Algorithmen und Datenstrukturen

Kapitel 3: Sortieren

Prof. Dr. Peter Kling

Wintersemester 2020/21

Übersicht

1. Insertionsort
2. Mergesort
3. Rekursion
4. Quicksort
5. Heapsort
6. Untere Schranke für Vergleichssortierer
7. Sortieren in linearer Zeit

Das Sortierproblem

Eingabe

- Folge von n Zahlen $(a_1, a_2, \dots, a_n)$

Ausgabe

- Umordnung $(b_1, b_2, \dots, b_n)$ mit $b_1 \leq b_2 \leq \dots b_n$

Beispiel

- **Eingabe:** $(7, 99, 12, \ 3, 17, \ 12)$
- **Ausgabe:** $(3, \ \ 7, 12, 12, 17, 99)$

1) Insertionsort

Definition 3.1

Ein inkrementeller Algorithmus berechnet eine Teillösung für die ersten i Objekte sukzessive für $i \in \{1, 2, \ldots, n\}$ aus einer bekannten Teillösung für die ersten $i - 1$ Objekte.

MinSearch(A)

```
min ← 1
for i ← 2 to length(A)
    if A[i] < A[min]
        min ← i
return min
```

- **Objekte:**
  Einträge des Arrays A
- **Teillösung für ersten i Objekte:**
  Minimum von $A[1], \ldots, A[i]$

Idee

Berechne sukzessive die Sortierungen der Teilarrays $A[1 \dots i]$ für $i \in \{1, 2, \dots, \mathrm{length}(A)\}$.

Algorithmus 3.1: INSERTIONSORT(A)

```
for j ← 2 to length(A)
    key ← A[j]
    i ← j − 1
    while i > 0 and A[i] > key
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← key
```

Beispiel

$$key = \boxed{99}$$

$$A = \langle \overbrace{\underset{\substack{\uparrow \\ i}}{7}}^{\text{sortiert}}, \underset{\substack{\uparrow \\ j}}{99}, 12, 3, 17, 12 \rangle$$

Was ist die Grundidee des Algorithmus?

- betrachte Variable $key \leftarrow A[j]$ im j-Schleifendurchlauf
- while: schiebe alle $A[1], \ldots, A[j-1]$ die größer key sind...
- ...um eins nach rechts
- key wird in entstandener Lücke gespeichert

```
INSERTIONSORT(A)
for j ← 2 to length(A)
    key ← A[j]
    i ← j − 1
    while i > 0 and A[i] > key
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← key
```

Schleifendurchlauf mit $j = 2$

$key = 99$

$A = \langle\ 7\ ,\ 99,\ 12\ ,\ 3\ ,\ 17\ ,\ 12\ \rangle$

Wie gut ist InsertionSort?

Theorem 3.1

InsertionSort löst das Sortierproblem. Das heißt der Algorithmus sortiert eine Folge von n Zahlen aufsteigend.

Theorem 3.2

Die worst-case Laufzeit von InsertionSort ist $\Theta(n^2)$.

Beweis von Theorem 3.1 (1/3)

- sei das Eingabearray $A = \langle a_1, a_2, \ldots, a_n \rangle$

Schleifeninvariante $I(j)$

$A[1 \ldots j-1]$ enthält die Zahlen $a_1, a_2, \ldots, a_{j-1}$ aufsteigend sortiert

(a) **Initialisierung:** ✓
- das einelementiges Array $A[1 \ldots 2-1] = A[1]$ ist sortiert
- also gilt $I(2)$ trivialerweise immer

$\Longrightarrow$ $I(2)$ gilt vor dem ersten for-Schleifendurchlauf

(b) **Erhaltung:** ⁉

(c) **Terminierung:** ✓
- am Ende der Schleife gilt $I(\mathbf{length}(A)+1) = I(n+1)$
- das heißt $A[1 \ldots n+1-1] = A[1 \ldots n]$ enthält die Zahlen…
- …$a_1, a_2, \ldots a_{n+1-1} = a_n$ aufsteigend sortiert

$\Longrightarrow$ InsertionSort ist korrekt

Beweis der Erhaltung: $I(j) \to I(j+1)$ (✓) Details auf nächster Folie

- gelte $I(j)$ am Anfang des j-Durchlaufs der for-Schleife
- InsertionSort merkt sich $A[j]$ in Variable *key*
- sei $k \in \{1, 2, \ldots, j-1\}$ minimal mit $A[k] > key$...
  - ...oder $k = j$ falls ein solches k nicht existiert
- der Algorithmus verschiebt $A[k \ldots j-1]$ nach $A[k+1 \ldots j]$...
- ...und setzt anschließend $A[k]$ auf den Wert *key*
- danach gilt:

  (1) $A[1] \leq A[2] \leq \cdots \leq A[k-1]$ wg. $I(j)$

  (2) $A[k-1] \leq A[k] \leq A[k+1]$ while-Schleife

  (3) $A[k+1] \leq A[k+2] \leq \cdots \leq A[j]$ wg. $I(j)$

$\implies$ $A[1] \leq A[2] \leq \cdots \leq A[j]$

$\implies$ $I(j+1)$ gilt am Ende des j-Durchlaufs der for-Schleife

Beweis von Theorem 3.1 (3/3)

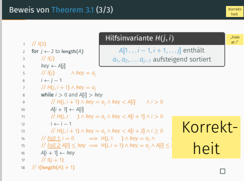
Hilfsinvariante $H(j,i)$

$A[1\ldots i-1, i+1, \ldots j]$ enthält
$a_1, a_2, \ldots a_{j-1}$ aufsteigend sortiert

„hole at i“

$// \; I(2)$
for $j \leftarrow 2$ to **length**(A)
 $// \; I(j)$
 $key \leftarrow A[j]$
 $// \; I(j) \qquad \wedge\ key = a_j$
 $i \leftarrow j-1$
 $// \; H(j, i+1) \wedge key = a_j$
 while $i > 0$ and $A[i] > key$
  $// \; H(j, i+1) \wedge key = a_j \wedge key < A[i] \qquad \wedge\ i > 0$
  $A[i+1] \leftarrow A[i]$
  $// \; H(j, i \qquad) \wedge key = a_j \wedge key < A[i+1] \wedge i > 0$
  $i \leftarrow i-1$
  $// \; H(j, i+1) \wedge key = a_j \wedge key < A[i+2] \wedge i \geq 0$
 $//$ Fall 1: $i = 0 \qquad \Longrightarrow\ H(j, 1 \qquad) \wedge key = a_j \wedge \qquad key < A[2]$
 $//$ Fall 2: $A[i] \leq key \ \Longrightarrow\ H(j, i+1) \wedge key = a_j \wedge A[i] \leq key < A[i+2]$
 $A[i+1] \leftarrow key$
 $// \; I(j+1)$
$// \; I(\mathbf{length}(A)+1)$

$\square$



- Initialisierung (Zeile 1) & Terminierung (Zeile 18) $\rightsquigarrow$ vorherige Folie
- hier im Wesentlichen die Erhaltung
- benötigen weitere (Hilfs-) Invariante für innere while-Schleife
- genauere Erläuterungen mündlich und/oder annotiert

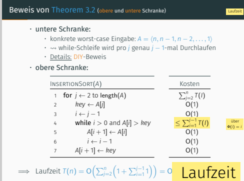
Beweis von Theorem 3.2 (obere und untere Schranke)

- **untere Schranke:**
  - konkrete worst-case Eingabe: $A = \langle n, n-1, n-2, \ldots, 1 \rangle$
  - $\rightsquigarrow$ while-Schleife wird pro j genau $j-1$-mal Durchlaufen
  - Details: DIY-Beweis
- **obere Schranke:**

InsertionSort(A)	Kosten
for $j \leftarrow 2$ to length(A)	$\sum_{j=2}^{n} T(I)$
$key \leftarrow A[j]$	$O(1)$
$i \leftarrow j-1$	$O(1)$
while $i > 0$ and $A[i] > key$	$\leq \sum_{i=1}^{j-1} T(I)$
$A[i+1] \leftarrow A[i]$	$O(1)$
$i \leftarrow i-1$	$O(1)$
$A[i+1] \leftarrow key$	$O(1)$

über $\Phi(i) = i$

$\Longrightarrow$ Laufzeit $T(n) = O\left(\sum_{j=2}^{n}\left(1 + \sum_{i=1}^{j-1} 1\right)\right) = O(n^2)$ $\square$



- Laufzeit der while-Schleife folgt mittels Potentialfunktion $\Phi(i) = i$

2) Mergesort

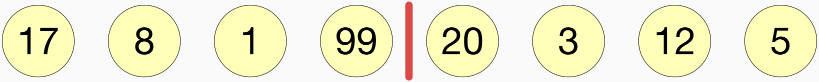
Definition 3.2

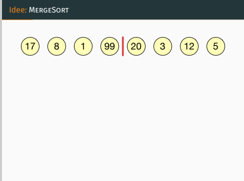
Ein Divide & Conquer Algorithmus nutzt Rekursion zur Lösung eines Problems in drei Schritten:

Teile & Erobere

1. Teile das Problem in mehrere Teilprobleme auf.
2. Erobere große Teilproblem durch rekursive Aufrufe und löse kleine Teilprobleme direkt.
3. Kombiniere die Lösungen der Teilprobleme zu einer Gesamtlösung.

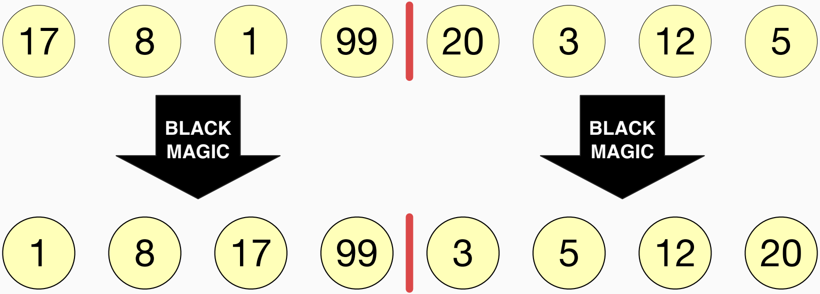
Idee: MergeSort



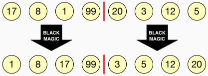


- Teile: rote Linie
- Erobere: Black Magic bzw. Mathematik
- Kombiniere: Merge

Idee: MergeSort

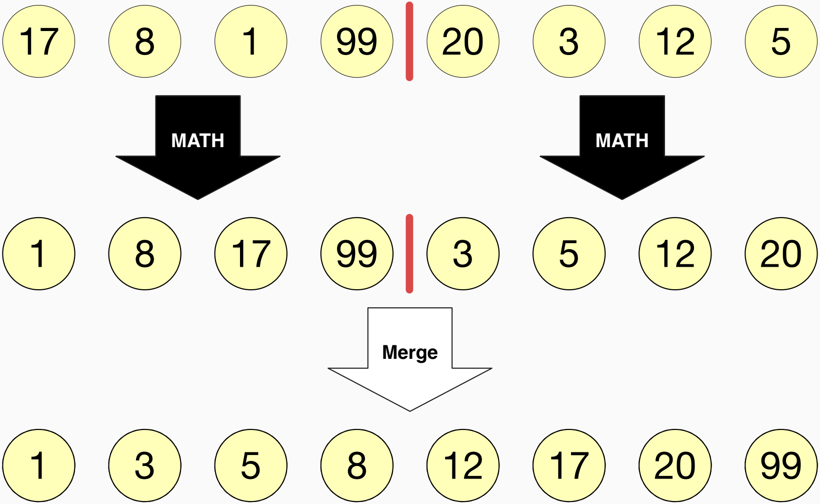


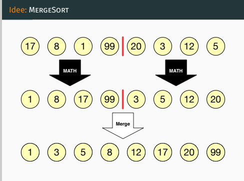
Idee: MergeSort



- Teile: rote Linie
- Erobere: Black Magic bzw. Mathematik
- Kombiniere: Merge

Idee: MergeSort





- Teile: rote Linie
- Erobere: Black Magic bzw. Mathematik
- Kombiniere: Merge

Pseudocode zu MergeSort

Algorithmus 3.2: MergeSort(A, l, r)

```
if l < r
    p ← ⌊(l + r)/2⌋
    MergeSort(A, l, p)
    MergeSort(A, p + 1, r)
    Merge(A, l, p, r)
```

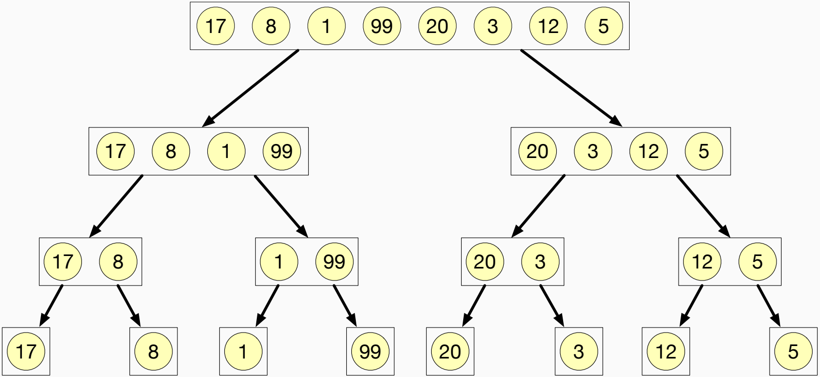
- erstmaliger Aufruf als MergeSort$(A, 1, \text{length}(A))$
- Hilfsalgorithmus Merge mischt zwei sortierte Teilfolgen
- eine Mögliche Umsetzung des D&C Ansatzes zum Sortieren

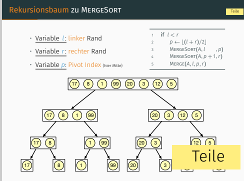
Rekursionsbaum zu MergeSort

Teile

- Variable l: linker Rand
- Variable r: rechter Rand
- Variable p: Pivot Index (hier Mitte)

$$
\begin{array}{l}
\textbf{if } l < r \\
\quad p \leftarrow \lfloor (l + r)/2 \rfloor \\
\quad \textsc{MergeSort}(A, l, p) \\
\quad \textsc{MergeSort}(A, p+1, r) \\
\quad \textsc{Merge}(A, l, p, r)
\end{array}
$$





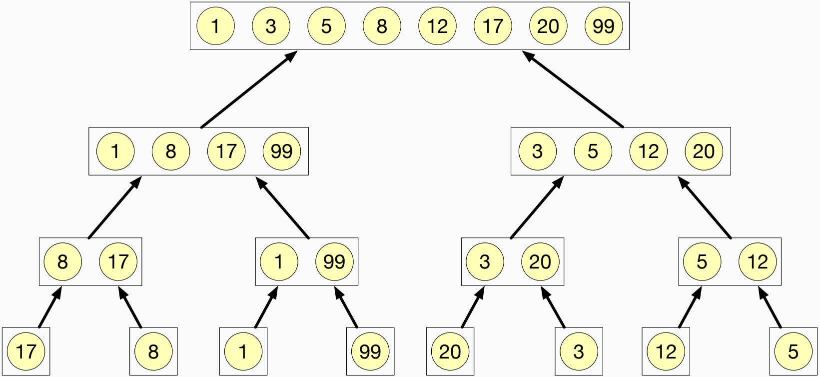
- MergeSort teilt das Array in der Mitte
- andere Teilungsstrategien denkbar; werden wir noch sehen
- Pivot Index nicht mit Pivot Element verwechseln; kommt später

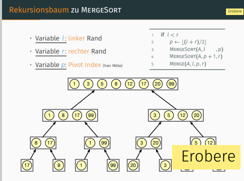
Rekursionsbaum zu MERGESORT

Erobere

- Variable l: linker Rand
- Variable r: rechter Rand
- Variable p: Pivot Index (hier Mitte)

```
if l < r
    p ← ⌊(l + r)/2⌋
    MERGESORT(A, l    , p)
    MERGESORT(A, p + 1, r)
    MERGE(A, l, p, r)
```





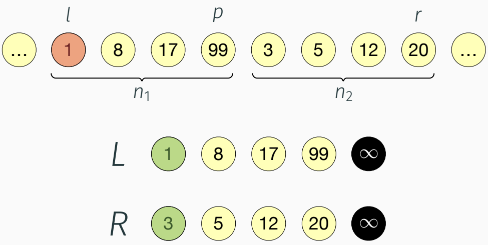
- MergeSort teilt das Array in der Mitte
- andere Teilungsstrategien denkbar

Wie genau funktioniert Merge?

Algorithmus 3.3: Merge(A, l, p, r)

$n_1 \leftarrow p - l + 1$
$n_2 \leftarrow r - p$
for $i \leftarrow 1$ to n_1: $L[i] \leftarrow A[l + i - 1]$
for $j \leftarrow 1$ to n_2: $R[j] \leftarrow A[p + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1; j \leftarrow 1$
for $k \leftarrow l$ to r
  if $L[i] \leq R[j]$
    $A[k] \leftarrow L[i]$
    $i \leftarrow i + 1$
  else
    $A[k] \leftarrow R[j]$
    $j \leftarrow j + 1$

- Variablen n_1, n_2: Länge der Teillösungen
- Variablen L, R: Arrays mit Teillösungen
- Variablen i, j, k: „Merge-Indizes“



Wie gut ist MERGESORT?

Theorem 3.3

MERGESORT löst das Sortierproblem. Das heißt der Algorithmus sortiert eine Folge von n Zahlen aufsteigend.

Theorem 3.4

Die Laufzeit von MERGESORT ist $\Theta(n \cdot \log n)$.

Wie gut ist MergeSort?

Theorem 3.3
MergeSort löst das Sortierproblem. Das heißt der Algorithmus sortiert eine Folge von n Zahlen aufsteigend.

Theorem 3.4
Die Laufzeit von MergeSort ist $\Theta(n \cdot \log n)$.

- wir reden hier explizit nicht von worst-case Laufzeit
- d.h. MergeSort hat selbst im best-case Laufzeit $\Theta(n \cdot \log n)$

Wie beweist man Korrektheit rekursiver Algorithmen?

Üblicherweise ähnlich zur vollständigen Induktion

1. Initialisierung:
   Algorithmus ist korrekt für Basisfall
2. Erhaltung:
   rekursiver Aufruf korrekt $\implies$ aktueller Aufruf korrekt

Anmerkung zur Erhaltung

- die Annhame der Korrektheit der rekursiven Aufrufe...
- ...setzt Terminierung voraus!

$\implies$ Müssen wir zeigen! (oder direkt Laufzeitanalyse machen)

Beweis von Theorem 3.3 (1/2)

Korrektheit

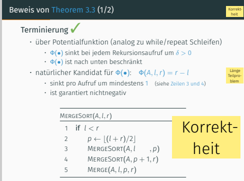
Terminierung ✓

- über Potentialfunktion (analog zu while/repeat Schleifen)
  - $\Phi(\bullet)$ sinkt bei jedem Rekursionsaufruf um $\delta > 0$
  - $\Phi(\bullet)$ ist nach unten beschränkt
- natürlicher Kandidat für $\Phi(\bullet)$: $\quad \Phi(A, l, r) = r - l$
  - sinkt pro Aufruf um mindestens 1 (siehe Zeilen 3 und 4)
  - ist garantiert nichtnegativ

Länge Teilproblem

```
MergeSort(A, l, r)

1  if l < r
2      p ← ⌊(l + r)/2⌋
3      MergeSort(A, l, p)
4      MergeSort(A, p + 1, r)
5      Merge(A, l, p, r)
```



- δ sollte nicht von der Rekursionstiefe abhängen
- analog kann $\Phi(\bullet)$ steigen und nach oben beschränkt sein
- $\Phi(A, l, r)$ halbiert sich sogar (im Wesentlich) pro Aufruf!
- implizit nehmen wir hier die Terminierung von Merge an
- formal zeigen wir die Terminierung von Merge in Lemma 3.2

Beweis von Theorem 3.3 (2/2)

Korrektheit

Initialisierung & Erhaltung (✓)

- Behauptung: MergeSort(A, l, r) sortiert $A[l \dots r]$
- Initialisierung: Basisfall $l \geq r$ ist trivialerweise sortiert

nur ein Element

- Erhaltung:
  - nach rekursiven Aufrufen sind $A[l \dots p]$ und $A[p+1, r]$ sortiert

$\Longrightarrow$ wenn Merge(A, l, p, r) diese Teillösungen...
...korrekt zusammenführt, so ist $A[l \dots r]$ am Ende sortiert

```
MergeSort(A, l, r)

if l < r
    p ← ⌊(l + r)/2⌋
    MergeSort(A, l    , p)
    MergeSort(A, p + 1, r)
    Merge(A, l, p, r)
```

□

Müssen also noch MERGE analysieren!

Lemma 3.1

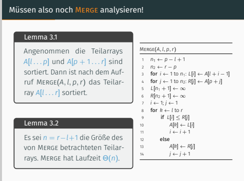
Angenommen die Teilarrays $A[l \ldots p]$ und $A[p+1 \ldots r]$ sind sortiert. Dann ist nach dem Aufruf $\textsc{Merge}(A, l, p, r)$ das Teilarray $A[l \ldots r]$ sortiert.

Lemma 3.2

Es sei $n = r - l + 1$ die Größe des von MERGE betrachteten Teilarrays. MERGE hat Laufzeit $\Theta(n)$.

```
MERGE(A, l, p, r)

n₁ ← p − l + 1
n₂ ← r − p
for i ← 1 to n₁: L[i] ← A[l + i − 1]
for j ← 1 to n₂: R[j] ← A[p + j]
L[n₁ + 1] ← ∞
R[n₂ + 1] ← ∞
i ← 1; j ← 1
for k ← l to r
    if L[i] ≤ R[j]
        A[k] ← L[i]
        i ← i + 1
    else
        A[k] ← R[j]
        j ← j + 1
```



- auch hier: selbst im best-case $\Theta(n)$

Beweis von Lemma 3.1 (1/2)

Korrekt-heit

Schleifeninvariante $I(i, j, k)$

$A[l \ldots k-1]$ enthält die $k-l$ kleinsten Zahlen aus L und R in sortierter Reihenfolge. Außerdem sind $L[i]$ und $R[j]$ die kleinsten noch nicht nach A kopierten Elemente.

(a) **Initialisierung:** ✓

- die Aussage $I(1, 1, l)$ gilt trivialerweise

$\Longrightarrow$ $I(1, 1, l)$ gilt vor dem ersten Schleifendurchlauf

(b) **Erhaltung:** ⁉

(c) **Terminierung:** ✓

- am Ende der Schleife gilt $I(\bullet, \bullet, r+1)$

$\Longrightarrow$ $A[l \ldots r]$ enthält die $r-l+1$ kleinsten Zahlen aus L und R...
...in sortierter Reihenfolge

Also alle!

$\Longrightarrow$ MERGE ist korrekt

Beweis von Lemma 3.1 (2/2)


```
MERGE(A, l, p, r)
1   n1 ← p − l + 1
2   n2 ← r − p
3   for i ← 1 to n1: L[i] ← A[l + i − 1]
4   for j ← 1 to n2: R[j] ← A[p + j]
5   L[n1 + 1] ← ∞
6   R[n2 + 1] ← ∞
7   i ← 1; j ← 1
8   // I(i, j, l)
9   for k ← l to r
10      // I(i, j, k)
11      if L[i] ≤ R[j]
12          // I(i, j, k) ∧ L[i] ≤ R[j]
13          A[k] ← L[i]
14          i ← i + 1
15          // I(i, j, k + 1)
16      else
17          // I(i, j, k) ∧ L[i] > R[j]
18          A[k] ← R[j]
19          j ← j + 1
20          // I(i, j, k + 1)
21      // I(i, j, k + 1)
22  // I(•, •, r + 1)
```

Schleifeninvariante $I(i, j, k)$

$A[l \dots k-1]$ enthält die $k - l$ kleinsten Zahlen aus L und R in sortierter Reihenfolge. Außerdem sind $L[i]$ und $R[j]$ die kleinsten noch nicht wieder nach A kopierten Elemente.

- gelte $I(i, j, k)$ vor dem k-Schleifendurchlauf
- o.B.d.A. sei $L[i] \leq R[j]$, also $L[i]$ das kleinste noch nicht einsortierte Element
  - Fall $L[i] > R[j]$ geht analog
- nach Zeile 13 enthält $A[l \dots k]$ die $k - l + 1$ kleinsten Elemente aus L und R in sortierter Reihenfolge
- nach Zeile 14 gilt dann $I(i, j, k+1)$

$\Longrightarrow$ $I(i, j, k+1)$ gilt am Ende des k-Schleifendurchlaufs $\square$

✓

Beweis von Lemma 3.2

$\textsc{Merge}(A, l, p, r)$	Kosten
$n_1 \leftarrow p - l + 1$	$\Theta(1)$
$n_2 \leftarrow r - p$	$\Theta(1)$
for $i \leftarrow 1$ to n_1: $L[i] \leftarrow A[l + i - 1]$	$\Theta(n_1)$
for $j \leftarrow 1$ to n_2: $R[j] \leftarrow A[p + j]$	$\Theta(n_2)$
$L[n_1 + 1] \leftarrow \infty$	$\Theta(1)$
$R[n_2 + 1] \leftarrow \infty$	$\Theta(1)$
$i \leftarrow 1; j \leftarrow 1$	$\Theta(1)$
for $k \leftarrow l$ to r	$\Theta(r - l)$
if $L[i] \leq R[j]$	$\Theta(1)$
$A[k] \leftarrow L[i]$	$\Theta(1)$
$i \leftarrow i + 1$	$\Theta(1)$
else	$\Theta(1)$
$A[k] \leftarrow R[j]$	$\Theta(1)$
$j \leftarrow j + 1$	$\Theta(1)$

- Eingabegröße $n = r - l + 1$
- Behauptung: Laufz. $\Theta(n)$
- $n_1 + n_2 = r - l + 1 = n$
- exakt $r - l + 1 = n$ Durchläufe der for-Schleife
- alle anderen Operationen haben Laufzeit $\Theta(1)$
- also hat $\textsc{Merge}$ Laufzeit $\Theta(n)$

✓

□

Es bleibt die Laufzeit von MergeSort zu beweisen!

Laufzeitanalyse für D&C Algorithmen

Die Laufzeit eines D&C Algorithmus lässt sich beschränken durch

$$T(n) \leq \begin{cases} c_B & \text{, falls } n \leq n_B, \\ a \cdot T(n/b) + D(n) + C(n) & \text{, sonst.} \end{cases}$$

Dabei ist:

- $T(n)$: worst-case Laufzeit bei Eingabegröße n
- c_B & n_B: Basisfälle haben Größe $\leq n_B$ und Laufzeit $\leq c_B$
- a: Anzahl der Teilprobleme durch Teilung
- n/b: Größe der Teilprobleme
- $D(n)$: Laufzeit für die Teilung
- $C(n)$: Laufzeit für die Kombinierung

Divide

Combi-
ne

Rekursionsformel für MergeSort

obere Schranke

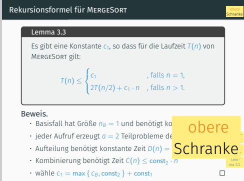
Lemma 3.3

Es gibt eine Konstante c_1, so dass für die Laufzeit $T(n)$ von MergeSort gilt:

$$T(n) \leq \begin{cases} c_1 & \text{, falls } n = 1, \\ 2T(n/2) + c_1 \cdot n & \text{, falls } n > 1. \end{cases}$$

Beweis.

- Basisfall hat Größe $n_B = 1$ und benötigt konstante Zeit c_B
- jeder Aufruf erzeugt $a = 2$ Teilprobleme der Größe $\approx n/2$ vereinfacht
- Aufteilung benötigt konstante Zeit $D(n) = \text{const}_1$ 2 rek. Aufrufe
- Kombinierung benötigt Zeit $C(n) \leq \text{const}_2 \cdot n$ Lemma 3.2
- wähle $c_1 = \max\{c_B, \text{const}_2\} + \text{const}_1$ □



- wir gehen hier vereinfachend davon aus, dass die Länge der Eingabe einer Zweierpotenz ist

Rekursionsformel für MERGESORT

untere Schranke

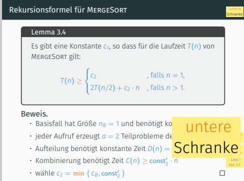
Lemma 3.4

Es gibt eine Konstante c_1, so dass für die Laufzeit $T(n)$ von MERGESORT gilt:

$$T(n) \geq \begin{cases} c_2 & \text{, falls } n = 1, \\ 2T(n/2) + c_2 \cdot n & \text{, falls } n > 1. \end{cases}$$

Beweis.

- Basisfall hat Größe $n_B = 1$ und benötigt konstante Zeit c_B
- jeder Aufruf erzeugt $a = 2$ Teilprobleme der Größe $\approx n/2$ — vereinfacht
- Aufteilung benötigt konstante Zeit $D(n) = \text{const}_1$ — 2 rek. Aufrufe
- Kombinierung benötigt Zeit $C(n) \geq \text{const}_2' \cdot n$ — Lemma 3.2
- wähle $c_2 = \min \{ c_B, \text{const}_2' \}$ $\square$



Rekursionsformel für MERGESORT

untere Schranke

Lemma 3.4

Es gibt eine Konstante c_1, so dass für die Laufzeit $T(n)$ von MERGESORT gilt:

$$T(n) \geq \begin{cases} c_2 & \text{, falls } n = 1, \\ 2T(n/2) + c_2 \cdot n & \text{, falls } n > 1. \end{cases}$$

Beweis.

- Basisfall hat Größe $n_B = 1$ und benötigt ko
- jeder Aufruf erzeugt $a = 2$ Teilprobleme de
- Aufteilung benötigt konstante Zeit $D(n) =$
- Kombinierung benötigt Zeit $C(n) \geq \mathrm{const}'_2 \cdot n$
- wähle $c_2 = \min \{ c_B, \mathrm{const}'_2 \}$ □

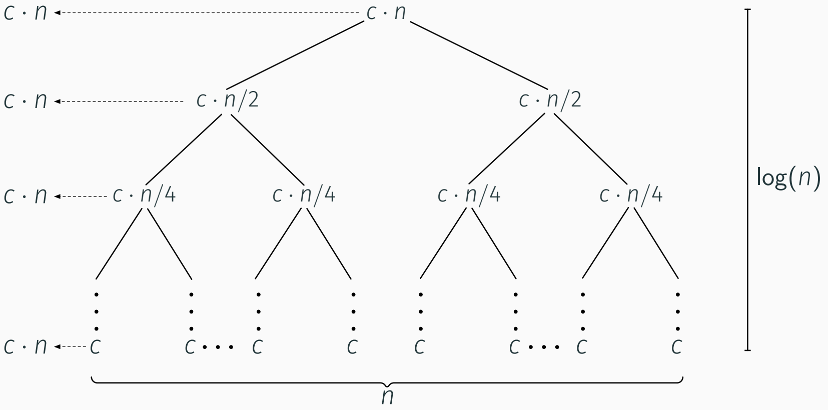
untere Schranke

Lemma 3.2

- wir gehen hier vereinfachend davon aus, dass die Länge der Eingabe einer Zweierpotenz ist

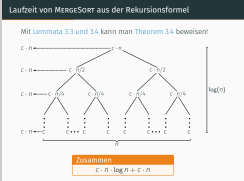
Laufzeit von MergeSort aus der Rekursionsformel

Mit Lemmata 3.3 und 3.4 kann man Theorem 3.4 beweisen!



Zusammen

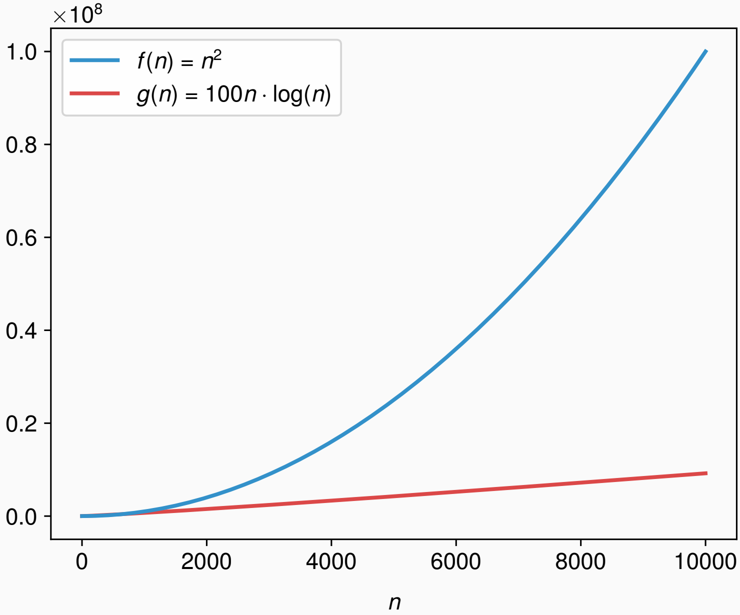
$$c \cdot n \cdot \log n + c \cdot n$$

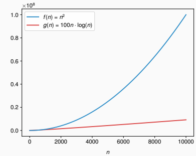


- jede Kante ist ein rekursiver Aufruf $\rightsquigarrow$ Kosten $\Theta(1)$ pro Kante
- jedes Blatt ist ein Basisfall $\rightsquigarrow$ Kosten $\Theta(1)$ pro Blatt
- lernen noch systematische Methode kennen, um die Lösung solch rekursiver Gleichungen für Laufzeiten zu berechnen

$\rightsquigarrow$ Stichwort Master Theorem

InsertionSort vs MergeSort





- n^2 wächst viel stärker als $n \cdot \log n$
- Konstanten spielen kaum eine Rolle (für große n ist asymptotische Laufzeit entscheidend)

3) Rekursion

Laufzeitanalyse rekursiver Algorithmen

- insbesondere – aber nicht nur – für D&C Algorithmen

Verschiedene Ansätze

1. Substitutionsmethode
2. Rekursionsbaum-Methode
3. Master Theorem

Beispiele rekursiver Laufzeiten

- Rekursion für FACTORIAL

$$T(n) = \begin{cases} \Theta(1) & \text{, falls } n = 1, \\ T(n-1) + \Theta(1) & \text{, falls } n > 1. \end{cases}$$

- Rekursion für MERGESORT

$$T(n) = \begin{cases} \Theta(1) & \text{, falls } n = 1, \\ 2T(n/2) + \Theta(n) & \text{, falls } n > 1. \end{cases}$$

Substitutionsmethode

Idee

- rate eine Lösung
- beweise Korrektheit über Induktion

Beispiel

$$T(n) \leq \begin{cases} c_1 & \text{, falls } n = 1, \\ T(n-1) + c_2 & \text{, falls } n > 1. \end{cases}$$

Wir rechnen...

$$\begin{aligned} T(n) &\leq T(n-1) + c_2 \\ &\leq \big(T(n-2) + c_2\big) + \ \ c_2 = T(n-2) + 2c_2 \\ &\leq \big(T(n-3) + c_2\big) + 2c_2 = T(n-3) + 3c_2 \\ &\leq \quad \vdots \end{aligned}$$

Wir raten...

$$\begin{aligned} T(n) &\leq T(1) + (n-1) \cdot c_2 \\ &\leq c_1 + (n-1) \cdot c_2. \end{aligned}$$

Korrektheit...

DIY-Induktions-Beweis!

Substitutionsmethode

Idee

- rate eine Lösung
- beweise Korrektheit über Induktion

Beispiel

$$T(n) \le \begin{cases} c_1 & \text{, falls } n = 1, \\ T(n-1) + c_2 & \text{, falls } n > 1. \end{cases}$$

Wir rechnen...

$$\begin{aligned} T(n) &\le T(n-1) + c_2 \\ &\le (T(n-2) + c_2) + \ \ c_2 = T(n-2) + 2c_2 \\ &\le (T(n-3) + c_2) + 2c_2 = T(n-3) + 3c_2 \\ &\le \ \vdots \end{aligned}$$

Wir raten...

$$\begin{aligned} T(n) &\le T(1) + (n-1) \cdot c_2 \\ &\le c_1 + (n-1) \cdot c_2. \end{aligned}$$

Korrektheit...

DIY-Induktions-Beweis!

- man kann auch gut „von unten“ anfangen
- also sukzessiv $T(1) = \dots$, $T(2) = \dots$, etc. berechnen

Laufzeit von FACTORIAL

Theorem 3.5

Die Laufzeit von FACTORIAL ist $\Theta(n)$.

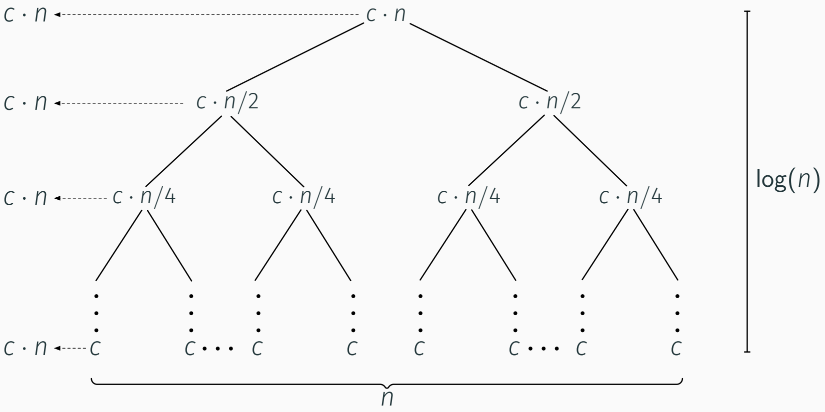
- folgt aus der eben gesehener Rekursion
- beachte: funktioniert für obere und untere Schranke
- mit der Zeit sammelt man Erfahrung & Intuition
- mehr Erfahrung $\implies$ weniger rechnen

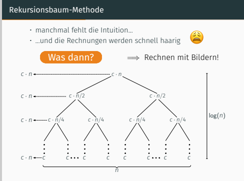
Also übt!

Rekursionsbaum-Methode

- manchmal fehlt die Intuition...
- ...und die Rechnungen werden schnell haarig

Was dann? $\Longrightarrow$ Rechnen mit Bildern!





- Rekursionsbaum aufbauen und...
  - Höhe abschätzen
  - Anzahl der Knoten pro Ebene abschätzen
  - in jeder Ebene Kosten pro Knoten abschätzen
- Anwendung 1:
  - wenn man nur eine Lösung raten möchte...
  - ...und diese später per Induktion verifiziert
- Anwendung 2:
  - wenn man sehr genau „malt"/rechnet...
  - ...dient er auch direkt als Beweis
  - (so werden wir das Master-Theorem beweisen)

Theorem 3.6: Master-Theorem

einfache Version

Es seien $a \geq 1$ und $b > 1$ Konstanten und $n = b^k \in \mathbb{N}$ für ein $k \in \mathbb{N}$. Weiterhin sei $f(n) = n$ und

$$T(n) = \begin{cases} \Theta(1) & \text{, falls } n = 1, \\ a \cdot T(n/b) + f(n) & \text{, falls } n > 1. \end{cases}$$

Dann gilt:

(a) $T(n) = \Theta(n^{\log_b a})$ falls $a > b$,

(b) $T(n) = \Theta\left(n \cdot \log n\right)$ falls $a = b$,

(c) $T(n) = \Theta(n)$ falls $a < b$.

Das (lineare) Master-Theorem

Theorem 3.6: Master-Theorem einfache Version

Es seien $a \geq 1$ und $b > 1$ Konstanten und $n = b^k \in \mathbb{N}$ für ein $k \in \mathbb{N}$. Weiterhin sei $f(n) = n$ und

$$T(n) = \begin{cases} \Theta(1) & \text{, falls } n = 1, \\ a \cdot T(n/b) + f(n) & \text{, falls } n > 1. \end{cases}$$

Dann gilt:

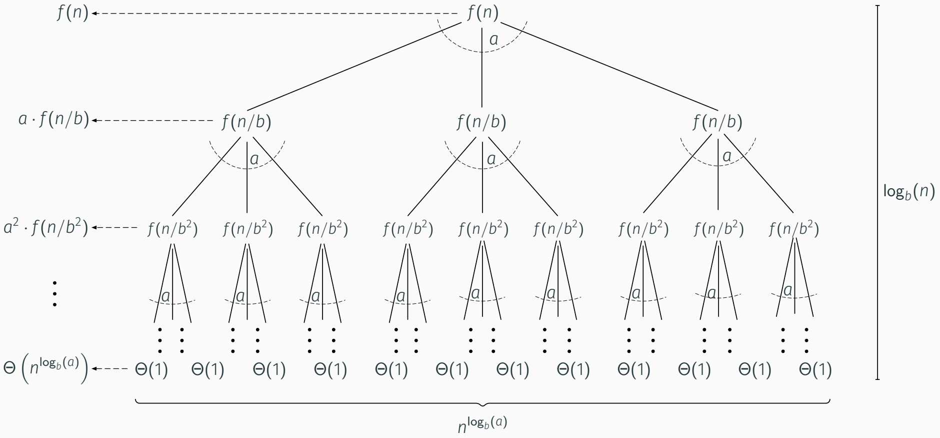
(a) $T(n) = \Theta(n^{\log_b a})$ falls $a > b$,
(b) $T(n) = \Theta(n \cdot \log n)$ falls $a = b$,
(c) $T(n) = \Theta(n)$ falls $a < b$.

- man hätte sich das f hier natürlich sparen können...
- ...und statt $f(n)$ einfach n in der Rekursion schreiben können.
  - so muss ich aber nur einen Rekursionsbaum malen... 😉

Beweis von Theorem 3.6 (1/2)

$$T(n) = \begin{cases} \Theta(1) & \text{, falls } n = 1, \\ a \cdot T(n/b) + f(n) & \text{, falls } n > 1. \end{cases}$$

- Beweis mittels Rekursionsbaum
- insgesamt erhalten wir $T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b(n)-1} a^j \cdot f(n/b^j)$



Beweis von Theorem 3.6 (2/2)

$T(n) = \begin{cases} \Theta(1) & \text{, falls } n = 1, \\ a \cdot T(n/b) + f(n) & \text{, falls } n > 1. \end{cases}$

- Beweis mittels Rekursionsbaum
- insgesamt erhalten wir $T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b(n)-1} a^i \cdot f(n/b^i)$
- einsetzen von $f(n) = n$

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b(n)-1} a^i \cdot n/b^i \\ &= \Theta(n^{\log_b a}) + n \cdot \sum_{i=0}^{\log_b(n)-1} (a/b)^i \end{aligned}$$

Falls $a = b$: $T(n) = \Theta(n^1) + n \cdot \log_b n = \Theta(n \log n)$ ✓

Falls $a \neq b$: nutze $\sum_{i=0}^{k} z^i = \frac{z^{k+1}-1}{z-1}$ für $z \neq 1$ und...
...unterscheide die verbleibenden Fälle $a > b$ und $a < b$

□

Beweis von Theorem 3.6 (2/2)

$T(n) = \begin{cases} \Theta(1) & \text{, falls } n = 1, \\ a \cdot T(n/b) + f(n) & \text{, falls } n > 1. \end{cases}$

- Beweis mittels Rekursionsbaum
- insgesamt erhalten wir $T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b(n)-1} a^i \cdot f(n/b^i)$
- einsetzen von $f(n) = n$

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b(n)-1} a^i \cdot n/b^i \\ &= \Theta(n^{\log_b a}) + n \cdot \sum_{i=0}^{\log_b(n)-1} (a/b)^i \end{aligned}$$

Falls $a = b$: $T(n) = \Theta(n^1) + n \cdot \log_b n = \Theta(n \log n)$ ✓

Falls $a \neq b$: nutze $\sum_{i=0}^{k} z^i = \frac{z^{k+1}-1}{z-1}$ für $z \neq 1$ und...
...unterscheide die verbleibenden Fälle $a > b$ und $a < b$

□

- $\sum_{i=0}^{k} z^i = \frac{z^{k+1}-1}{z-1}$ ist eine Partialsumme der geometrischen Reihe

Theorem 3.7: Master-Theorem (M-Thm.)

Es seien $a \geq 1$ und $b > 1$ Konstanten und $f(n)$ eine nichtnegative Funktion. Weiterhin sei

$$T(n) = \begin{cases} \Theta(1) & \text{, falls } n = 1, \\ a \cdot T(\ n/b\ ) + f(n) & \text{, falls } n > 1. \end{cases}$$

Dann gilt:

(a) $T(n) = \Theta(n^{\log_b a})$ falls $f(n) = O(n^{\log_b(a)-\epsilon})$ für ein $\epsilon > 0$,

(b) $T(n) = \Theta(n^{\log_b a} \cdot \log n)$ falls $f(n) = \Theta(n^{\log_b a})$,

(c) $T(n) = \Theta(f(n))$ falls $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ für ein $\epsilon > 0$ und falls $a \cdot f(n/b) \leq c \cdot f(n)$ für konstantes $c < 1$ und große n.

Das Master-Theorem

Theorem 3.7: Master-Theorem (M-Thm.)

Es seien $a \geq 1$ und $b > 1$ Konstanten und $f(n)$ eine nichtnegative Funktion. Weiterhin sei

$$T(n) = \begin{cases} \Theta(1) & \text{, falls } n = 1, \\ a \cdot T(\ n/b\ ) + f(n) & \text{, falls } n > 1. \end{cases}$$

Dann gilt:

(a) $T(n) = \Theta(n^{\log_b a})$ falls $f(n) = O(n^{\log_b(a)-\epsilon})$ für ein $\epsilon > 0$,

(b) $T(n) = \Theta(n^{\log_b a} \cdot \log n)$ falls $f(n) = \Theta(n^{\log_b a})$,

(c) $T(n) = \Theta(f(n))$ falls $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ für ein $\epsilon > 0$ und falls $a \cdot f(n/b) \leq c \cdot f(n)$ für konstantes $c < 1$ und große n.

- man schreibt das Master Theorem meist einfach mit n/b, …
- … und meint damit symbolisch entweder $\lceil n/b \rceil$ oder $\lfloor n/b \rfloor$
- Details dazu in Cormen 4.6.2

Theorem 3.7: Master-Theorem (M-Thm.)

Es seien $a \geq 1$ und $b > 1$ Konstanten und $f(n)$ eine nichtnegative Funktion. Weiterhin sei

$$T(n) = \begin{cases} \Theta(1) & \text{, falls } n = 1, \\ a \cdot T(\lceil n/b \rceil) + f(n) & \text{, falls } n > 1. \end{cases}$$

Dann gilt:

(a) $T(n) = \Theta(n^{\log_b a})$ falls $f(n) = O(n^{\log_b (a) - \epsilon})$ für ein $\epsilon > 0$,

(b) $T(n) = \Theta(n^{\log_b a} \cdot \log n)$ falls $f(n) = \Theta(n^{\log_b a})$,

(c) $T(n) = \Theta(f(n))$ falls $f(n) = \Omega(n^{\log_b (a) + \epsilon})$ für ein $\epsilon > 0$ und falls $a \cdot f(n/b) \leq c \cdot f(n)$ für konstantes $c < 1$ und große n.

Das Master-Theorem

Theorem 3.7: Master-Theorem (M-Thm.)

Es seien $a \geq 1$ und $b > 1$ Konstanten und $f(n)$ eine nichtnegative Funktion. Weiterhin sei

$$T(n) = \begin{cases} \Theta(1) & \text{, falls } n = 1, \\ a \cdot T(\lceil n/b \rceil) + f(n) & \text{, falls } n > 1. \end{cases}$$

Dann gilt:

(a) $T(n) = \Theta(n^{\log_b a})$ falls $f(n) = O(n^{\log_b(a)-\epsilon})$ für ein $\epsilon > 0$,

(b) $T(n) = \Theta(n^{\log_b a} \cdot \log n)$ falls $f(n) = \Theta(n^{\log_b a})$,

(c) $T(n) = \Theta(f(n))$ falls $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ für ein $\epsilon > 0$ und falls $a \cdot f(n/b) \leq c \cdot f(n)$ für konstantes $c < 1$ und große n.

- man schreibt das Master Theorem meist einfach mit n/b, …
- … und meint damit symbolisch entweder $\lceil n/b \rceil$ oder $\lfloor n/b \rfloor$
- Details dazu in Cormen 4.6.2

Theorem 3.7: Master-Theorem (M-Thm.)

Es seien $a \geq 1$ und $b > 1$ Konstanten und $f(n)$ eine nichtnegative Funktion. Weiterhin sei

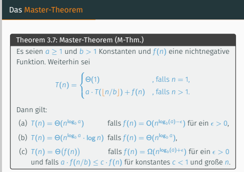
$$T(n) = \begin{cases} \Theta(1) & \text{, falls } n = 1, \\ a \cdot T(\lfloor n/b \rfloor) + f(n) & \text{, falls } n > 1. \end{cases}$$

Dann gilt:

(a) $T(n) = \Theta(n^{\log_b a})$ falls $f(n) = O(n^{\log_b(a) - \epsilon})$ für ein $\epsilon > 0$,

(b) $T(n) = \Theta(n^{\log_b a} \cdot \log n)$ falls $f(n) = \Theta(n^{\log_b a})$,

(c) $T(n) = \Theta(f(n))$ falls $f(n) = \Omega(n^{\log_b(a) + \epsilon})$ für ein $\epsilon > 0$ und falls $a \cdot f(n/b) \leq c \cdot f(n)$ für konstantes $c < 1$ und große n.



Das Master-Theorem

Theorem 3.7: Master-Theorem (M-Thm.)

Es seien $a \geq 1$ und $b > 1$ Konstanten und $f(n)$ eine nichtnegative Funktion. Weiterhin sei

$$T(n) = \begin{cases} \Theta(1) & \text{, falls } n = 1, \\ a \cdot T(\lfloor n/b \rfloor) + f(n) & \text{, falls } n > 1. \end{cases}$$

Dann gilt:

(a) $T(n) = \Theta(n^{\log_b a})$ falls $f(n) = O(n^{\log_b(a)-\epsilon})$ für ein $\epsilon > 0$,

(b) $T(n) = \Theta(n^{\log_b a} \cdot \log n)$ falls $f(n) = \Theta(n^{\log_b a})$,

(c) $T(n) = \Theta(f(n))$ falls $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ für ein $\epsilon > 0$ und falls $a \cdot f(n/b) \leq c \cdot f(n)$ für konstantes $c < 1$ und große n.

- man schreibt das Master Theorem meist einfach mit n/b, …
- … und meint damit symbolisch entweder $\lceil n/b \rceil$ oder $\lfloor n/b \rfloor$
- Details dazu in Cormen 4.6.2

Ist einfacher als es aussieht...

- gegeben Rekursion der Form $T(n) = a \cdot T(n/b) + f(n)$
- Vergleiche die Funktionen $f(n)$ und $n^{\log_b(a)}$

(a) $\underline{f(n) = O(n^{\log_b(a)-\epsilon})}$:

- d. h. $f(n)$ ist polynomiell kleiner als $n^{\log_b(a)}$
- Lösung ist $\Theta(n^{\log_b(a)})$

(b) $\underline{f(n) = \Theta(n^{\log_b(a)})}$:

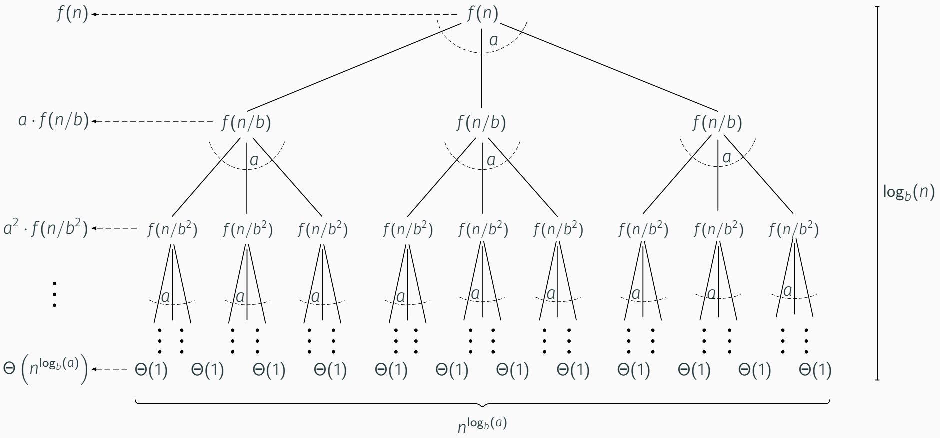
- d. h. $f(n)$ und $n^{\log_b(a)}$ asymptotisch gleich gros
- Lösung ist $\Theta(n^{\log_b(a)} \cdot \log n) = \Theta(f(n) \cdot \log n)$

(c) $\underline{f(n) = \Omega(n^{\log_b(a)-\epsilon})}$:

- d. h. $f(n)$ ist polynomiell größer als $n^{\log_b(a)}$
- Lösung ist $\Theta(f(n))$, wenn „Regularitätsbedingung" erfüllt ist
- also falls $a \cdot f(n/b) \leq c \cdot f(n)$ für $c < 1$ und große n

(Lösung ist also im Wesentlichen die größere der Funktionen, ggfs. mit log-Faktor)

Beweisidee zu Theorem 3.7



+ mehr Rechnerei
(aber selbe Grundidee)

Beispiele

- <u>MergeSort:</u> $a = b = 2$ und $f(n) = \Theta(n)$
  - $n^{\log_b a} = n \implies f(n) = \Theta(n^{\log_b a})$

$\implies$ (M-Thm. (b)) $T(n) = \Theta(n \cdot \log n)$

- <u>$T(n) = 9T(n/3) + n$:</u> $a = 9$, $b = 3$ und $f(n) = n$
  - $n^{\log_b a} = n^2 \implies f(n) = O(n^{\log_b(a)-\epsilon})$ für $\epsilon = 1$

$\implies$ (M-Thm. (a)) $T(n) = \Theta(n^2)$

- <u>$T(n) = 3T(n/4) + n \cdot \log n$:</u> $a = 3$, $b = 4$ und $f(n) = n \cdot \log n$
  - $n^{\log_b a} = O(n^{0.793}) \implies f(n) = \Omega(n^{\log_b(a)+\epsilon})$ für $\epsilon \approx 0.2$
  - außerdem gilt

$$a \cdot f(n/b) = (3/4) \cdot n \cdot \log n \leq (3/4) \cdot n \cdot \log n = (3/4) \cdot f(n)$$

$\implies$ (M-Thm. (c)) $T(n) = \Theta(n \cdot \log n)$

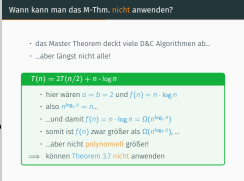
Wann kann man das M-Thm. nicht anwenden?

- das Master Theorem deckt viele D&C Algorithmen ab...
- ...aber längst nicht alle!

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

- hier wären $a = b = 2$ und $f(n) = n \cdot \log n$
- also $n^{\log_b a} = n$...
- ...und damit $f(n) = n \cdot \log n = \Omega(n^{\log_b a})$
- somit ist $f(n)$ zwar größer als $\Omega(n^{\log_b a})$, ...
- ...aber nicht polynomiell größer!

$\Longrightarrow$ können Theorem 3.7 nicht anwenden



- polynomiell Größer: um einen Faktor n^{ϵ} für ein beliebiges $\epsilon > 0$

4) Quicksort

Pseudocode zu QuickSort

Algorithmus 3.4: QuickSort(A, l, r)

```
if l < r
    p ← Partition(A, l, r)
    QuickSort(A, l, p − 1)
    QuickSort(A, p + 1, r)
```

- ein alternativer D&C-Ansatz zum Sortieren
- erstmaliger Aufruf als QuickSort$(A, 1, \text{length}(A))$
- Hilfsalgorithmus Partition wählt ein Pivot Element…
- …und nutzt es zur Aufteilung der Array Elemente

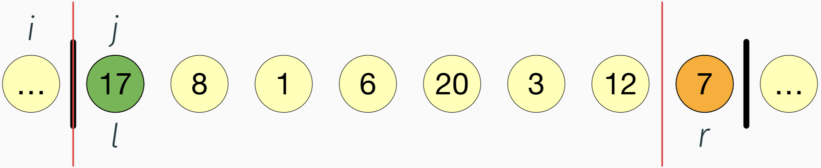
nächste Folie

Pseudocode zu Partition

Algorithmus 3.5: Partition(A, l, r)

```
x ← A[r]
i ← l − 1
for j ← l to r − 1
    if A[j] ≤ x
        i ← i + 1
        A[i] ↔ A[j]
A[i + 1] ↔ A[r]
return i + 1
```

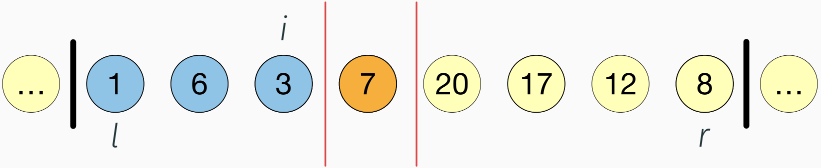
- Pivot Element $x = A[r]$
- **Ziel:** ordne $A[l \dots r]$ so um, dass
  - Elemente $\leq x$ links von x stehen
  - Element $> x$ rechts von x stehen



Algorithmus 3.5: PARTITION(A, l, r)

```
x ← A[r]
i ← l − 1
for j ← l to r − 1
    if A[j] ≤ x
        i ← i + 1
        A[i] ↔ A[j]
A[i + 1] ↔ A[r]
return i + 1
```

- Pivot Element $x = A[r]$
- **Ziel:** ordne $A[l \ldots r]$ so um, dass
  - Elemente $\leq x$ links von x stehen
  - Element $> x$ rechts von x stehen



Anmerkungen zu Quicksort

- wie MergeSort auch ein D&C Algorithmus
- Umordnung findet hier vor der Teilung statt
- es existieren viele Varianten…
- …von denen einige in der Praxis besonders effizient sind

Pivot Wahl!

Theorem 3.8

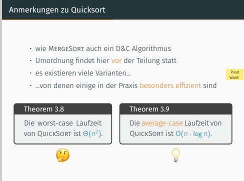
Die worst-case Laufzeit von QuickSort ist $\Theta(n^2)$.

🤔

Theorem 3.9

Die average-case Laufzeit von QuickSort ist $O(n \cdot \log n)$.

💡



- average-case LZ: durchschnittliche Laufzeit über alle Eingaben
- randomisierte Variante hat erwartete Laufzeit $\Theta(n \cdot \log n)$

Zunächst: Analyse von Partition

- sei $x = A[r]$ letztes Element von $A[l \dots r]$ vor $\textsc{Partition}(A, l, r)$
- sei $p \in \{ l, l+1, \dots, r \}$ die Ausgabe von $\textsc{Partition}(A, l, r)$

Lemma 3.5

$\textsc{Partition}(A, l, r)$ ordnet $A[l \dots r]$ so um, dass $A[p] = x$ sowie

- $A[k] \leq x$ für alle $l \leq k \leq p$ und
- $A[k] > x$ für alle $p < k \leq r$.

$\textsc{Partition}(A, l, r)$

```
x ← A[r]
i ← l − 1
for j ← l to r − 1
    if A[j] ≤ x
        i ← i + 1
        A[i] ↔ A[j]
A[i + 1] ↔ A[r]
return i + 1
```

Lemma 3.6

Es sei $n = r - l + 1$ die Größe des von Partition betrachteten Teilarrays. Partition hat Laufzeit $\Theta(n)$.

Laufzeitbeweis & Invariante

Laufzeit

Beweis von Lemma 3.6.

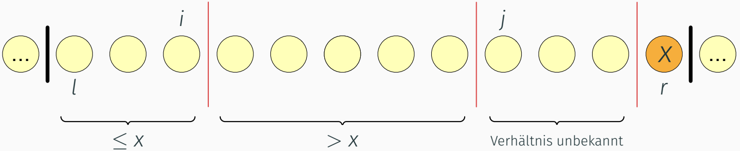
- jede Zeile für sich hat offensichtlich konstante Laufzeit
- Schleife in Zeilen 3 bis 6 wird $r - l = n - 1$ mal durchlaufen
- zusammen also $\Theta(n)$ $\square$

Geeignete Schleifeninvariante für Korrektheitsbeweis?

Schleifeninvariante $I(i, j)$

Für alle $k \in \{ l, l+1, \ldots, r \}$ gilt:

1. $l \leq k \leq i \implies A[k] \leq x$
2. $i < k < j \implies A[k] > x$
3. $k = r \implies A[k] = x$



Beweis von Lemma 3.5 (1/2)

(a) **Initialisierung:** ✓

- betrachte $I(l-1, l)$ gilt direkt vor Zeile 3
- Punkte 1 und 2 sind triviale Aussagen
- Punkt 3 gilt wegen Zeile 1

$\Longrightarrow$ $I(l-1, l)$ gilt direkt vor Zeile 3

(b) **Erhaltung:** !?

(c) **Terminierung:** ✓

- am Ende der Schleife gilt $I(i, r)$, also
  1. $l \leq k \leq i \implies A[k] \leq x$
  2. $i < k < r \implies A[k] > x$
  3. $k = r \implies A[k] = x$

$\Longrightarrow$ nach Zeile 7 gilt für Ausgabe $p = i + 1$:

1. $l \leq k \leq p \implies A[k] \leq x$
2. $p < k \leq r \implies A[k] > x$
3. $k = p \implies A[k] = x$

Schleifeninvariante $I(i, j)$

Für alle $k \in \{ l, l+1, \dots, r \}$ gilt:

1. $l \leq k \leq i \implies A[k] \leq x$
2. $i < k < j \implies A[k] > x$
3. $k = r \implies A[k] = x$

PARTITION(A, l, r)

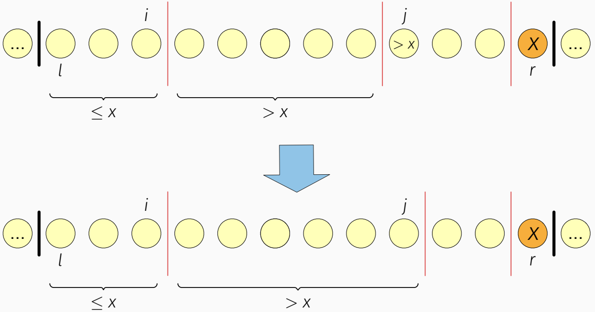
```
1  x ← A[r]
2  i ← l − 1
3  for j ← l to r − 1
4          if A[j] ≤ x
5                  i ← i + 1
6                  A[i] ↔ A[j]
7  A[i + 1] ↔ A[r]
8  return i + 1
```

Beweis von Lemma 3.5 (2/2)

Beweis der Erhaltung: $I(i, j) \to I(i, j+1)$

- gelte $I(i, j)$ am Anfang des j-Durchlaufs der Schleife
- Fall 1: $A[j] > x$
  - zweite Bedingung auch für $k = j$ wahr

$\implies$ $I(i, j+1)$ gilt



PARTITION(A, l, r)

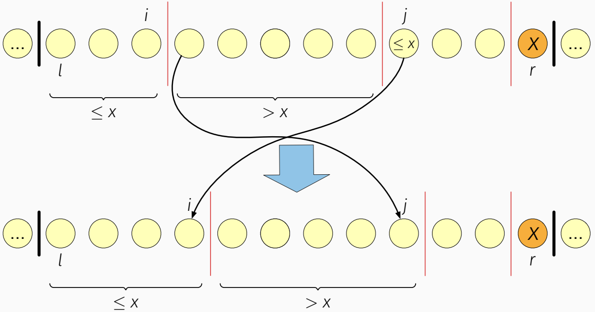
```
x ← A[r]
i ← l − 1
for j ← l to r − 1
    if A[j] ≤ x
        i ← i + 1
        A[i] ↔ A[j]
A[i + 1] ↔ A[r]
return i + 1
```

Beweis von Lemma 3.5 (2/2)

Beweis der Erhaltung: $I(i,j) \to I(i,j+1)$ ✓

- gelte $I(i,j)$ am Anfang des j-Durchlaufs der Schleife
- Fall 2: $A[j] \le x$
  - i wird auf $i+1$ gesetzt (Zeile 5)...
  - ...und (das neue) $A[i] > x$ wird mit $A[j] \le x$ vertauscht (Zeile 6)

$\Longrightarrow$ $I(i,j+1)$ gilt direkt nach Zeile 6 $\square$




```
PARTITION(A, l, r)
1  x ← A[r]
2  i ← l − 1
3  for j ← l to r − 1
4      if A[j] ≤ x
5          i ← i + 1
6          A[i] ↔ A[j]
7  A[i + 1] ↔ A[r]
8  return i + 1
```

Nun können wir Quicksort analysieren!

Theorem 3.10

QuickSort löst das Sortierproblem. Das heißt der Algorithmus sortiert eine Folge von n Zahlen aufsteigend.

QuickSort(A, l, r)

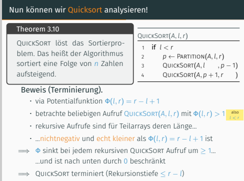
```
if l < r
    p ← Partition(A, l, r)
    QuickSort(A, l, p − 1)
    QuickSort(A, p + 1, r)
```

Beweis (Terminierung).

- via Potentialfunktion $\Phi(l, r) = r - l + 1$
- betrachte beliebigen Aufruf QuickSort(A, l, r) mit $\Phi(l, r) > 1$ also $l < r$
- rekursive Aufrufe sind für Teilarrays deren Länge...
- ...nichtnegativ und echt kleiner als $\Phi(l, r) = r - l + 1$ ist

$\Longrightarrow$ Φ sinkt bei jedem rekursiven QuickSort Aufruf um ≥ 1...
...und ist nach unten durch 0 beschränkt

$\Longrightarrow$ QuickSort terminiert (Rekursionstiefe $\leq r - l$)



- beachte: nach Lemma 3.6 terminiert auch der Aufruf von Partition!

Nun können wir Quicksort analysieren!

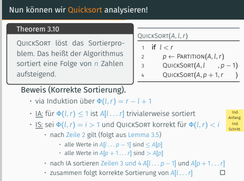
Theorem 3.10

QuickSort löst das Sortierproblem. Das heißt der Algorithmus sortiert eine Folge von n Zahlen aufsteigend.

```
QuickSort(A, l, r)
1  if l < r
2      p ← Partition(A, l, r)
3      QuickSort(A, l, p − 1)
4      QuickSort(A, p + 1, r)
```

Beweis (Korrekte Sortierung).

- via Induktion über $\Phi(l, r) = r - l + 1$
- IA: für $\Phi(l, r) \leq 1$ ist $A[l \ldots r]$ trivialerweise sortiert — Ind. Anfang
- IS: sei $\Phi(l, r) = i > 1$ und QuickSort korrekt für $\Phi(l, r) < i$ — Ind. Schritt
  - nach Zeile 2 gilt (folgt aus Lemma 3.5)
    - alle Werte in $A[l \ldots p-1]$ sind $\leq A[p]$
    - alle Werte in $A[p+1 \ldots r]$ sind $> A[p]$
  - nach IA sortieren Zeilen 3 und 4 $A[l \ldots p-1]$ und $A[p+1 \ldots r]$
  - zusammen folgt korrekte Sortierung von $A[l \ldots r]$ □



- beachte: nach Lemma 3.6 terminiert auch der Aufruf von Partition!

worst-case LZ $\Theta(n^2)$

Beweisskizze Theorem 3.8.

- Laufzeitrekursion für Laufzeit $T(n)$ von QUICKSORT:

$$T(n) = \begin{cases} \Theta(1) & , n \leq 1, \\ \max_{0 \leq q < n} \left(T(q) + T(n-q-1) \right) + \Theta(n) & , n > 1. \end{cases}$$

- rate Laufzeit $\Theta(n^2)$ und beweise Laufzeit per Induktion
- Details: DIY-Beweis □

Best-case Laufzeit

- worst-case tritt auf, wenn PARTITION nicht gut „balanciert“
- best-case bei gleichmäßiger Aufteilung liefert

$$T(n) = \begin{cases} \Theta(1) & , n \leq 1, \\ 2 \cdot T(n/2) + \Theta(n) & , n > 1. \end{cases}$$

$\Longrightarrow$ (M-Thm. (b)) $T(n) = \Theta(n \cdot \log n)$

vereinfacht

Worst-case vs Best-case Laufzeit von Quicksort

Beweisskizze Theorem 3.8.

- Laufzeitrekursion für Laufzeit $T(n)$ von QUICKSORT:

$$T(n) = \begin{cases} \Theta(1) & , n \leq 1, \\ \max_{0 \leq q < n} (T(q) + T(n-q-1)) + \Theta(n) & , n > 1. \end{cases}$$

- rate Laufzeit $\Theta(n^2)$ und beweise Laufzeit per Induktion
- Details: DIY-Beweis □

worst-case LZ $\Theta(n^2)$

Best-case Laufzeit

- worst-case tritt auf, wenn PARTITION nicht gut „balanciert“
- best-case bei gleichmäßiger Aufteilung liefert

$$T(n) = \begin{cases} \Theta(1) & , n \leq 1, \\ 2 \cdot T(n/2) + \Theta(n) & , n > 1. \end{cases}$$

$\Longrightarrow$ (M-Thm. (b)) $T(n) = \Theta(n \cdot \log n)$

verein-facht

- Der Induktionsbeweis für die worst-case LZ von QUICKSORT muss für die obere und untere Laufzeitschranke geführt werden
- Laufzeit $\Theta(n^2)$ auch für bereits sortierte Folge
- INSERTIONSORT hat in dem Fall nur lineare Laufzeit

Average-case Laufzeit von Quicksort

- Erinnerung Theorem 3.9:
  average-case Laufzeit von QUICKSORT ist $O(n \log n)$
- Was ist average-case Laufzeit?
  - betrachte alle Permutationen der n Eingabezahlen
  - berechne für jede Permutation die Laufzeit von QUICKSORT
  - average-case LZ ist Durchschnitt all dieser Laufzeiten
- Alternative Sichtweise:
  - wähle als Eingabe uniform zufällige Permutation der Länge n
  - Was ist die erwartete Laufzeit für diese Eingabe?

Laufzeit von QuickSort für zufällige Permutation (1/2)

- sei $Q_E(n)$ der erwartete LZ von QuickSort für...
- ...eine uniform zufällig gewählte Permutation der Länge n

$\Longrightarrow$ $A[n]$ ist i-kleinste Zahl mit W'keit $1/n$ $(\forall i \in \{1, 2, \ldots, n\})$

- also

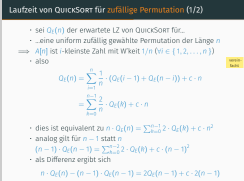
verein-
facht

$$Q_E(n) = \sum_{i=1}^{n} \frac{1}{n} \cdot (Q_E(i-1) + Q_E(n-i)) + c \cdot n$$

$$= \sum_{k=0}^{n-1} \frac{2}{n} \cdot Q_E(k) + c \cdot n$$

- dies ist equivalent zu $n \cdot Q_E(n) = \sum_{k=0}^{n-1} 2 \cdot Q_E(k) + c \cdot n^2$
- analog gilt für $n-1$ statt n
  $(n-1) \cdot Q_E(n-1) = \sum_{k=0}^{n-2} 2 \cdot Q_E(k) + c \cdot (n-1)^2$
- als Differenz ergibt sich

$$n \cdot Q_E(n) - (n-1) \cdot Q_E(n-1) = 2Q_E(n-1) + c \cdot 2(n-1)$$



- Vereinfachung: nehmen Gleichheit (statt getrennte obere/untere Schranken) im worst-case an
- Subtilität: uniforme Permutation kann unabhängig auf allen Rekursionsstufen angenommen werden

Laufzeit von QuickSort für zufällige Permutation (2/2)

- was wir umstellen können zu

$$n \cdot Q_E(n) = (n+1) \cdot Q_E(n-1) + c \cdot 2(n-1)$$

- und schließlich zu

$$\frac{Q_E(n)}{n+1} = \frac{Q_E(n-1)}{n} + 2c \cdot \frac{n-1}{n \cdot (n+1)} \leq \frac{Q_E(n-1)}{n} + \frac{2c}{n}$$

- sukzessives Einsetzen liefert

$$\frac{Q_E(n)}{n+1} \leq \frac{Q_E(n-1)}{n} + \frac{2c}{n} \leq \frac{Q_E(n-2)}{n-1} + \frac{2c}{n-1} + \frac{2c}{n}$$
$$\leq \cdots \leq \frac{Q_E(1)}{2} + 2c \cdot \sum_{i=2}^{n} \frac{1}{i} \leq \frac{Q_E(1)}{2} + 2c \cdot \ln n$$

$$\boxed{\Longrightarrow \quad Q_E(n) = O(n \log n)}$$

Was können wir daraus lernen?

- im worst-case ist QUICKSORT so schlecht wie INSERTIONSORT
  - für vorsortierte Folgen sogar schlechter
- im average-case ist QUICKSORT fast so gut wie im best-case
  - intuitiv, da für die meisten Eingaben nicht ständig...
  - ...vollständig mies partitioniert wird

Können wir grundsätzlich schlechte Eingaben vermeiden?

$\Longrightarrow$ Randomisierung!

- Gute LZ auch wenn PARTITION nicht perfekt partitioniert!
- z. B. Partitiosgrößen in $[(1/100) \cdot n, (99/100) \cdot n]$ ausreichend
- generell reicht beliebige Konstante ϵ mit Partitionsgrößen in $[\epsilon \cdot n, (1 - \epsilon) \cdot n]$ für logarithmische LZ
- selbst gelegentliche worst-case Partitionen sind ok

Wie könnte man QuickSort gut randomisieren?

Algorithmus 3.6: RndPartition(A, l, r)

```
i ← random(l, r)
A[r] ↔ A[i]
return Partition(A, l, r)
```

Algorithmus 3.7: RndQuickSort(A, l, r)

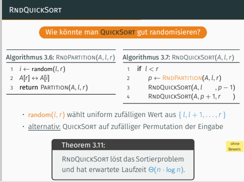
```
if l < r
    p ← RndPartition(A, l, r)
    RndQuickSort(A, l    , p − 1)
    RndQuickSort(A, p + 1, r    )
```

- random(l, r) wählt uniform zufälligen Wert aus $\{ l, l+1, \ldots, r \}$
- alternativ: QuickSort auf zufälliger Permutation der Eingabe

Theorem 3.11:

RndQuickSort löst das Sortierproblem und hat erwartete Laufzeit $\Theta(n \cdot \log n)$.

ohne Beweis



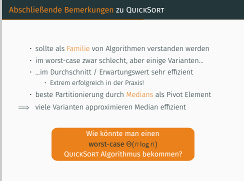
- Erwartungswert über dem Zufall aus den RndPartition Aufrufen

Abschließende Bemerkungen zu QuickSort

- sollte als Familie von Algorithmen verstanden werden
- im worst-case zwar schlecht, aber einige Varianten...
- ...im Durchschnitt / Erwartungswert sehr effizient
  - Extrem erfolgreich in der Praxis!
- beste Partitionierung durch Medians als Pivot Element

$\Longrightarrow$ viele Varianten approximieren Median effizient

Wie könnte man einen
worst-case $\Theta(n \log n)$
QuickSort Algorithmus bekommen?



- Median kann in linearer Zeit berechnet werden
- könnten also vor PARTITION immer median berechnen und als Pivot Element benutzen
- da Partition auch lineare LZ hat, ändert sich nichts an der asymptotischen LZ
- In der Praxis aber deutlich schlechter!

5) Heapsort

Motivation & Idee

- MaxSearch(A) gebe bei Eingabe eines Arrays...
- ...den Index eines maximalen Elementes zurück
- betrachte folgendes Sortierverfahren:

Algorithmus 3.8: MaxSort(A)

for $i \leftarrow$ **length**(A) downto 2
    $m \leftarrow$ MaxSearch($A[1 \dots i]$)
    $A[m] \leftrightarrow A[i]$

$\Longrightarrow$ naive Implementierung hat Laufzeit $\Theta(n^2)$

Geht das auch schneller?!

- <u>beachte:</u> mehrfache Maximum-Suche auf ähnlichen Daten!

Ziele

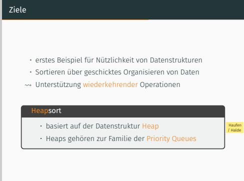
- erstes Beispiel für Nützlichkeit von Datenstrukturen
- Sortieren über geschicktes Organisieren von Daten

$\rightsquigarrow$ Unterstützung wiederkehrender Operationen

Heapsort

- basiert auf der Datenstruktur Heap
- Heaps gehören zur Familie der Priority Queues

Haufen / Halde



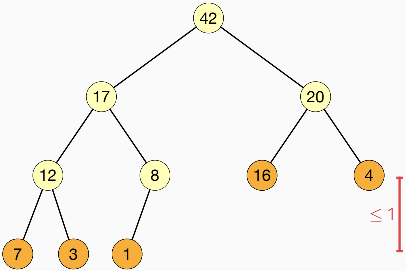
- Priority Queus: Prioritätswarteschlangen

Priority Queue

- sei U die Menge möglicher Elemente (Zahlen, Strings, ...) Universum
- sei M die Menge der aktuelle gespeicherten Elemente
- jedes $e \in U$ sei über numerischen Wert $\mathrm{key}(e)$ identifiziert

Operationen einer Priority Queue

- $\max(M)$: gib $e \in M$ mit maximalem $\mathrm{key}(e)$ aus
- $\textsc{Insert}(M, e)$: $M := M \cup \{ e \}$
- $\textsc{DeleteMax}(M)$: wie $\max(M)$, aber zusätzlich $M := M \setminus \{ e \}$



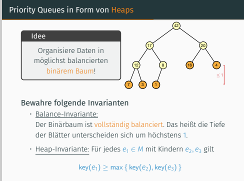
Idee

Organisiere Daten in möglichst balancierten binärem Baum!

Bewahre folgende Invarianten

- Balance-Invariante:
  Der Binärbaum ist vollständig balanciert. Das heißt die Tiefe der Blätter unterscheiden sich um höchstens 1.
- Heap-Invariante: Für jedes $e_1 \in M$ mit Kindern e_2, e_3 gilt

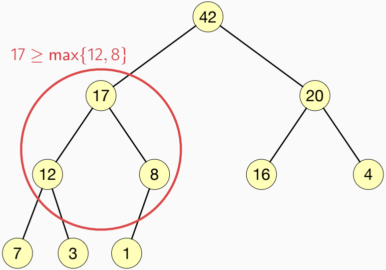
$$\mathrm{key}(e_1) \geq \max \{ \mathrm{key}(e_2), \mathrm{key}(e_3) \}$$



- Definition für max-heap; analoge Definition für min-heap

Idee

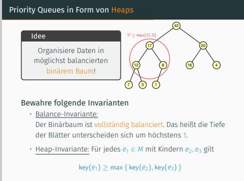
Organisiere Daten in möglichst balancierten binärem Baum!



Bewahre folgende Invarianten

- Balance-Invariante:
  Der Binärbaum ist vollständig balanciert. Das heißt die Tiefe der Blätter unterscheiden sich um höchstens 1.
- Heap-Invariante: Für jedes $e_1 \in M$ mit Kindern e_2, e_3 gilt

$$\text{key}(e_1) \geq \max \{ \text{key}(e_2), \text{key}(e_3) \}$$

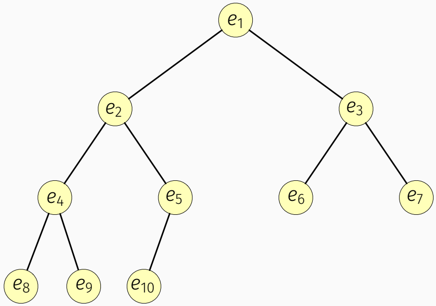


- Definition für max-heap; analoge Definition für min-heap

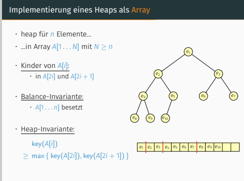
Implementierung eines Heaps als Array

- heap für n Elemente...
- ...in Array $A[1 \dots N]$ mit $N \geq n$
- Kinder von $A[i]$:
  - in $A[2i]$ und $A[2i+1]$
- Balance-Invariante:
  - $A[1 \dots n]$ besetzt
- Heap-Invariante:

$$\begin{aligned} &\mathrm{key}(A[i]) \\ \geq\ &\max\{\,\mathrm{key}(A[2i]), \mathrm{key}(A[2i+1])\,\} \end{aligned}$$



e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}		



- beachte: in der Darstellung benutzen wir oft der Einfachheit halber e sowohl für ein Element als auch für seinen key $\mathrm{key}(e)$

Heap Definitionen

Definition 3.3: Heap über Array

Ein Heap über einem Array A der Größe N ist das Array A zusammen mit einem Parameter $n := \text{heapsize}(A) \leq N$ und drei Funktionen

- $\text{Parent}(i) = \lfloor i/2 \rfloor$ für alle $i \in \{1, 2, \ldots, n\}$,
- $\text{Left}(i) = 2i$ für alle $i \in \{1, 2, \ldots, n\}$,
- $\text{Right}(i) = 2i + 1$ für alle $i \in \{1, 2, \ldots, n\}$.

Definition 3.4: max-/min-Heap

1. Ein Heap heißt max-Heap, falls für alle $i \in \{2, 3, \ldots, n\}$ $\text{key}(A[\text{Parent}(i)]) \geq \text{key}(A[i])$.
2. Ein Heap heißt min-Heap, falls für alle $i \in \{2, 3, \ldots, n\}$ $\text{key}(A[\text{Parent}(i)]) \leq \text{key}(A[i])$.

Implementierung der Heap Operationen

Zu implementieren:

- max(A): trivial („return $A[1]$“)
  - Laufzeit $\Theta(1)$
- $\textsc{Insert}(A, e)$: folgt
  - Ziel-Laufzeit $O(\log n)$
- $\textsc{DeleteMax}(A)$: folgt
  - Ziel-Laufzeit $O(\log n)$

Außerdem

- $\textsc{BuildHeap}(A)$: baue aus einem beliebiges Array A einen Heap
  - naiv: Laufzeit $O(n \log n)$
  - besser: Laufzeit $O(n)$ folgt

Implementierung der Heap Operationen

Zu implementieren:

- max(A): trivial („return $A[1]$“)
  - Laufzeit $\Theta(1)$
- Insert(A, e): folgt
  - Ziel-Laufzeit $O(\log n)$
- DeleteMax(A): folgt
  - Ziel-Laufzeit $O(\log n)$

Außerdem

- BuildHeap(A): baue aus einem beliebiges Array A einen Heap
  - naiv: Laufzeit $O(n \log n)$
  - besser: Laufzeit $O(n)$ folgt

- naive Implementierung benutzt n Insert(A, e) Operationen

Insert(A, e): Idee & Pseudocode

Algorithmus 3.9: Insert(A, e)

$n \leftarrow n + 1$

$A[n] \leftarrow e$

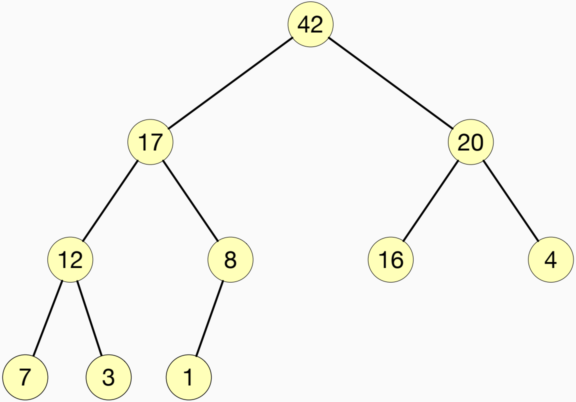
HeapifyUp(A, n)

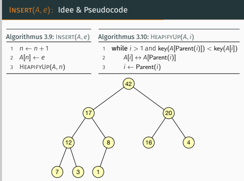
Algorithmus 3.10: HeapifyUp(A, i)

while $i > 1$ and key$(A[\text{Parent}(i)]) <$ key$(A[i])$

  $A[i] \leftrightarrow A[\text{Parent}(i)]$

  $i \leftarrow \text{Parent}(i)$





- Balance-Invariante trivialerweise erfüllt
- müssen aber beweisen, dass INSERT(A, e) die Heap-Invariante erhält

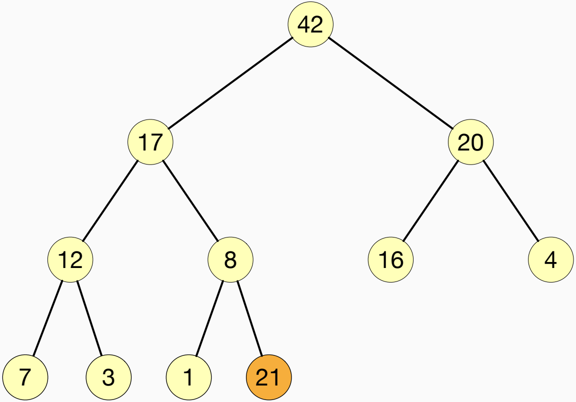
INSERT(A, e): Idee & Pseudocode

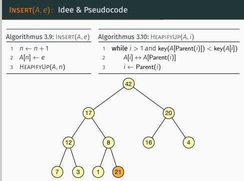
Algorithmus 3.9: INSERT(A, e)

```
n ← n + 1
A[n] ← e
HEAPIFYUP(A, n)
```

Algorithmus 3.10: HEAPIFYUP(A, i)

```
while i > 1 and key(A[Parent(i)]) < key(A[i])
    A[i] ↔ A[Parent(i)]
    i ← Parent(i)
```





- Balance-Invariante trivialerweise erfüllt
- müssen aber beweisen, dass INSERT(A, e) die Heap-Invariante erhält

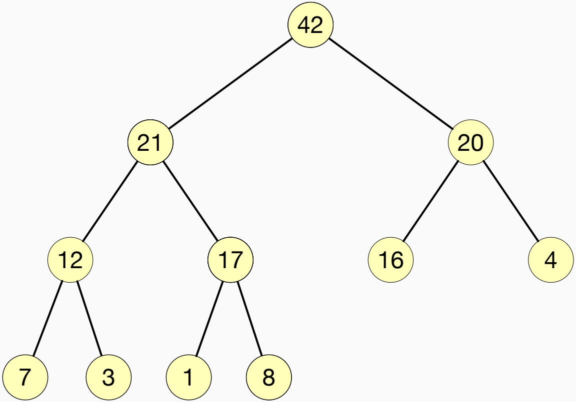
$\textsc{Insert}(A, e)$: Idee & Pseudocode

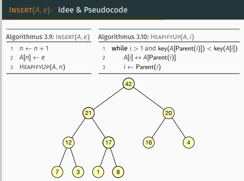
Algorithmus 3.9: $\textsc{Insert}(A, e)$

$n \leftarrow n + 1$
$A[n] \leftarrow e$
$\textsc{HeapifyUp}(A, n)$

Algorithmus 3.10: $\textsc{HeapifyUp}(A, i)$

while $i > 1$ **and** $\text{key}(A[\text{Parent}(i)]) < \text{key}(A[i])$
 $A[i] \leftrightarrow A[\text{Parent}(i)]$
 $i \leftarrow \text{Parent}(i)$





- Balance-Invariante trivialerweise erfüllt
- müssen aber beweisen, dass INSERT(A, e) die Heap-Invariante erhält

Insert(A, e): Laufzeitbeweis

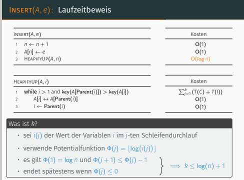
Insert(A, e)	Kosten
$n \leftarrow n + 1$	$O(1)$
$A[n] \leftarrow e$	$O(1)$
HeapifyUp(A, n)	$O(\log n)$

HeapifyUp(A, i)	Kosten
while $i > 1$ and key$(A[\text{Parent}(i)]) >$ key$(A[i])$	$\sum_{j=1}^{k} (T(C) + T(I))$
$A[i] \leftrightarrow A[\text{Parent}(i)]$	$O(1)$
$i \leftarrow \text{Parent}(i)$	$O(1)$

Was ist k?

- sei $i(j)$ der Wert der Variablen i im j-ten Schleifendurchlauf
- verwende Potentialfunktion $\Phi(j) = \lfloor \log(i(j)) \rfloor$
- es gilt $\Phi(1) = \log n$ und $\Phi(j+1) \leq \Phi(j) - 1$
- endet spätestens wenn $\Phi(j) \leq 0$

$$\left.\begin{array}{l} \Phi(1) = \log n \text{ und } \Phi(j+1) \leq \Phi(j) - 1 \\ \Phi(j) \leq 0 \end{array}\right\} \implies k \leq \log(n) + 1$$



- $\Phi(j)$ beschreibt die Tiefe des eingefügten Elements im j-ten Schleifendurchlauf
- $\Phi(1) = 0$ ist offensichtlich
- $\Phi(j)$ sinkt in jedem Durchlauf, da das eingefügte Element ein level nach oben wandert
- bei $\Phi(j) = 0$ hat das eingefügte Element die Wurzel erreicht

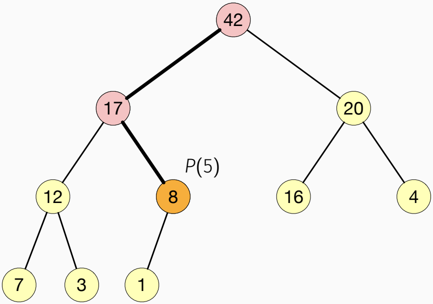
Definiere

- $P(i) = \{ \lfloor i/2^j \rfloor \mid j \in \{ 1, 2, \ldots, \lfloor \log(i) \rfloor \} \}$ parents
- $T(i) = \{ i \} \cup \{ j \in \{ 1, 2, \ldots, n \} \mid i \in P(j) \}$ subtree

Für Analyse: Nehmen o. B. d. A. an, dass $\mathrm{key}(e) = e$

HEAPIFYUP(A, i)

```
while i > 1 and A[Parent(i)] > A[i]
    A[i] ↔ A[Parent(i)]
    i ← Parent(i)
```

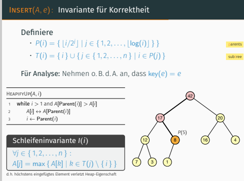


Schleifeninvariante $I(i)$

$\forall j \in \{ 1, 2, \ldots, n \}:$

$A[j] = \max \{ A[k] \mid k \in T(j) \setminus \{ i \} \}$

d. h. höchstens eingefügtes Element verletzt Heap-Eigenschaft



- funktioniert, da der Inhalt der Elemente für den Algorithmus keine Rolle spielt

INSERT(A, e): Invariante für Korrektheit

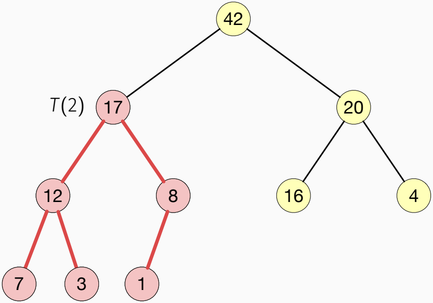
Definiere

- $P(i) = \{ \lfloor i/2^j \rfloor \mid j \in \{ 1, 2, \ldots, \lfloor \log(i) \rfloor \} \}$ parents
- $T(i) = \{ i \} \cup \{ j \in \{ 1, 2, \ldots, n \} \mid i \in P(j) \}$ subtree

Für Analyse: Nehmen o. B. d. A. an, dass $\mathrm{key}(e) = e$

HEAPIFYUP(A, i)

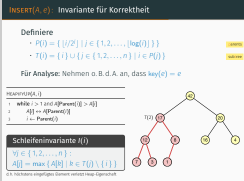
```
while i > 1 and A[Parent(i)] > A[i]
    A[i] ↔ A[Parent(i)]
    i ← Parent(i)
```



Schleifeninvariante $I(i)$

$\forall j \in \{ 1, 2, \ldots, n \}$:
$A[j] = \max \{ A[k] \mid k \in T(j) \setminus \{ i \} \}$

d. h. höchstens eingefügtes Element verletzt Heap-Eigenschaft



- funktioniert, da der Inhalt der Elemente für den Algorithmus keine Rolle spielt

INSERT(A, e): Korrektheitsbeweis (1/3)

$\forall j \in \{1, 2, \ldots, n\}$:
$A[j] = \max\{A[k] \mid k \in T(j) \setminus \{i\}\}$

```
INSERT(A, e)
n ← n + 1
A[n] ← e
HEAPIFYUP(A, n)
```

```
HEAPIFYUP(A, i)
while i > 1 and A[Parent(i)] < A[i]
    A[i] ↔ A[Parent(i)]
    i ← Parent(i)
```

(a) **Initialisierung:** ✓

- vor HEAPIFYUP wurden keine Heap-Elemente verändert...
- ...sondern nur ein neues Blatt eingefügt

$\Longrightarrow$ $I(n)$ gilt trivialerweise

$\Longrightarrow$ $I(i)$ gilt direkt vor der while-Schleife (da mit $i = n$ aufgerufen)

INSERT(A, e): Korrektheitsbeweis (2/3)

$\forall j \in \{1, 2, \dots, n\}$:
$A[j] = \max\{A[k] \mid k \in T(j) \setminus \{i\}\}$

```
INSERT(A, e)
n ← n + 1
A[n] ← e
HEAPIFYUP(A, n)
```

```
HEAPIFYUP(A, i)
while i > 1 and A[Parent(i)] < A[i]
    A[i] ↔ A[Parent(i)]
    i ← Parent(i)
```

(b) **Erhaltung:** ✓

- gelte $I(i)$ zu Beginn eines while-Schleifendurchlaufs

$\Longrightarrow$ (while-Bedingung + Invariante)

$$A[\text{Parent}(i)] < A[i]$$
$$\wedge \quad A[\text{Parent}(i)] \geq \max\{A[k] \mid k \in T(\text{Parent}(i)) \setminus \{i\}\}$$

$\Longrightarrow$ nach Vertauschung von $A[i]$ und $A[\text{Parent}(i)]$ gelten

$$A[i] \geq \max\{A[k] \mid k \in T(\text{Parent}(i)) \setminus \{\text{Parent}(i)\}\}$$
$$\wedge \quad A[\text{Parent}(i)] > A[i] \geq \max\{A[k] \mid k \in T(\text{Parent}(i)) \setminus \{\text{Parent}(i)\}\}$$

- nach Anweisung „$i \leftarrow \text{Parent}(i)$“ gilt wieder die Invariante $I(i)$

$\forall j \in \{1, 2, \ldots, n\}:$
$A[j] = \max\{A[k] \mid k \in T(j) \setminus \{i\}\}$

```
INSERT(A, e)
n ← n + 1
A[n] ← e
HEAPIFYUP(A, n)
```

```
HEAPIFYUP(A, i)
while i > 1 and A[Parent(i)] < A[i]
    A[i] ↔ A[Parent(i)]
    i ← Parent(i)
```

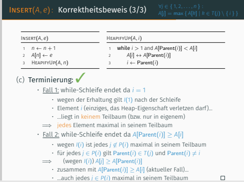
(c) **Terminierung:** ✓

- Fall 1: while-Schleife endet da $i = 1$
  - wegen der Erhaltung gilt $I(1)$ nach der Schleife
  - Element i (einziges, das Heap-Eigenschaft verletzen darf)...
  - ...liegt in keinem Teilbaum (bzw. nur in eigenem)

  $\Longrightarrow$ jedes Element maximal in seinem Teilbaum
- Fall 2: while-Schleife endet da $A[\text{Parent}(i)] \geq A[i]$
  - wegen $I(i)$ ist jedes $j \notin P(i)$ maximal in seinem Teilbaum
  - für jedes $j \in P(i)$ gilt $\text{Parent}(i) \in T(j)$ und $\text{Parent}(i) \neq i$

  $\Longrightarrow$ (wegen $I(i)$) $A[j] \geq A[\text{Parent}(i)]$
  - zusammen mit $A[\text{Parent}(i)] \geq A[i]$ (aktueller Fall)...
  - ...auch jedes $j \in P(i)$ maximal in seinem Teilbaum □

INSERT(A, e): Korrektheitsbeweis (3/3)



INSERT(A, e): Korrektheitsbeweis (3/3)

$\forall j \in \{1, 2, \ldots, n\}:$
$A[j] = \max\{A[k] \mid k \in T(j) \setminus \{i\}\}$

```
INSERT(A, e)
n ← n + 1
A[n] ← e
HEAPIFYUP(A, n)
```

```
HEAPIFYUP(A, i)
while i > 1 and A[Parent(i)] < A[i]
    A[i] ↔ A[Parent(i)]
    i ← Parent(i)
```

(c) Terminierung: ✓

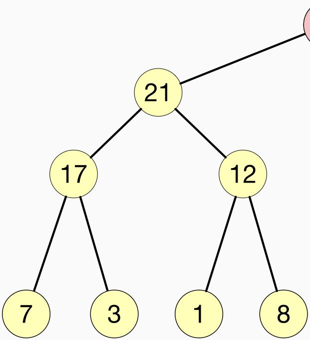
- Fall 1: while-Schleife endet da $i = 1$
  - wegen der Erhaltung gilt $I(1)$ nach der Schleife
  - Element i (einziges, das Heap-Eigenschaft verletzen darf)...
  - ...liegt in keinem Teilbaum (bzw. nur in eigenem)

  $\Longrightarrow$ jedes Element maximal in seinem Teilbaum
- Fall 2: while-Schleife endet da $A[\text{Parent}(i)] \geq A[i]$
  - wegen $I(i)$ ist jedes $j \notin P(i)$ maximal in seinem Teilbaum
  - für jedes $j \in P(i)$ gilt $\text{Parent}(i) \in T(j)$ und $\text{Parent}(i) \neq i$

  $\Longrightarrow$ (wegen $I(i)$) $A[j] \geq A[\text{Parent}(i)]$
  - zusammen mit $A[\text{Parent}(i)] \geq A[i]$ (aktueller Fall)...
  - ...auch jedes $j \in P(i)$ maximal in seinem Teilbaum □

- wenn ein Element maximal in seinem Teilbaum ist, ist es natürlich insbesondere mindestens so groß wie seine Kinder, so dass die Heap-Eigenschaft gilt

DeleteMax(A): Idee & Pseudocode

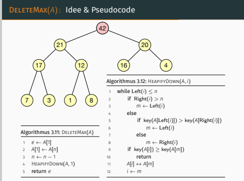


Algorithmus 3.11: DeleteMax(A)

```
e ← A[1]
A[1] ← A[n]
n ← n − 1
HeapifyDown(A, 1)
return e
```

Algorithmus 3.12: HeapifyDown(A, i)

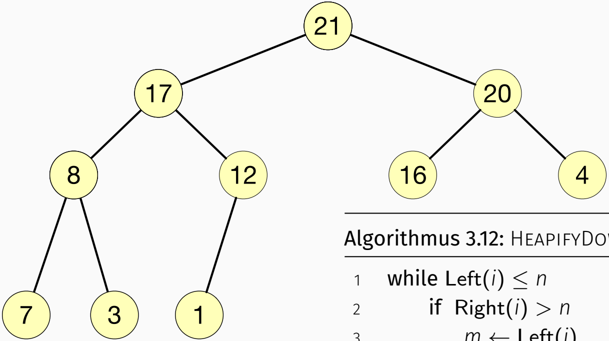
```
while Left(i) ≤ n
    if Right(i) > n
        m ← Left(i)
    else
        if key(A[Left(i)]) > key(A[Right(i)])
            m ← Left(i)
        else
            m ← Right(i)
    if key(A[i]) ≥ key(A[m])
        return
    A[i] ↔ A[m]
    i ← m
```



- HeapifyDown vergleicht Element e das (vlt.) Heap-Eigenschaft verletzt mit seinen Kindern
- ist das Element größer als seine Kinder, so ist alles in Ordnung
- andernfalls tausche mit größerem der beiden Kinder

$\Longrightarrow$ Heap-Eigenschaft wieder höchstens durch e verletzt, nun aber eine Ebene tiefer

DeleteMax(A): Idee & Pseudocode

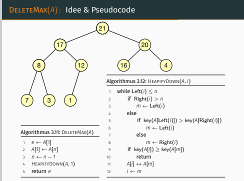


Algorithmus 3.12: HeapifyDown(A, i)

```
while Left(i) ≤ n
    if Right(i) > n
        m ← Left(i)
    else
        if key(A[Left(i)]) > key(A[Right(i)])
            m ← Left(i)
        else
            m ← Right(i)
    if key(A[i]) ≥ key(A[m])
        return
    A[i] ↔ A[m]
    i ← m
```

Algorithmus 3.11: DeleteMax(A)

```
e ← A[1]
A[1] ← A[n]
n ← n − 1
HeapifyDown(A, 1)
return e
```



- HeapifyDown vergleicht Element e das (vlt.) Heap-Eigenschaft verletzt mit seinen Kindern
- ist das Element größer als seine Kinder, so ist alles in Ordnung
- andernfalls tausche mit größerem der beiden Kinder

$\Longrightarrow$ Heap-Eigenschaft wieder höchstens durch e verletzt, nun aber eine Ebene tiefer

DeleteMax(A): Laufzeitbeweis

- Laufzeit $O(\log n)$...
- ...lässt sich analog zur Laufzeit von Insert zeigen
- nutze wieder Potentialfunktion
- jede Iteration verringert betrachtetes Level im Baum

DIY-Beweis

DeleteMax(A): Invariante für Korrektheit

Definiere

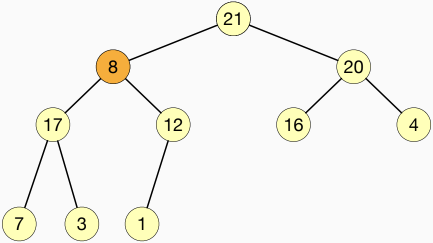
- $T(i) = \{ i \} \cup \{ j \in \{ 1, 2, \ldots, n \} \mid i \in P(j) \}$

subtree

Für Analyse: Nehmen o. B. d. A. an, dass $\text{key}(e) = e$

HeapifyDown(A, i)

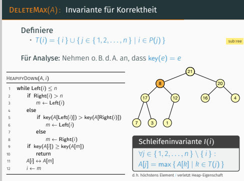
```
while Left(i) ≤ n
    if Right(i) > n
        m ← Left(i)
    else
        if key(A[Left(i)]) > key(A[Right(i)])
            m ← Left(i)
        else
            m ← Right(i)
    if key(A[i]) ≥ key(A[m])
        return
    A[i] ↔ A[m]
    i ← m
```



Schleifeninvariante $I(i)$

$\forall j \in \{ 1, 2, \ldots, n \} \setminus \{ i \} :$
$A[j] = \max \{ A[k] \mid k \in T(j) \}$

d. h. höchstens Element i verletzt Heap-Eigenschaft



- funktioniert, da der Inhalt der Elemente für den Algorithmus keine Rolle spielt

(a) **Initialisierung:** ✓

$\Longrightarrow$ $I(1)$ gilt trivialerweise zu Beginn von HeapifyDown

$\Longrightarrow$ $I(i)$ gilt direkt vor der while-Schleife (da mit $i = 1$ aufgerufen)

(b) **Erhaltung:** ✓

- o. B. d. A. sei $A[\mathrm{Left}(i)] = \max\{A[\mathrm{Left}(i)], A[\mathrm{Right}(i)]\}$

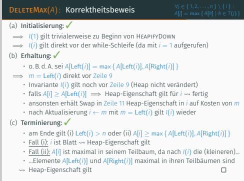
$\Longrightarrow$ $m = \mathrm{Left}(i)$ direkt vor Zeile 9

- Invariante $I(i)$ gilt noch vor Zeile 9 (Heap nicht verändert)
- falls $A[i] \geq A[\mathrm{Left}(i)]$ $\Longrightarrow$ Heap-Eigenschaft gilt für i $\rightsquigarrow$ fertig
- ansonsten erhält Swap in Zeile 11 Heap-Eigenschaft in i auf Kosten von m
- nach Aktualisierung $i \leftarrow m$ mit $m = \mathrm{Left}(i)$ gilt $I(i)$ wieder

(c) **Terminierung:** ✓

- am Ende gilt (i) $\mathrm{Left}(i) > n$ oder (ii) $A[i] \geq \max\{A[\mathrm{Left}(i)], A[\mathrm{Right}(i)]\}$
- <u>Fall (i):</u> i ist Blatt $\rightsquigarrow$ Heap-Eigenschaft gilt
- <u>Fall (ii):</u> $A[i]$ ist maximal in seinem Teilbaum, da nach $I(i)$ die (kleineren)...
- ...Elemente $A[\mathrm{Left}(i)]$ und $A[\mathrm{Right}(i)]$ maximal in ihren Teilbäumen sind

$\rightsquigarrow$ Heap-Eigenschaft gilt $\square$



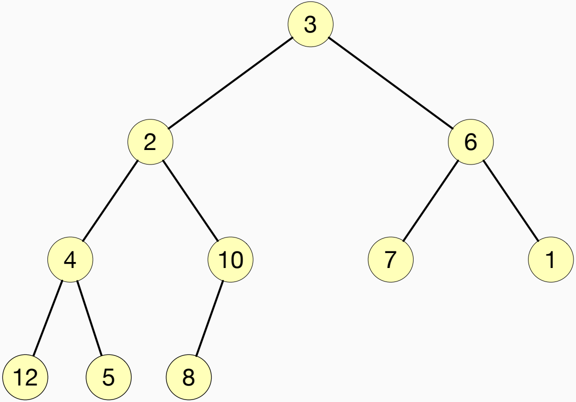
- $I(1)$ gilt, da alle Teilbäume $T(j)$ mit $j \neq 1$ höchstens ein Blatt weniger haben als vor Aufruf von DeleteMax
- falls $\text{Right}(i) > n$ denken wir uns ein Dummy-Element $A[\text{Right}(i)] = \infty$
- Swap erhält Heap Eigenschaft, da $A[\text{Left}(i)] = \max\{A[\text{Left}(i)], A[\text{Right}(i)]\}$ nach unserer o. B. d. A.-Annahme

BuildHeap(A): Idee & Pseudocode

- jedes Blatt ist ein gültiger Heap
- konstruiere Heap levelweise…
- …„von unten nach oben“

Algorithmus 3.13: BuildHeap(A)

$n \leftarrow$ heapsize(A)
for $i \leftarrow \lfloor n/2 \rfloor$ downto 1
    HeapifyDown(A, i)

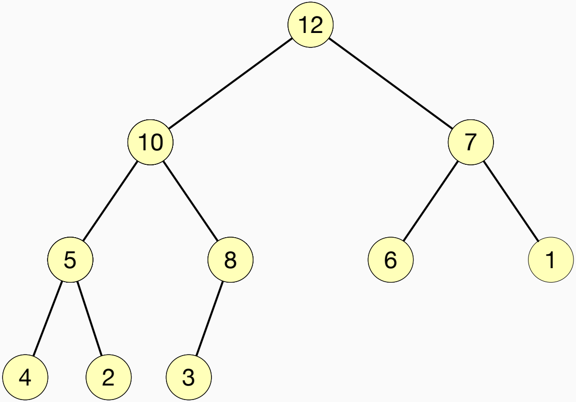


BuildHeap(A): Idee & Pseudocode

- jedes Blatt ist ein gültiger Heap
- konstruiere Heap levelweise…
- …„von unten nach oben“

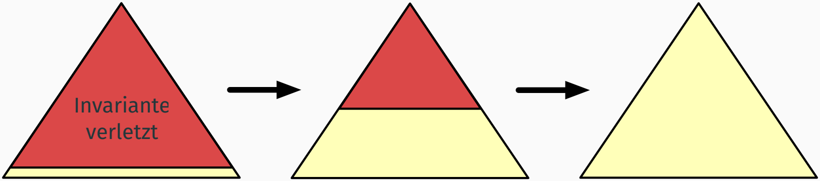
Algorithmus 3.13: BuildHeap(A)

$n \leftarrow$ heapsize(A)
for $i \leftarrow \lfloor n/2 \rfloor$ downto 1
    HeapifyDown(A, i)



BuildHeap(A): Idee der Analyse

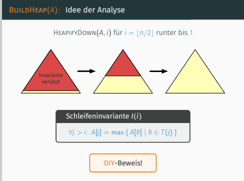
HeapifyDown(A, i) für $i = \lfloor n/2 \rfloor$ runter bis 1



Schleifeninvariante $I(i)$

$$\forall j > i\colon A[j] = \max \{ A[k] \mid k \in T(j) \}$$

DIY-Beweis!



- triviale Analyse gibt Laufzeit $O(n \cdot \log n)$ (pro Knoten gehen wir höchstens $\log n$ Level runter)
- man erhält lineare Laufzeit, wenn man genauer rechnet
  - Nur wenige Knoten mit großer Höhe!
  - genauer: Anzahl Knoten der Höhe h ist $\leq \lceil n/2^{h+1} \rceil$

😩... und nun?

🎉 Heapsort! 🎉

Algorithmus 3.14: $\textsc{HeapSort}(A)$

$\textsc{BuildHeap}(A)$
for $i \leftarrow \text{length}(A)$ **downto** 2
  $A[i] \leftarrow \textsc{DeleteMax}(A)$

Theorem 3.12

$\textsc{HeapSort}$ löst das Sortierproblem und hat Laufzeit $O(n \cdot \log n)$.

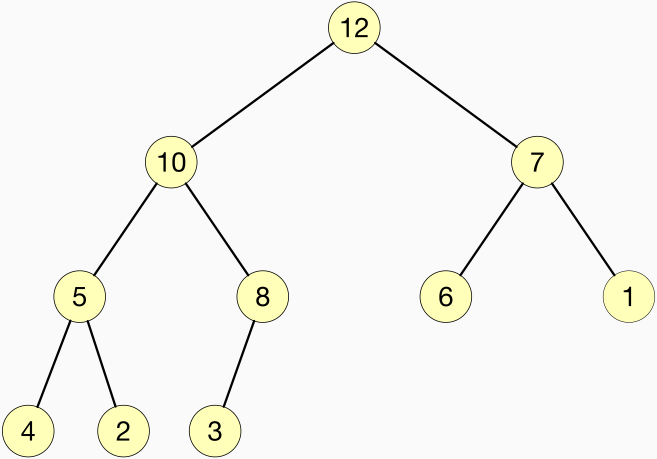
Skizze Korrektheit

- Korrektheit von $\textsc{BuildHeap}(A)$
- Korrektheit von $\textsc{DeleteMax}(A)$
- Schleifeninvariante:
  „$A[i+1 \ldots \text{length}(A)]$ enthält maximale Eingabezahlen von A aufsteigend sortiert“

Skizze Laufzeit

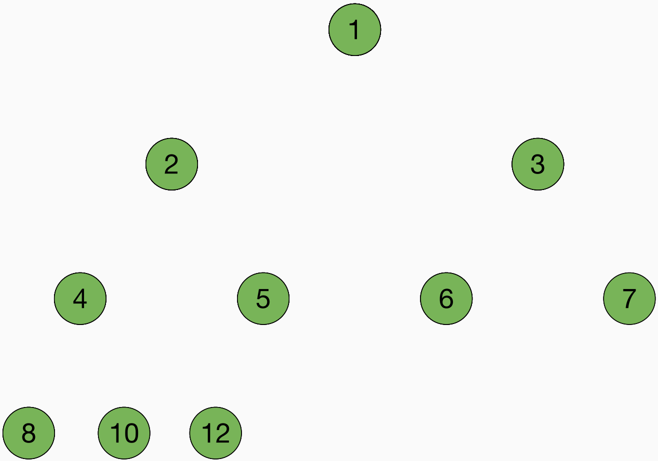
- Aufruf von $\textsc{BuildHeap}$: $O(n)$
- n Durchläufe der for-Schleife
- pro Durchlauf Laufzeit $O(\log n)$ ($\textsc{DeleteMax}$)

Illustration von HeapSort



12	10	7	5	8	6	1	4	2	3

Illustration von HEAPSORT



1	2	3	4	5	6	7	8	10	12

6) Untere Schranke für Vergleichssortierer

Übersicht zu Laufzeiten von Sortieralgorithmen

		Laufzeit		
		best-case	average-case	worst-case
Algorithmus	Insertionsort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
	Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
	Quicksort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$
	Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$

- alle bisherigen Algorithmen haben worst-case LZ $\Omega(n \log n)$
- Ist das Zufall oder gibt es hierfür einen Grund?

🎯 **Ziel**

- fasse gemeinsame Eigenschaften durch Modell der…
- …Vergleichssortierer zusammen und beweis, dass jeder…
- …Vergleichssortierer worst-case LZ $\Omega(n \log n)$ besitzt

Sortieren über Vergleiche

Definition 3.5

Ein Vergleichssortierer ist ein Algorithmus, der das Sortierproblem löst und dazu nur die Operationen $=$, $\neq$, $\leq$, $\geq$, $<$ und $>$ auf je zwei Eingabezahlen a_i und a_j benutzt, um Informationen über die Eingabe zu gewinnen.

- also sonst nur Kontrolloperationen + Zuweisungen
- zur Vereinfachung nehmen wir $a_i \neq a_j$ für alle $i \neq j$ an

$\Longrightarrow$ benötigen kein $=$ oder $\neq$

- erhalten durch $\leq$, $\geq$, $<$ und $>$ die gleiche Information

$\Longrightarrow$ o. B. d. A. betrachten wir nur $\leq$

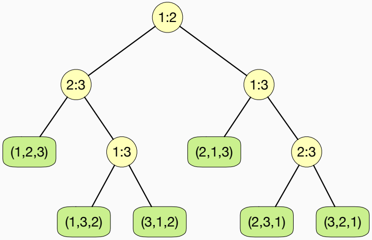
Baumstruktur der Vergleichsoperationen

Definition 3.6

Ein Entscheidungsbaum über n Zahlen ist ein binärer Baum dessen Knoten wie folgt gelabelt sind:

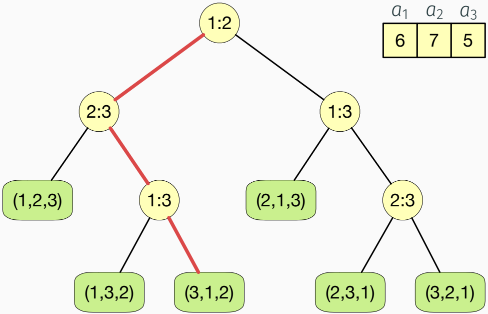
- innere Knoten: Label ist $i : j$ mit $i, j \in \{1, 2, \ldots, n\}$
- Blatt: Label ist Permutation $\pi\colon \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$

- spiegelt mögliche Ausführungen eines Vergleichssortierers wider



- Pfad von Wurzel zu Blatt $\equiv$ Ausführung
- Knoten mit Label „$i : j$“: $a_i \leq a_j$?
  - ja $\Longrightarrow$ gehe zu linkem Kind
  - nein $\Longrightarrow$ gehe zu rechtem Kind
- Blatt mit Label „π“: Permutation π sortiert die Eingabe

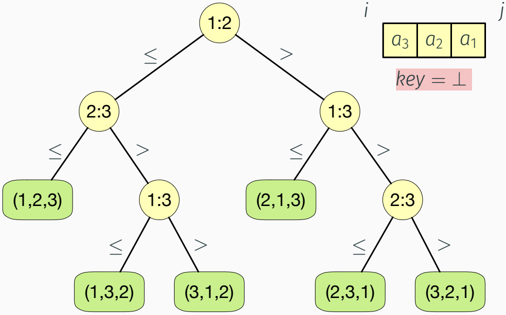
Beispiel: Entscheidungsbaum & Pfad für Insertionsort



InsertionSort(A)

```
for j ← 2 to length(A)
    key ← A[j]
    i ← j − 1
    while i > 0 and A[i] > key
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← key
```

Beispiel: Konstruktion Entscheidungsbaum für Insertionsort



$\textsc{InsertionSort}(A)$

```
for j ← 2 to length(A)
    key ← A[j]
    i ← j − 1
    while i > 0 and A[i] > key
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← key
```

$\Longrightarrow$ erhalten so Ent.-Baum für jeden Vergleichssortierer

Lemma 3.7

Für Eingaben der Größe n hat ein Entscheidungsbaum eines Vergleichssortierers mindestens $n!$ Blätter.

Beweis.

- Annahme: eine der $n!$ möglichen Permutationen nicht im Ent.-Baum sei dies π
- wähle Eingabe $(a_1, a_2, \ldots, a_n)$ mit $a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}$
- zugrundeliegender Vergleichssortierer löst Sortierproblem
  $\Longrightarrow$ Ent.-Baum liefert Permutation $\pi' \neq \pi$
  mit $a_{\pi'(1)} < a_{\pi'(2)} < \cdots < a_{\pi'(n)}$
- haben zwei unterschiedliche sortierte Reihenfolgen gefunden ⚡
  (Sortierung eindeutig, da alle keine zwei Eingabewerte identisch) □

Theorem 3.13

Jeder Vergleichssortierer benötigt für Eingabegröße n im worst-case $\Omega(n \log n)$ Vergleiche.

Beweis.

- betrachte den zugehörigen Entscheidungsbaum
- $\overset{\text{Lemma 3.7}}{\Longrightarrow}$ Entscheidungsbaum hat $\geq n!$ Blätter
- $\Longrightarrow$ Entscheidungsbaum hat Höhe $\geq \log(n!)$
  - Anzahl der Elemente in Baum auf Ebene i ist $\leq 2^i$
  - angenommen der Baum hätte Höhe $\leq \log(n!) - 1$...
    ...so hätte er $\leq 2^{\log(n!)-1} = n!/2 < n!$ viele Blätter ⚡
- nach Konstruktion des Ent.-Baums gibt es also eine Eingabe...
  ...die $\geq \log(n!)$ Vergleiche benötigt
- Theorem folgt, da $\log(n!) = \Theta(n \log n)$

Hilfslemma für Beweis von Theorem 3.13

Lemma 3.8

Es gilt $\log(n!) = \Theta(n \log n)$.

Beweis.

- $n! = 1 \cdot 2 \cdot \dots \cdot n > (n/2 + 1) \cdot \dots \cdot n > (n/2)^{n/2}$
- $n! = 1 \cdot 2 \cdot \dots \cdot n < n^n$
- mit diesen Abschätzungen können wir wie folgt rechnen:

$$\begin{aligned} & \log\left((n/2)^{n/2}\right) < \log(n!) < \log(n^n) \\ \iff \quad & (n/2) \cdot \log(n/2) < \log(n!) < n \cdot \log(n) \end{aligned}$$

- die Aussage folgt, da

$$(n/2) \cdot \log(n/2) = (n/2) \cdot \left(\log(n) - 1\right) > (n/4) \cdot \log n$$

für alle $n > 4$ $\square$

Takeaway

- Mergesort & Heapsort sind asympt. optimale Vergleichssortierer
- Grundidee der unteren Laufzeitschranke
  - um korrekt zu sortieren muss ein Vergleichssortierer…
    …alle $n!$ mögliche Reihenfolgen ausgeben können
  - das geht nur mit $\log(n!)$ vielen Vergleichsoperationen
    (da Entscheidungsbaum sonst nicht alle $n!$ Reihenfolgen abdeckt)
  - $\log(n!) = \Theta(n \log n)$

Gibt es also keine Möglichkeit schneller als $\Theta(n \log n)$ zu sortieren?

7) Sortieren in linearer Zeit

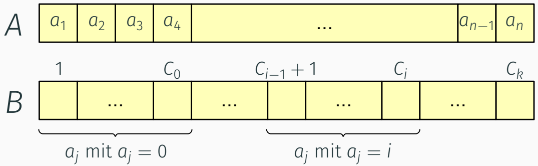
Countingsort – Sortieren durch Abzählen

Annahme

- gegeben eine Folge $(a_1, a_2, \dots, a_n)$
- es gelte $a_i \in \{0, 1, \dots, k\}$ für alle $i \in \{1, 2, \dots, n\}$

Idee

- bestimmte $\forall i \in \{0, 1, \dots, k\}$ die Anzahl C_i der a_j mit $a_j \leq i$
- es gibt also genau $C_i - C_{i-1}$ Elemente a_j mit $a_j = i$
  - dabei sei $C_{-1} = 0$
- erstelle ein leeres Array B der Länge n
- kopiere die $C_i - C_{i-1}$ Elemente mit $a_j = i$ nach $B[C_{i-1} + 1 \dots C_i]$



Pseudocode von CountingSort

- Zeilen 1 bis 2: erstelle & initialisiere Zählarray C
- Zeilen 3 bis 4: zähle in $C[i]$ wie viele Elemente in A Wert $=$ i haben
- Zeilen 5 bis 6: zähle in $C[i]$ wie viele Elemente in A Werte $\leq$ i haben
- Zeilen 7 bis 9: erstelle sortierte Permutation B von A

Algorithmus 3.15: CountingSort(A, B, k)

```
1  for i ← 0 to k
2      C[i] ← 0
3  for j ← 1 to length(A)
4      C[A[j]] ← C[A[j]] + 1
5  for i ← 1 to k
6      C[i] ← C[i] + C[i − 1]
7  for j ← length(A) downto 1
8      B[C[A[j]]] ← A[j]
9      C[A[j]] ← C[A[j]] − 1
```

Illustration von CountingSort

A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C

0	1	2	3	4	5
2	0	2	3	0	1

nach Zeile 4

B

1	2	3	4	5	6	7	8
0	0	2	2	3	3	3	5

C

0	1	2	3	4	5
2	2	4	7	7	8

nach Zeile 6

C

0	1	2	3	4	5
0	2	2	4	7	7

nach Zeile 9

Analyse von CountingSort

- n sei die Länge des Eingabearrays A
- alle Einträge seien aus $\{0, 1, \ldots, k\}$

Theorem 3.14

CountingSort löst das Sortierproblem und hat Laufzeit $O(n + k)$.

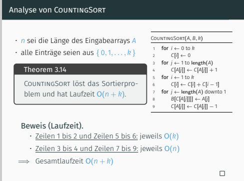
```
CountingSort(A, B, k)
1  for i ← 0 to k
2      C[i] ← 0
3  for j ← 1 to length(A)
4      C[A[j]] ← C[A[j]] + 1
5  for i ← 1 to k
6      C[i] ← C[i] + C[i − 1]
7  for j ← length(A) downto 1
8      B[C[A[j]]] ← A[j]
9      C[A[j]] ← C[A[j]] − 1
```

Beweis (Laufzeit).

- Zeilen 1 bis 2 und Zeilen 5 bis 6: jeweils $O(k)$
- Zeilen 3 bis 4 und Zeilen 7 bis 9: jeweils $O(n)$

$\Longrightarrow$ Gesamtlaufzeit $O(n + k)$

□



- für $k = O(n)$ hat CountingSort lineare Laufzeit

Skizze zum Korrektheitsbeweis von CountingSort (1/2)

- Korrektheit von vier Schleifen
- Schleife 1: (Zeilen 1 bis 2)
  - initialisiert C mit 0
- Schleife 2: (Zeilen 3 bis 4)
  - benutze Invariante $I_1(j)$:

    $$\forall i \in \{0, 1, \ldots, k\}: \quad C[i] = |\{\, l \in \{1, 2, \ldots, j-1\} \mid A[l] = i \,\}|$$

```
CountingSort(A, B, k)
1  for i ← 0 to k
2      C[i] ← 0
3  for j ← 1 to length(A)
4      C[A[j]] ← C[A[j]] + 1
5  for i ← 1 to k
6      C[i] ← C[i] + C[i − 1]
7  for j ← length(A) downto 1
8      B[C[A[j]]] ← A[j]
9      C[A[j]] ← C[A[j]] − 1
```

- Schleife 3: (Zeilen 5 bis 6)
  - benutze Invariante $I_2(i)$:

    $$\forall j \in \{0, 1, \ldots, i-1\}: C[j] = |\{\, l \in \{1, 2, \ldots, n\} \mid A[l] \leq j \,\}|$$

    $$\forall j \in \{i, i+1, \ldots, k\}: C[j] = |\{\, l \in \{1, 2, \ldots, n\} \mid A[l] = j \,\}|$$

Skizze zum Korrektheitsbeweis von Counting Sort (2/2)

- Schleife 4: (Zeilen 7 bis 9)
  - benutze Invariante $I_3(j)$:

$$\forall i \in \{ j+1, j+2, \ldots, n \} : \text{ Pos. von } A[i] \text{ in } B \text{ ist}$$
$$|\{ l \in \{1,2,\ldots,n\} \mid A[l] < A[i] \}|$$
$$+\, |\{ l \in \{1,2,\ldots,i-1\} \mid A[l] = A[i] \}|$$
$$+\, 1$$

$$\forall i \in \{0,1,\ldots,k\} : C[i] = |\{ l \in \{1,2,\ldots,n\} \mid A[l] < i \}| + |\{ l \in \{1,2,\ldots,j\} \mid A[l] = i \}|$$

$\square$

Beobachtung: Invariante beinhaltet Stabilität

CountingSort erhält die relative Ordnung gleicher Elemente aus dem Eingabearray.

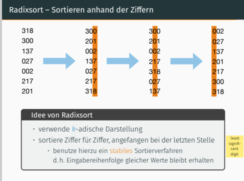
Radixsort – Sortieren anhand der Ziffern

318	300	300	002
300	201	201	027
137	002	002	137
027	137	217	201
002	027	318	217
217	217	027	300
201	318	137	318

Idee von Radixsort

- verwende k-adische Darstellung
- sortiere Ziffer für Ziffer, angefangen bei der letzten Stelle
  - benutze hierzu ein stabiles Sortierverfahren
    d. h. Eingabereihenfolge gleicher Werte bleibt erhalten

least signifi- cant digit



- der Begriff Radix bezeichnet die Anzahl an Ziffern in einer gegebenen Zahlendarstellung

Annahmen

- alle Zahlen in A bestehen aus d Ziffern
- jede Ziffer nimmt einen von k Werten an
- STABLESORT(A, i) sei ein Algorithmus der das Array A...
  ...Aufsteigend nach der i-ten Ziffer (von rechts) sortiert

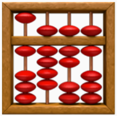
Algorithmus 3.16: RADIXSORT(A, k)

```
for i ← 1 to d
    STABLESORT(A, i)
```

Laufzeit

- (leicht angepasstes) COUNTINGSORT für STABLESORT...
- ...liefert Laufzeit $O(d \cdot (n + k))$

Beweisskizze zur Korrektheit von RadixSort

 $\rightsquigarrow$ 

- betrachte beliebiges Paar $a < b$ aus Eingabearray A
- dann existiert ein i mit $a_i < b_i$ und $a_j = b_j$ für alle $j > i$
  - hier bezeichnet x_i die i-te Ziffer der Zahl x (von rechts)
- nach i-ten Schleifendurchlauf von RadixSort:
  - $\text{pos}(a) < \text{pos}(b)$
  - (soll heißen, a liegt an einer Position vor b in Array A)
- nach j-ten Schleifendurchlauf von RadixSort für alle $j > i$:
  - Stabilität garantiert, dass $\text{pos}(a) < \text{pos}(b)$ erhalten bleibt
- gilt für alle Zahlenpaare $\Longrightarrow$ Folge ist am Ende sortiert $\square$

Fragen?

😴