Algorithmen und Datenstrukturen

Kapitel 3: Sortieren

Prof. Dr. Peter Kling

Wintersemester 2020/21

Übersicht

1. Insertion Sort
2. Merge Sort
3. ...more to come!

Das Sortierproblem

Eingabe

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

Ausgabe

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

Beispiel

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

1) Insertion Sort

Inkrementelle Algorithmen

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)

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

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

InsertionSort

Idee

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

Algorithmus 3.1: InsertionSort(A)

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

Beispiel

$key = 99$

$$A = \langle \overbrace{7}^{\text{sortiert}}, 99, 12, 3, 17, 12 \rangle$$

i $\quad$ j

Was ist die Grundidee des Algorithmus?

- betrachte Variable $key \leftarrow A[j]$ im j-ten 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)

```
1  for j ← 2 to length(A)
2      key ← A[j]
3      i ← j − 1
4      while i > 0 and A[i] > key
5          A[i + 1] ← A[i]
6          i ← i − 1
7      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 von Theorem 3.1 (2/3)

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

- gelte $I(j)$ am Anfang des j-ten 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. $I(j)$

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

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

Beweis von Theorem 3.1 (3/3)

Hilfsinvariante $H(j, i)$

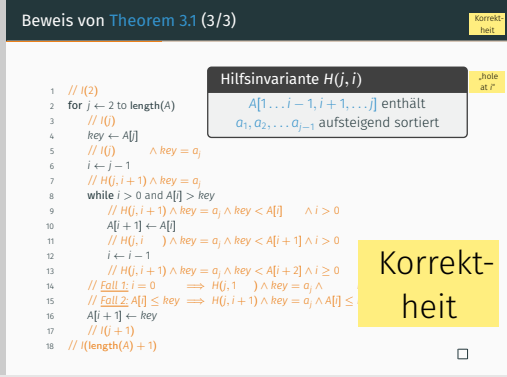
$A[1 \ldots i-1, i+1, \ldots j]$ enthält
$a_1, a_2, \ldots a_{j-1}$ aufsteigend sortiert

„hole at i“

```
1   // I(2)
2   for j ← 2 to length(A)
3       // I(j)
4       key ← A[j]
5       // I(j)          ∧ key = a_j
6       i ← j − 1
7       // H(j, i + 1) ∧ key = a_j
8       while i > 0 and A[i] > key
9           // H(j, i + 1) ∧ key = a_j ∧ key < A[i]      ∧ i > 0
10          A[i + 1] ← A[i]
11          // H(j, i     ) ∧ key = a_j ∧ key < A[i + 1] ∧ i > 0
12          i ← i − 1
13          // H(j, i + 1) ∧ key = a_j ∧ key < A[i + 2] ∧ i ≥ 0
14      // Fall 1: i = 0          ⟹ H(j, 1     ) ∧ key = a_j ∧          key < A[2]
15      // Fall 2: A[i] ≤ key ⟹ H(j, i + 1) ∧ key = a_j ∧ A[i] ≤ key < A[i + 2]
16      A[i + 1] ← key
17      // I(j + 1)
18  // I(length(A) + 1)
```

□

Beweis von Theorem 3.1 (3/3)



- 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
1	**for** $j \leftarrow 2$ to **length**(A)	$\sum_{j=2}^{n} T(l)$
2	$key \leftarrow A[j]$	$O(1)$
3	$i \leftarrow j-1$	$O(1)$
4	**while** $i > 0$ and $A[i] > key$	$\leq \sum_{i=1}^{j-1} T(l)$
5	$A[i+1] \leftarrow A[i]$	$O(1)$
6	$i \leftarrow i-1$	$O(1)$
7	$A[i+1] \leftarrow key$	$O(1)$

über $\Phi(i) = i$

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

Beweis von Theorem 3.2 (obere und untere Schranke)

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
1 **for** $j \leftarrow 2$ to **length**(A)	$\sum_{j=2}^{n} T(I)$
2 $key \leftarrow A[j]$	O(1)
3 $i \leftarrow j-1$	O(1)
4 **while** $i > 0$ and $A[i] > key$	$\leq \sum_{i=1}^{j-1} T(I)$ über $\Phi(i) = i$
5 $A[i+1] \leftarrow A[i]$	O(1)
6 $i \leftarrow i-1$	O(1)
7 $A[i+1] \leftarrow key$	O(1)

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

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

2) Merge Sort

Divide & Conquer Algorithmen

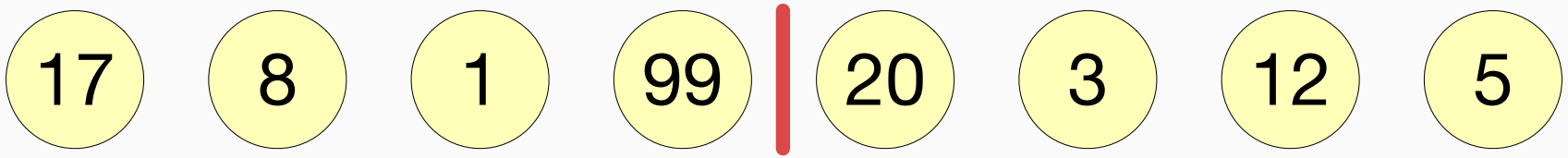
Definition 3.2

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

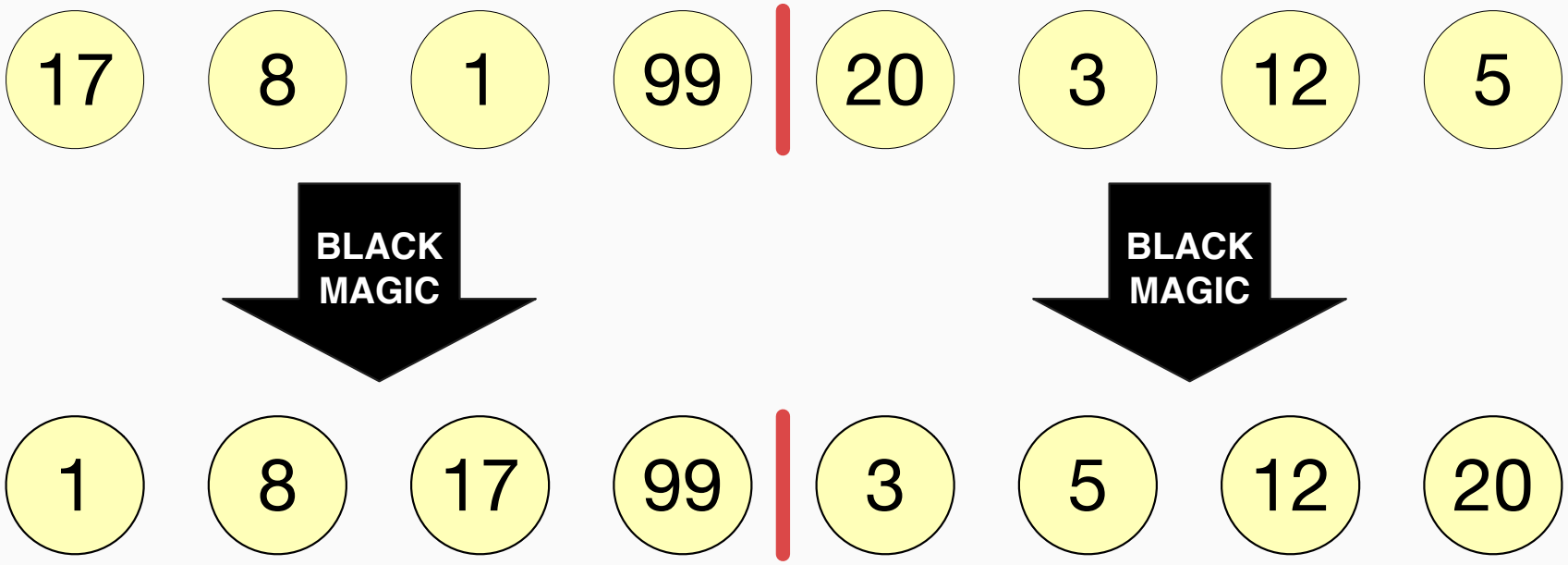
1. Teile das Problem in mehrere Teilprobleme auf.
2. Erobere große Teilproblem durch rekursive Aufrufe und löse kleine Teilprobleme direkt.
3. Kombiniere die Lösungen der Teilprobleme zu einer Gesamtlösung.

Teile & Erobere

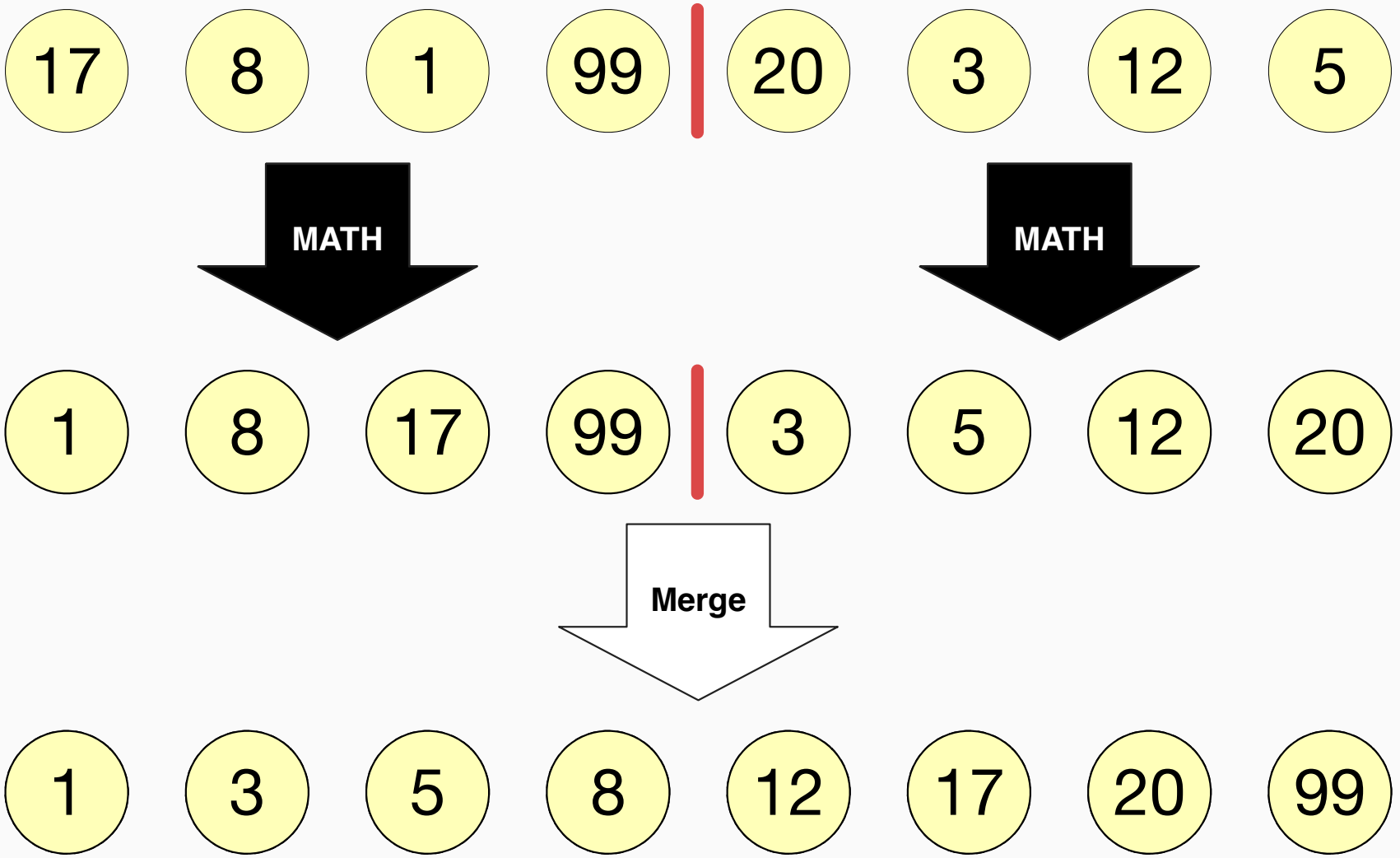
Idee: MergeSort



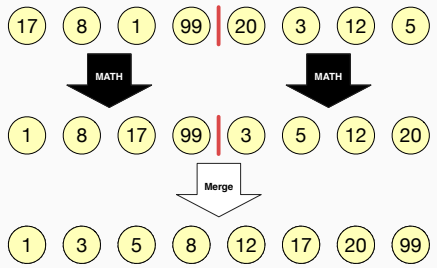
Idee: MergeSort



Idee: MergeSort



Idee: MergeSort



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

Pseudocode zu MergeSort

Algorithmus 3.2: MergeSort(A, l, r)

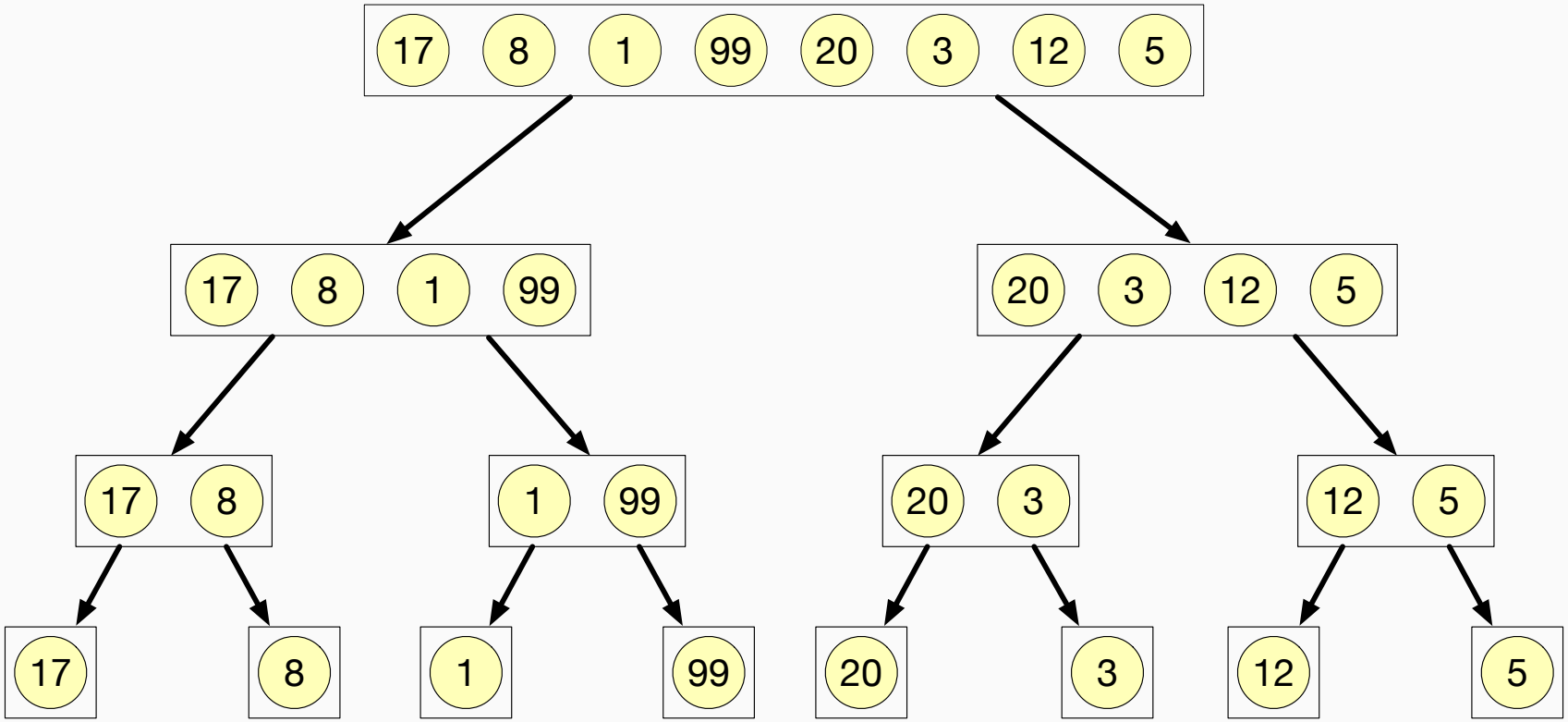
```
1  if l < r
2      p ← ⌊(l + r)/2⌋
3      MergeSort(A, l     , p)
4      MergeSort(A, p + 1, r)
5      Merge(A, l, p, r)
```

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

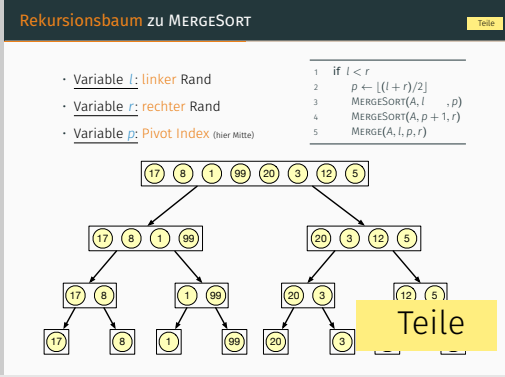
Rekursionsbaum zu MergeSort

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

```
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)
```



Rekursionsbaum zu MergeSort

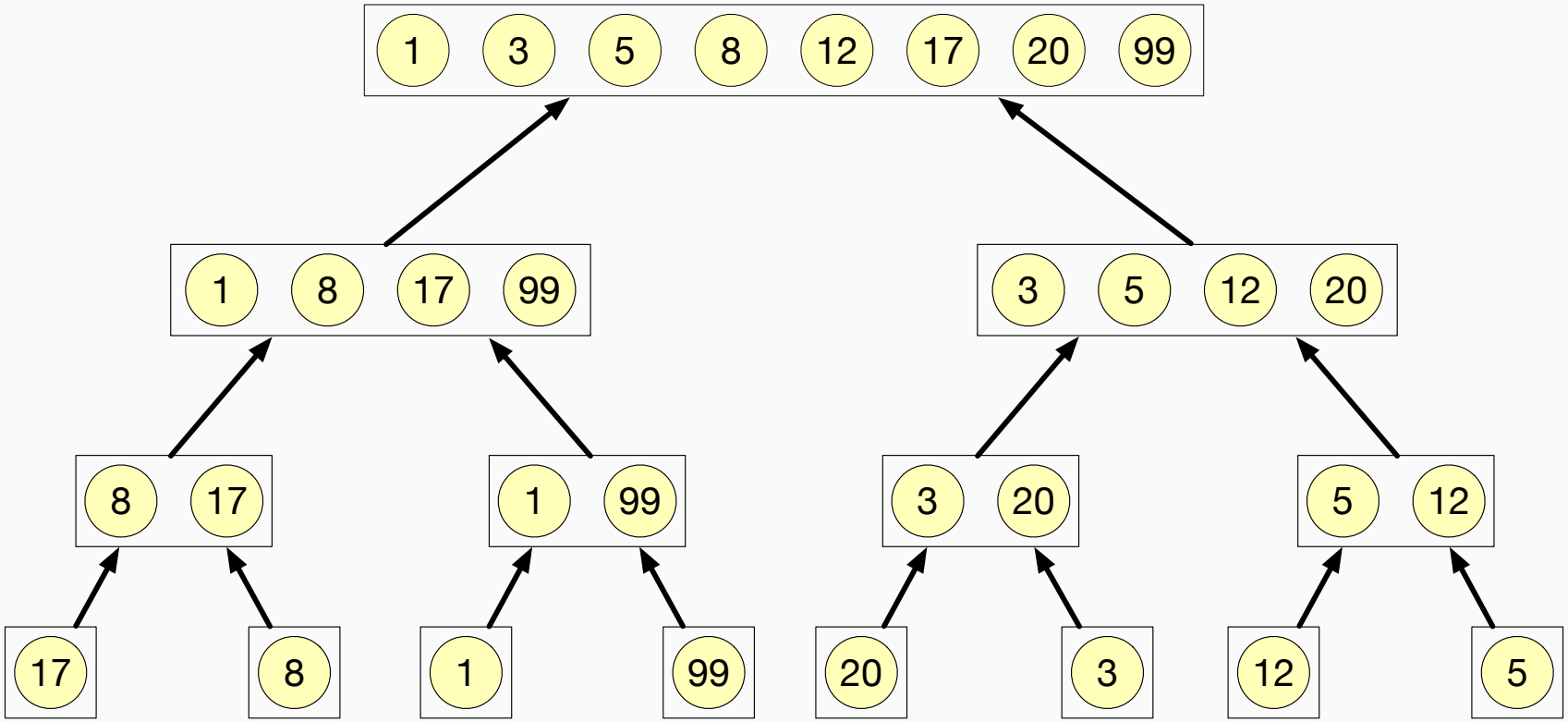


- MergeSort teilt das Array in der Mitte
- andere Teilungsstrategien denkbar; werden wir noch sehen
- Pivot Index nicht mit Pivot Element verwechseln; kommt später

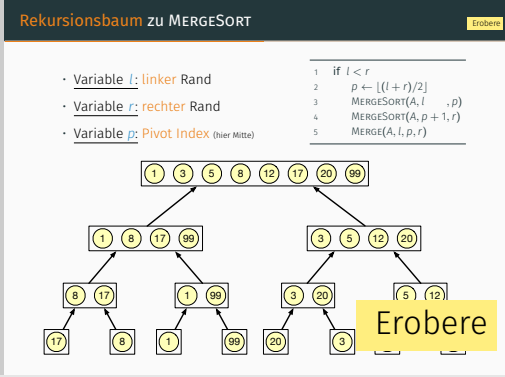
Rekursionsbaum zu MergeSort

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

```
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)
```



Rekursionsbaum zu MergeSort



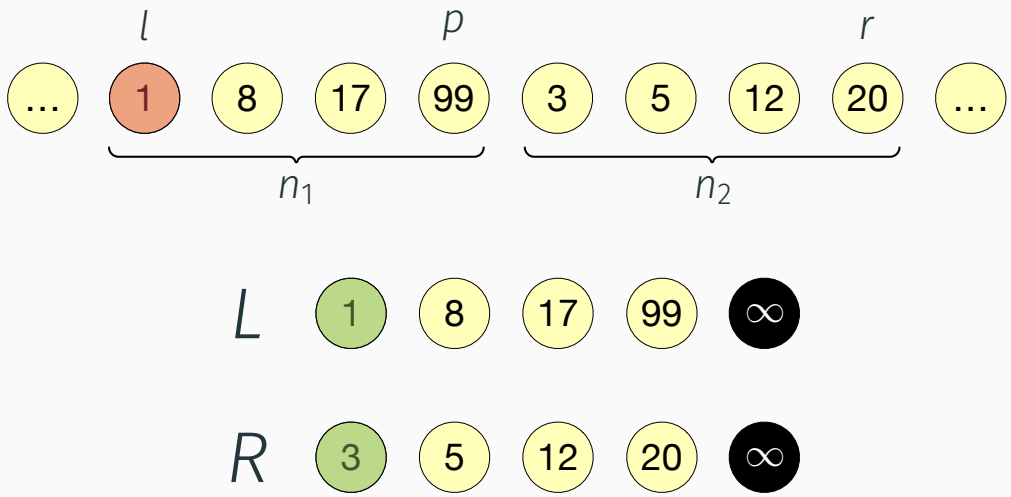
- MergeSort teilt das Array in der Mitte
- andere Teilungsstrategien denkbar; werden wir noch sehen
- Pivot Index nicht mit Pivot Element verwechseln; kommt später

Wie genau funktioniert MERGE?

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

```
1   n_1 ← p − l + 1
2   n_2 ← r − p
3   for i ← 1 to n_1: L[i] ← A[l + i − 1]
4   for j ← 1 to n_2: R[j] ← A[p + j]
5   L[n_1 + 1] ← ∞
6   R[n_2 + 1] ← ∞
7   i ← 1; j ← 1
8   for k ← l to r
9       if L[i] ≤ R[j]
10          A[k] ← L[i]
11          i ← i + 1
12      else
13          A[k] ← R[j]
14          j ← 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 $\Longrightarrow$ aktueller Aufruf korrekt

Anmerkung zur Erhaltung

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

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

Beweis von Theorem 3.3 (1/2)

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 $\Phi(\bullet)$: $\Phi(A, l, r) = l - r$
  - sinkt pro Aufruf um mindestens 1 (siehe Zeilen 3 und 4)
  - ist garantiert nichtnegativ

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)
```

Beweis von Theorem 3.3 (1/2)

Beweis von Theorem 3.3 (1/2)

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 $\Phi(\bullet)$: $\Phi(A, l, r) = l - r$
  - sinkt pro Aufruf um mindestens 1 (siehe Zeilen 3 und 4)
  - ist garantiert nichtnegativ

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)
```

Korrekt-heit

- 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)

Initialisierung & Erhaltung (✓)

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

nur ein Element

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

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

---

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)
```

□

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 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)

```
1   n_1 ← p − l + 1
2   n_2 ← r − p
3   for i ← 1 to n_1: L[i] ← A[l + i − 1]
4   for j ← 1 to n_2: R[j] ← A[p + j]
5   L[n_1 + 1] ← ∞
6   R[n_2 + 1] ← ∞
7   i ← 1; j ← 1
8   for k ← l to r
9       if L[i] ≤ R[j]
10          A[k] ← L[i]
11          i ← i + 1
12      else
13          A[k] ← R[j]
14          j ← j + 1
```

Müssen also noch MERGE analysieren!

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 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)

```
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  for k ← l to r
9      if L[i] ≤ R[j]
10         A[k] ← L[i]
11         i ← i + 1
12     else
13         A[k] ← R[j]
14         j ← j + 1
```

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

Beweis von Lemma 3.1 (1/2)

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 wieder 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

$\Longrightarrow$ MERGE ist korrekt

Also alle!

Beweis von Lemma 3.1 (2/2)

MERGE(A, l, p, r)

```
1   n_1 ← p − l + 1
2   n_2 ← r − p
3   for i ← 1 to n_1: L[i] ← A[l + i − 1]
4   for j ← 1 to n_2: R[j] ← A[p + j]
5   L[n_1 + 1] ← ∞
6   R[n_2 + 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 $I(i, j, k)$

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

- gelte $I(i, j, k)$ vor dem k-ten 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 $A[l \ldots k]$ 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-ten Schleifendurchlaufs ✓ □

Beweis von Lemma 3.2

MERGE(A, l, p, r)	Kosten
1 $n_1 \leftarrow p - l + 1$	$\Theta(1)$
2 $n_2 \leftarrow r - p$	$\Theta(1)$
3 **for** $i \leftarrow 1$ to n_1: $L[i] \leftarrow A[l + i - 1]$	$\Theta(n_1)$
4 **for** $j \leftarrow 1$ to n_2: $R[j] \leftarrow A[p + j]$	$\Theta(n_2)$
5 $L[n_1 + 1] \leftarrow \infty$	$\Theta(1)$
6 $R[n_2 + 1] \leftarrow \infty$	$\Theta(1)$
7 $i \leftarrow 1; j \leftarrow 1$	$\Theta(1)$
8 **for** $k \leftarrow l$ to r	$\Theta(r - l)$
9 **if** $L[i] \leq R[j]$	$\Theta(1)$
10 $A[k] \leftarrow L[i]$	$\Theta(1)$
11 $i \leftarrow i + 1$	$\Theta(1)$
12 **else**	$\Theta(1)$
13 $A[k] \leftarrow R[j]$	$\Theta(1)$
14 $j \leftarrow j + 1$	$\Theta(1)$

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

Combine

Rekursionsformel für MergeSort

obere Schranke

Lemma 3.3

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

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

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 $D(n) = \Theta(1)$ — 2 rek. Aufrufe
- Kombinierung benötigt Zeit $C(n) \leq \text{const} \cdot n$ — Lemma 3.2
- wähle $c_1 = \max\{ c_B, \text{const} \}$

□

Rekursionsformel für MergeSort

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

Rekursionsformel für MergeSort

Lemma 3.3

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

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

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 $C(n) \leq$ const $\cdot n$
- wähle $c_1 = \max\{ c_B, \text{const} \}$ □

obere Schranke

Rekursionsformel für MergeSort

untere Schranke

Lemma 3.4

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

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

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 $D(n) = \Theta(1)$ — 2 rek. Aufrufe
- Kombinierung benötigt Zeit $C(n) \geq \text{const}' \cdot n$ — Lemma 3.2
- wähle $c_2 = \min\{c_B, \text{const}'\}$

□

Rekursionsformel für MergeSort

Rekursionsformel für MergeSort

Lemma 3.4

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

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

Beweis.

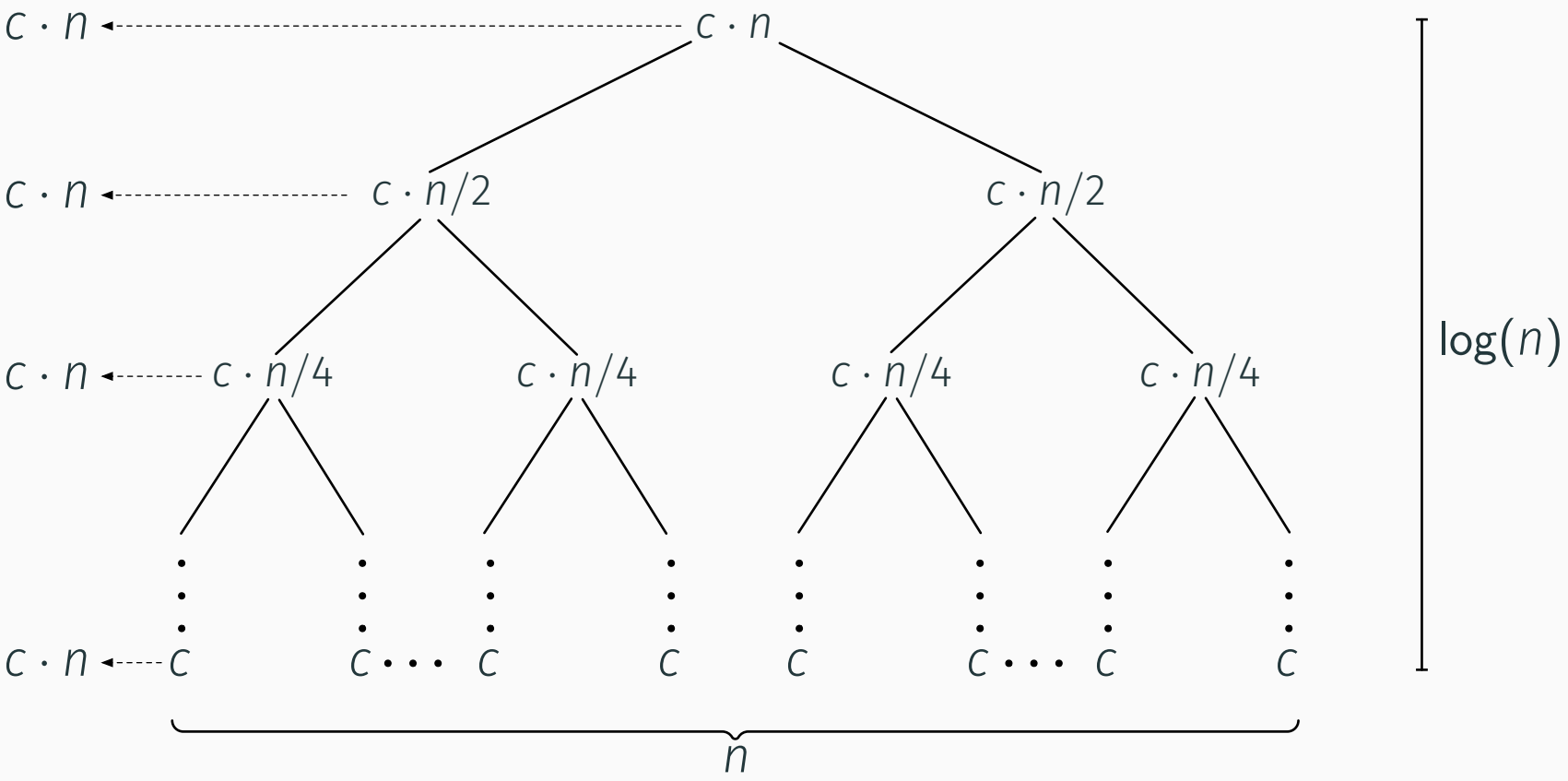
- Basisfall hat Größe $n_B = 1$ und benötigt ko
- jeder Aufruf erzeugt $a = 2$ Teilprobleme de
- Aufteilung benötigt konstante Zeit $D(n) =$
- Kombinierung benötigt Zeit $C(n) \geq \text{const}' \cdot n$
- wähle $c_2 = \min \{ c_B, \text{const}' \}$ □

untere Schranke

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

Laufzeit von MergeSort aus der Rekursionsformel

Mit Lemma 3.4 und Lemma 3.3 kann man Theorem 3.4 beweisen!



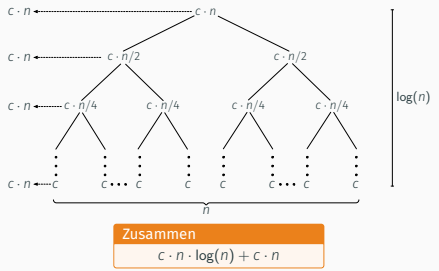
Zusammen

$$c \cdot n \cdot \log(n) + c \cdot n$$

Laufzeit von MergeSort aus der Rekursionsformel

Laufzeit von MergeSort aus der Rekursionsformel

Mit Lemma 3.4 und Lemma 3.3 kann man Theorem 3.4 beweisen!



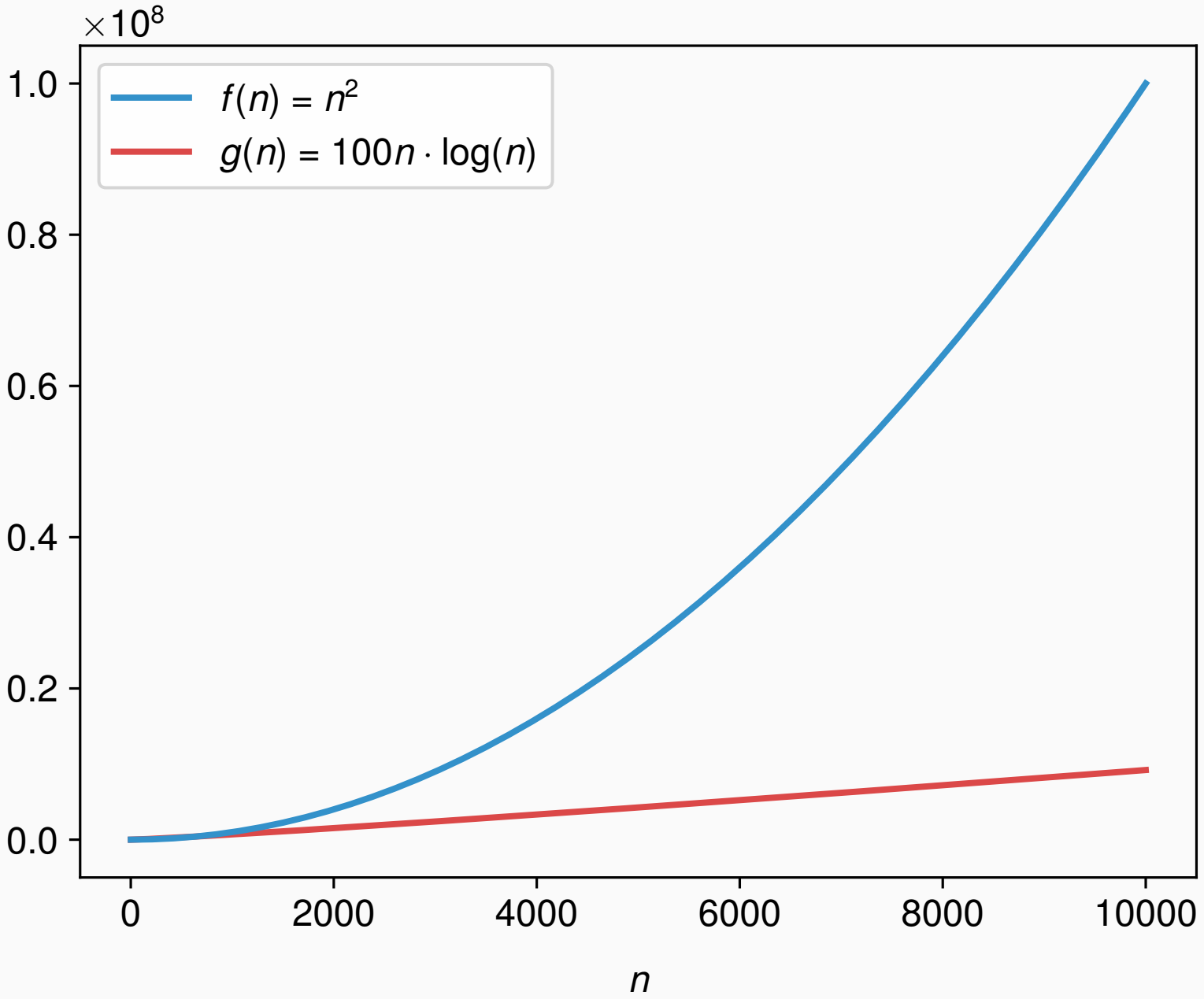
Zusammen

$c \cdot n \cdot \log(n) + c \cdot n$

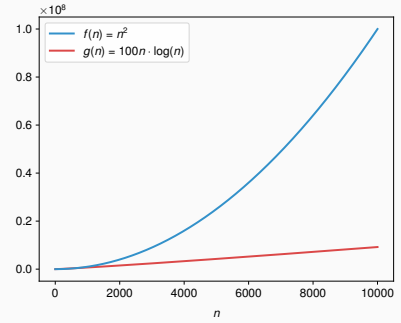
- jede Kante ist ein rekursiver Aufruf $\rightsquigarrow$ Kosten $\Theta(1)$ pro Kante
- jedes Blatt ist ein Basisfall $\rightsquigarrow$ Kosten $\Theta(1)$ pro Blatt
- lernen noch systematische Methode kennen, um die Lösung solch rekursiver Gleichungen für Laufzeiten zu berechnen

$\rightsquigarrow$ Stichwort Master Theorem

InsertionSort vs MergeSort



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) …more to come!