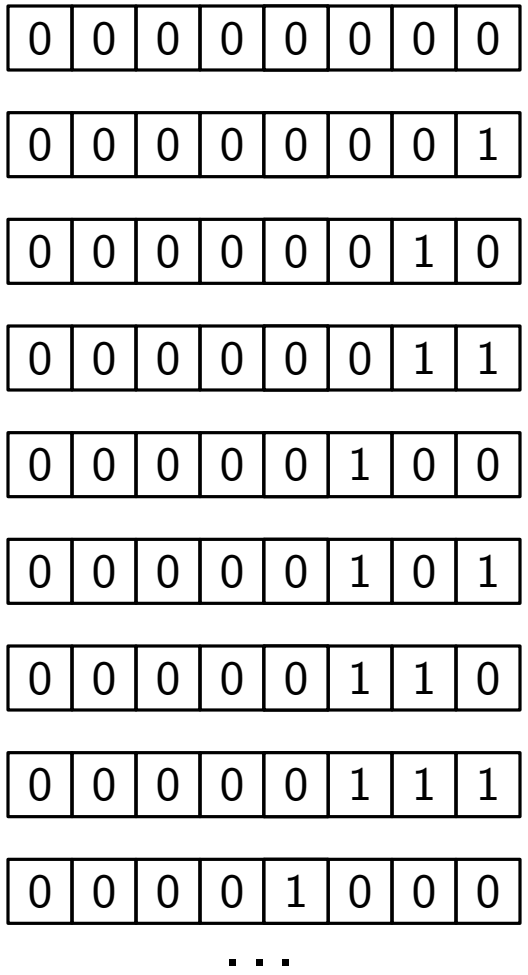
- nicht jedes Bit flipt bei jeder Operation
- $A[0]$ flipt jedes mal, $A[1]$ jedes zweite, etc.
- $A[i]$ flipt bei jedem 2^i-ten Inkrement
- Bei n Aufrufen: $A[i]$ flipt $\lfloor \frac{n}{2^i} \rfloor$ mal

In Summe ergibt sich für k bits:

$$\sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

D.h.: pro Operation nur 2 Bit-Flips

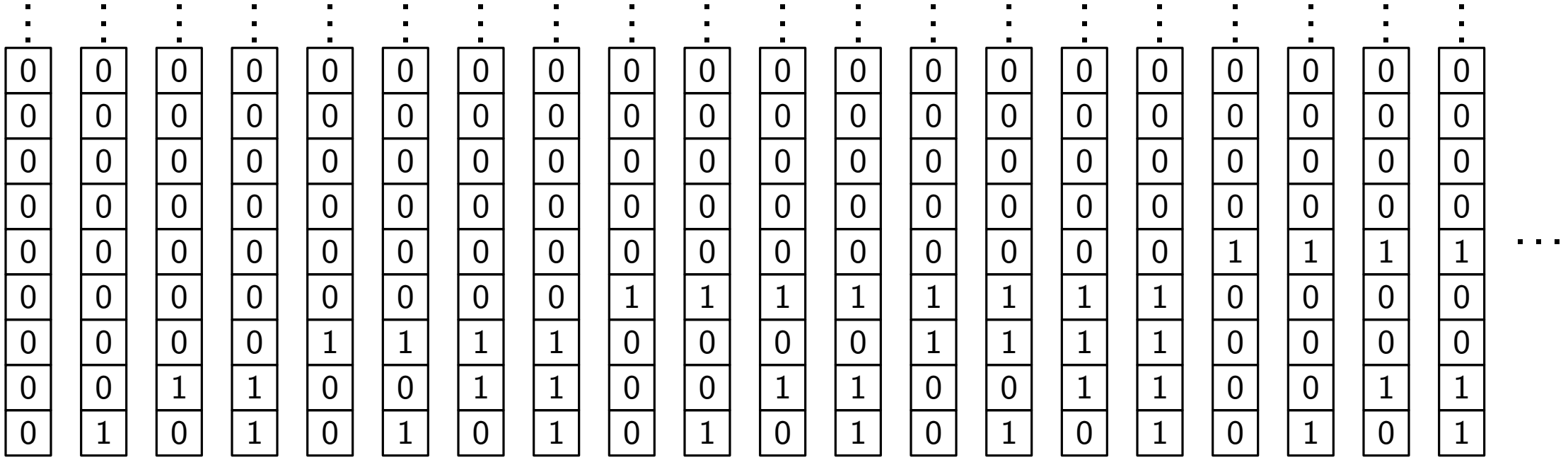
Somit: **amortisierte Kosten** in $O(1)$



Analyse mittels Charging

Idee

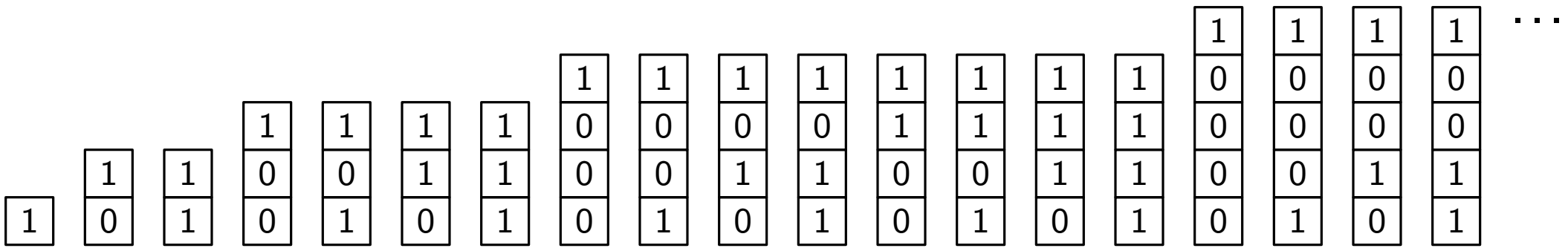
- teure Operationen legen Kosten um auf günstige Operationen



Analyse mittels Charging

Idee

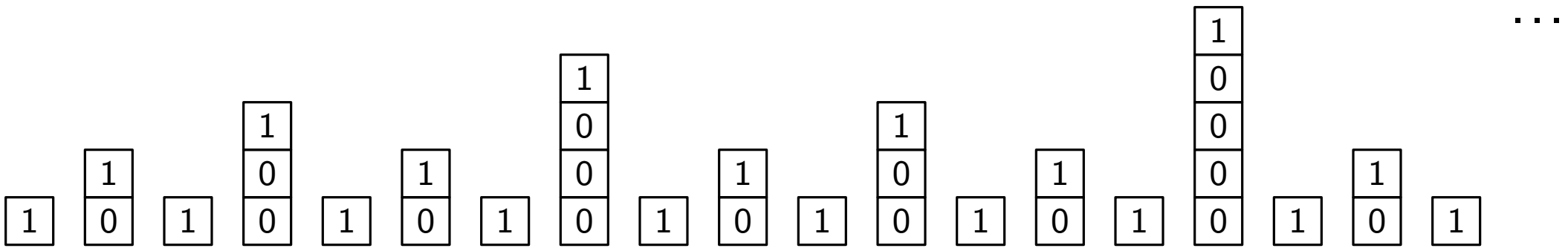
- teure Operationen legen Kosten um auf günstige Operationen



Analyse mittels Charging

Idee

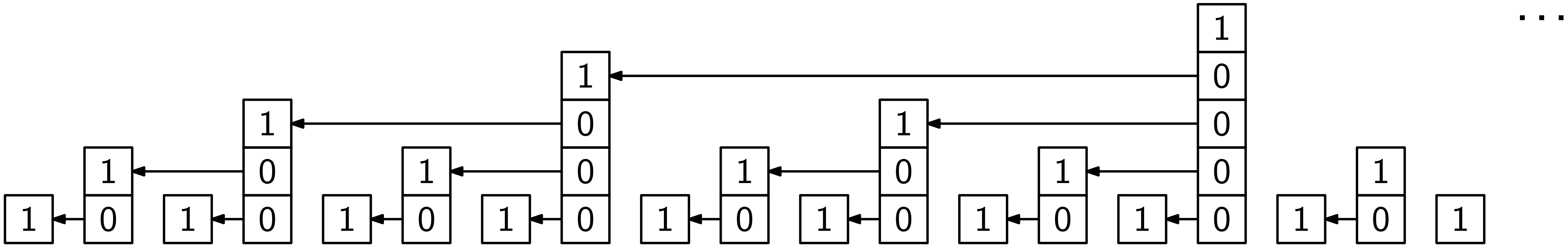
- teure Operationen legen Kosten um auf günstige Operationen



Analyse mittels Charging

Idee

- teure Operationen legen Kosten um auf günstige Operationen



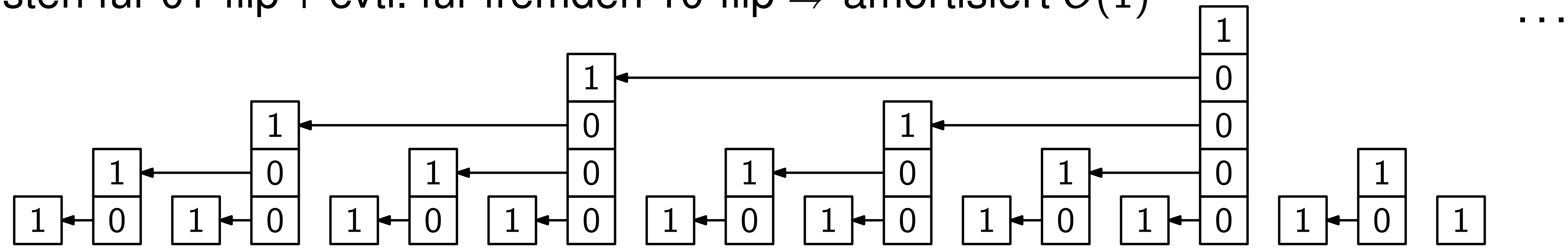
Analyse mittels Charging

Idee

- teure Operationen legen Kosten um auf günstige Operationen

Vorgehen hier:

- lege Kosten von 10-flip auf letzte Operation welche zuvor 01-flip ausgeführt hat
  - das geht für jeden 10-flip
- pro Operation übrig:
  - Kosten für 01-flip + evtl. für fremden 10-flip $\Rightarrow$ amortisiert $O(1)$



Analyse: Potentialmethode

Idee

- definiere Potentialfunktion $\Phi(D_i)$
  - Maß für Unaufgeräumtheit von D zum Zeitpunkt i
- definiere amortisierte Kosten $\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\text{Anstieg des Potenzials}}$

tatsächliche Kosten

Analyse: Potentialmethode

Idee

- definiere Potentialfunktion $\Phi(D_i)$
  - Maß für Unaufgeräumtheit von D zum Zeitpunkt i
- definiere amortisierte Kosten $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

Definition sinnvoll?

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} (c_i) + \sum_{i=1}^{n} (\Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} (c_i) + \Phi(D_n) - \Phi(D_0)$$

Falls $\Phi(D_n) \geq \Phi(D_0)$: amortisierte Kosten obere Schranke für tatsächliche Kosten

$\Rightarrow$ fordere $\Phi(D_i) \geq \Phi(D_0)$ f.a. $i > 0$ oder: $\Phi(D_0) = 0$ und $\Phi(D_i) \geq 0$ f.a. $i > 0$

Analyse: Potentialmethode

Potentialfunktion für Binärzähler:

- def. $\Phi(D_i) = \sum_{j=0}^{k-1} A[j]$ (Anzahl 1-bits zum Zeitpunkt i)

Dann ergibt sich:

- $\Phi(D_0) = 0$, $\Phi(D_i) \geq 0$ f.a. $i > 0$ *gültige Pot.fun.* :)
- tatsächliche Kosten: $c_i = \#01\text{-flips} + \#10\text{-flips}$
- Potentialänderung: $\#01\text{-flips} - \#10\text{-flips}$
- amort. Kosten: $\hat{c}_i = 2 \cdot \#01\text{-flips} \leq 2$

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	0	1	1
0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	1
0	0	0	0	0	1	1	0
0	0	0	0	0	1	1	1
0	0	0	0	1	0	0	0

...

Definition: amortisierte Kosten $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

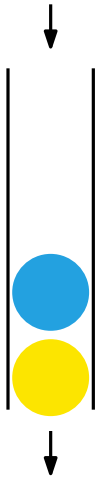
Beispiel 2: Stacks und Queues

Wiederholung

Queue

- **push**(e) $O(1)$
- **pop**() $O(1)$
- **size**() $O(1)$

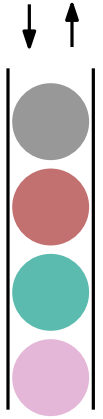
z.B. mittels verketteter Liste



Stack

- **push**(e) $O(1)$
- **pop**() $O(1)$
- **size**() $O(1)$

z.B. mittels verketteter Liste



Frage

Angenommen wir können Stacks (als Blackbox) verwenden.

(Wie) können wir daraus eine Queue bauen?

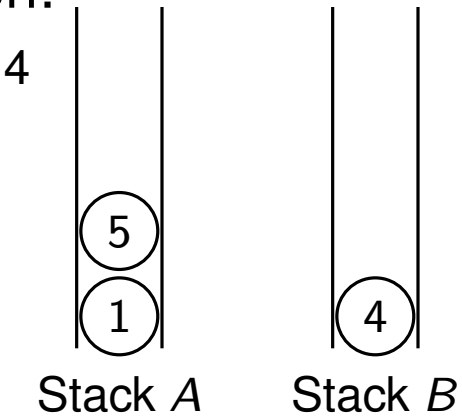
Eine Queue aus Stacks

Algorithmische Umsetzung

- **push**: auf A pushen $O(1)$
- **pop**: $O(|A|)$
  - falls B voll: von B poppen $O(1)$
  - falls B leer: alles von A nach B verschieben $O(|A|)$

Operationen:

- **push**: 3, 1, 4
- **pop**()
- **pop**()
- **push**: 1,5



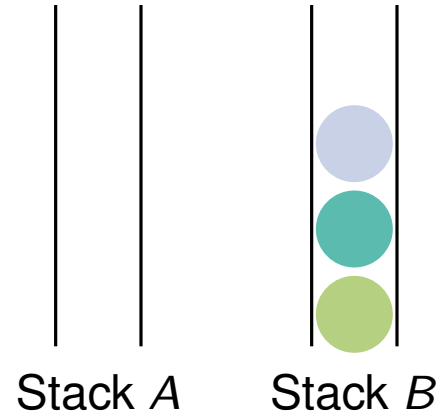
Analyse: Kontomethode

Idee

- günstige Operationen bauen Guthaben auf
- teure Operationen nutzen Guthaben

Umsetzung hier

- Gesamtkosten linear in **push** und **pop** Aufrufen
- Bei **push**: zahle 3
- Bei **pop**: zahle 1
  - Falls B leer: nutze 2 Guthaben für A.**pop()** und B.**push()** (für alle Elemente in A)
  - Falls B voll: B.**pop**() mit Kosten 1
- Konto wird nie negativ
- $3, 1 \in \Theta(1) \Rightarrow$ konstante amortisierte Kosten



Analyse: Potentialmethode

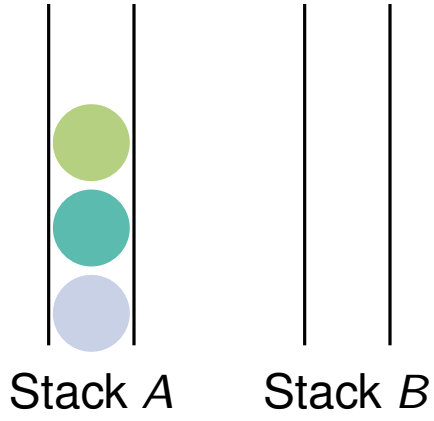
Potentialfunktion

- Definiere $\Phi(D_i) = 2 \cdot |A_i|$ (Anzahl Elemente auf A zum Zeitpunkt i)

Analyse

- $\Phi(D_0) = 0$, $\Phi(D_i) \geq 0$ f.a. $i > 0$ (d.h. gültige Potentialfunktion)
- amortisierte Kosten **push**:
  - $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 3$
- amortisierte Kosten **pop**:
  - $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1$
  - $c_i = 2 \cdot |A_{i-1}| + 1$
  - $\Phi(D_i) - \Phi(D_{i-1}) = -2|A_{i-1}|$

} amortisierte Kosten in $O(1)$



Definition: amortisierte Kosten $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$