4. Divide & Conquer – Merge-Sort

Definition 4.1: Divide&Conquer (Teile&Erobere) ist eine auf Rekursion beruhende Algorithmentechnik.

Eine Divide&Conquer-Algorithmus löst ein Problem in 3 Schritten:

- Teile ein Problem in mehrere Unterprobleme.
- Erobere jedes einzelne Unterproblem durch rekursive Lösung. Ausnahme sind kleine Unterprobleme, diese werden direkt gelöst.
- Kombiniere die Lösungen der Teilprobleme zu einer Gesamtlösung.

Divide&Conquer und Sortieren

- Teile ein Problem in Unterprobleme.
- Erobere jedes einzelne Teilproblem, durch rekursive Lösung.
- Kombiniere die Lösungen zu einer Gesamtlösung.

- Teile eine *n*-elementige Teilfolge auf in zwei Teilfolgen mit jeweils etwa *n/2* Elementen.
- Sortiere die beiden Teilfolgen rekursiv.
- Mische die sortierten Teilfolgen zu einer sortierten Gesamtfolge.

Merge-Sort

Merge-Sort ist eine mögliche Umsetzung des Divide&Conquer-Prinzips auf das Sortierproblem.

$\text{Merge-Sort}(A,p,r)$

1 **if** $p < r$
2 **then** $q \leftarrow \lfloor (p+r)/2 \rfloor$
3 $\text{Merge-Sort}(A,p,q)$
4 $\text{Merge-Sort}(A,q+1,r)$
5 $\text{Merge}(A,p,q,r)$

- Merge ist Algorithmus zum Mischen zweier sortierter Teilfolgen
- Aufruf zu Beginn mit Merge-Sort(*A*,1,*length*(*A*)).

Illustration von Merge-Sort (1)

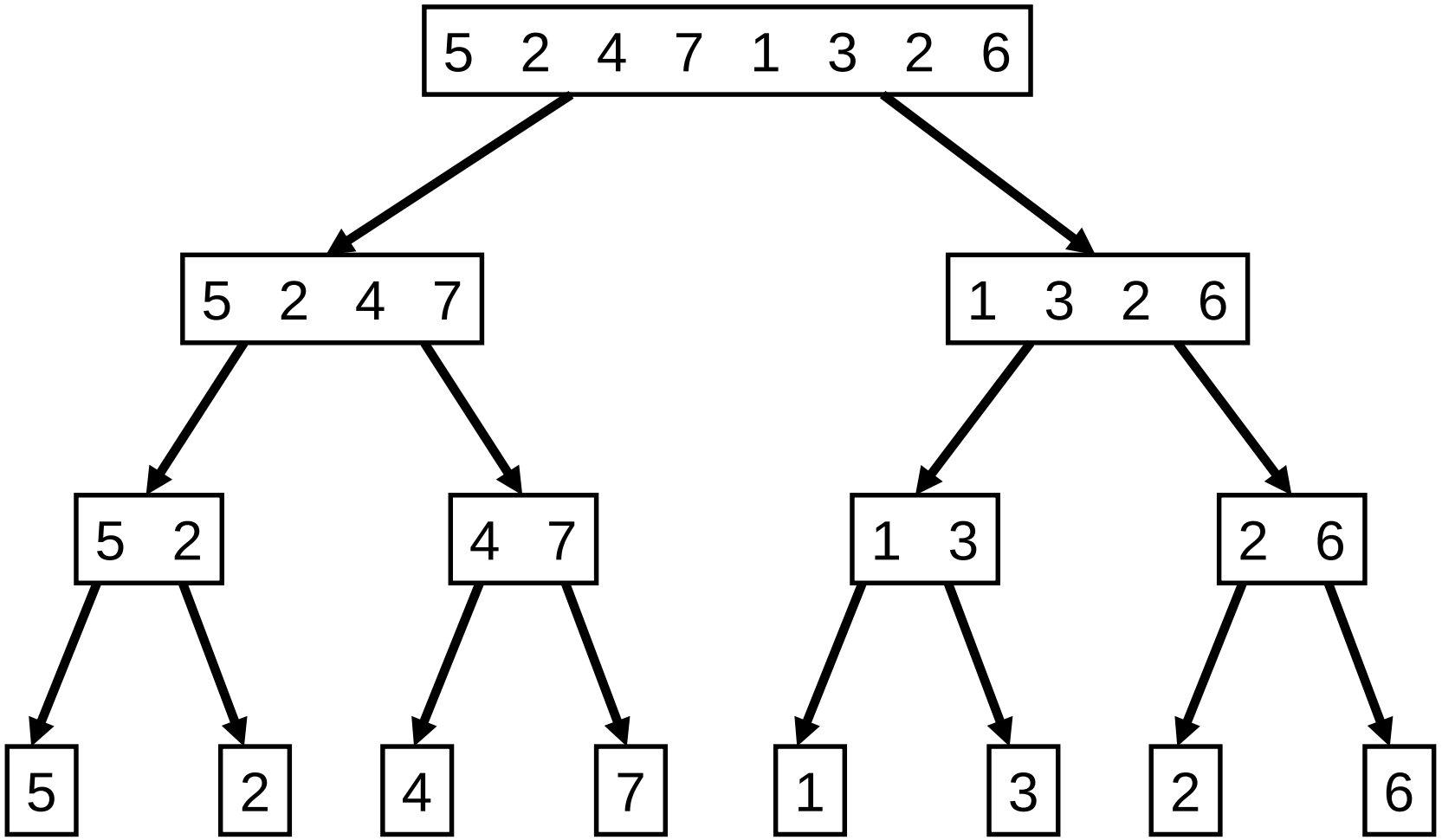
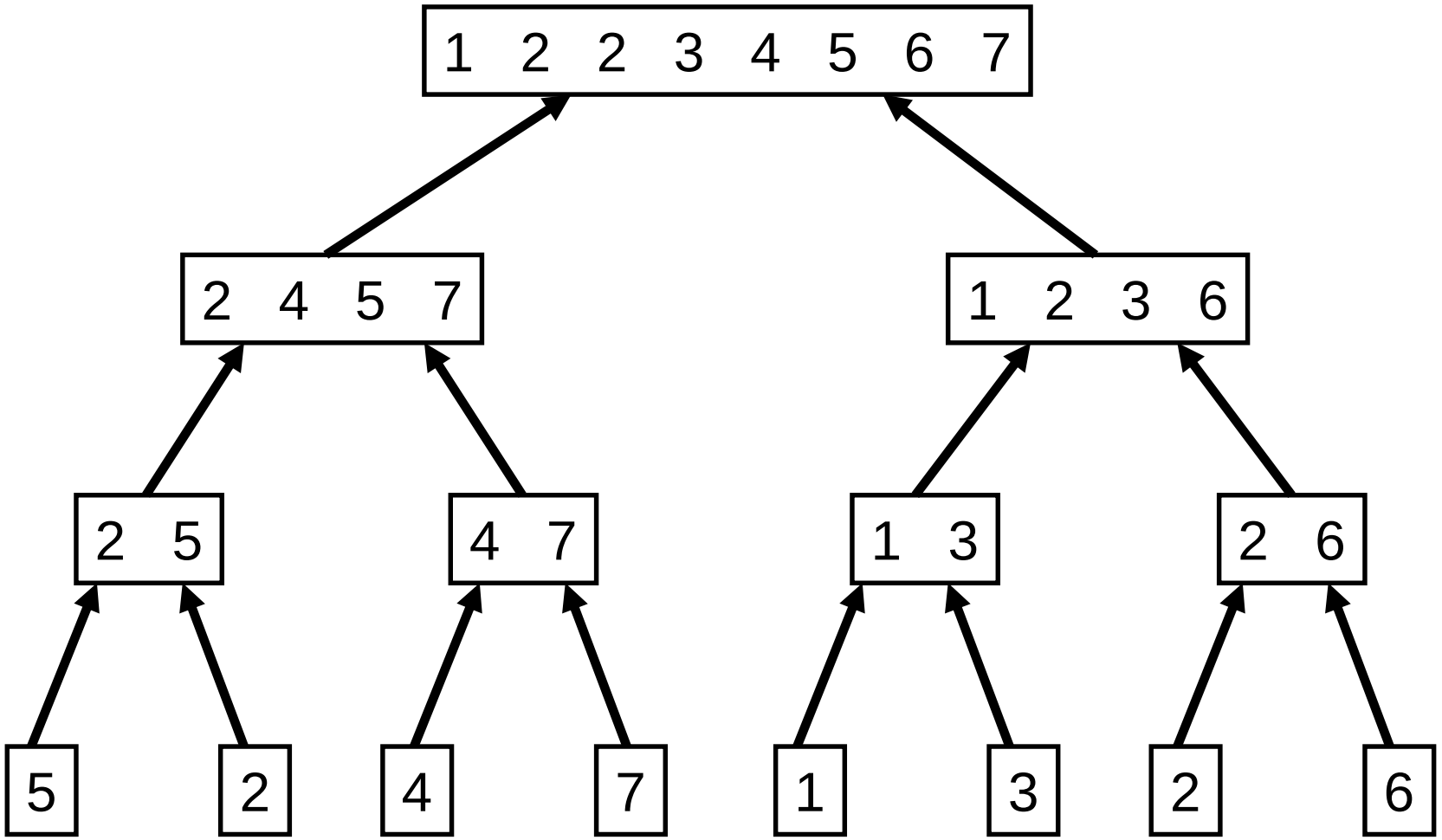


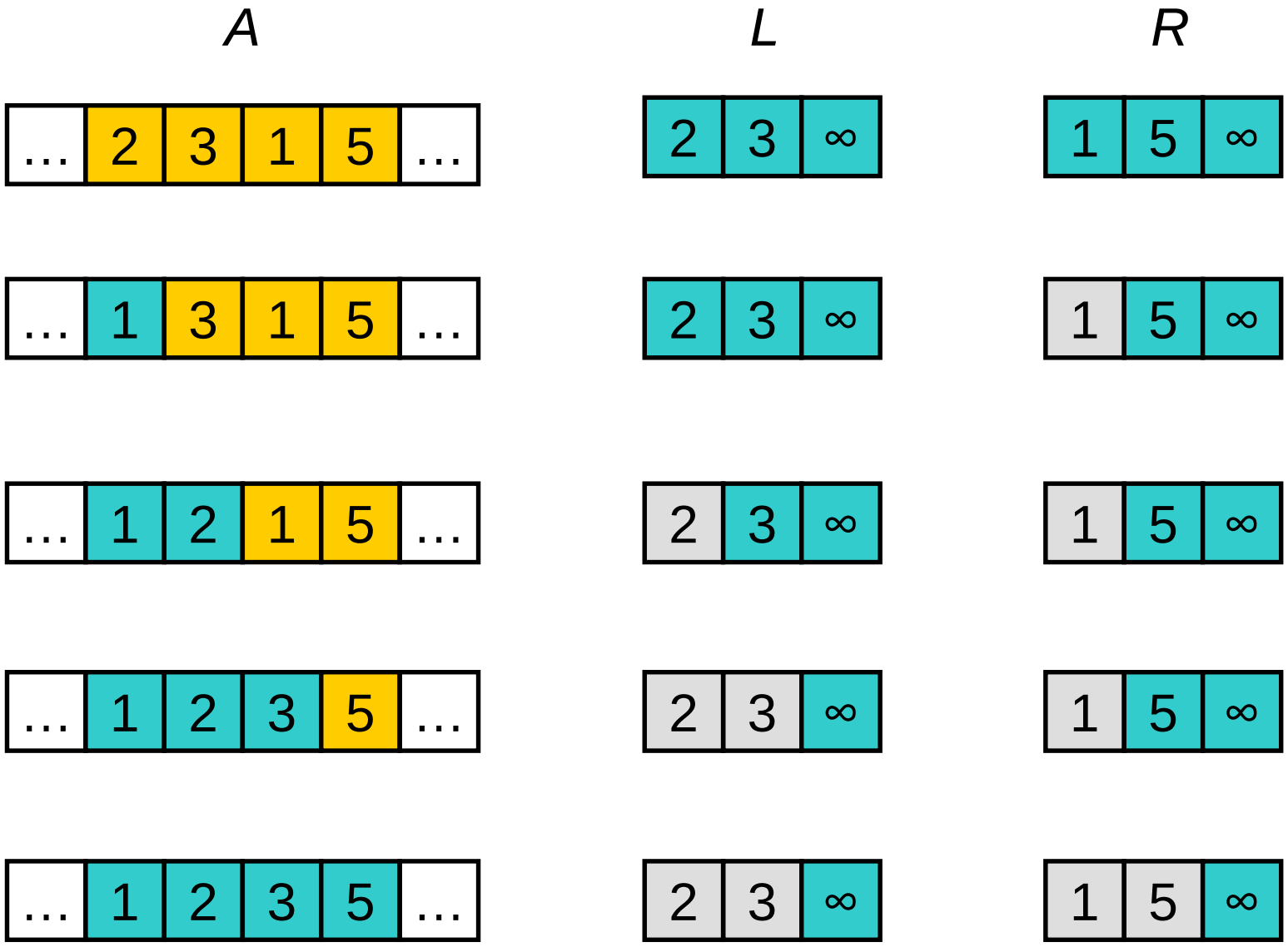
Illustration von Merge-Sort (2)



Kombination von Teilfolgen - Merge

```
Merge(A, p, q, r)
 1  n_1 ← q − p + 1
 2  n_2 ← r − q
 3  ▷ Erzeuge Arrays L[1.. n_1 + 1], R[1.. n_2 + 1]
 4  for i ← 1 to n_1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n_2
 7       do R[j] ← A[q + j]
 8  L[n_1 + 1] ← ∞
 9  R[n_2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14            then A[k] ← L[i]
15                 i ← i + 1
16            else A[k] ← R[j]
17                 j ← j + 1
```

Illustration von Merge



Korrektheit rekursiver Algorithmen

Die Korrektheit rekursiver Algorithmen wie Merge-Sort wird üblicherweise ähnlich zur vollständigen Induktion gezeigt.

Konkret muss gezeigt werden, dass

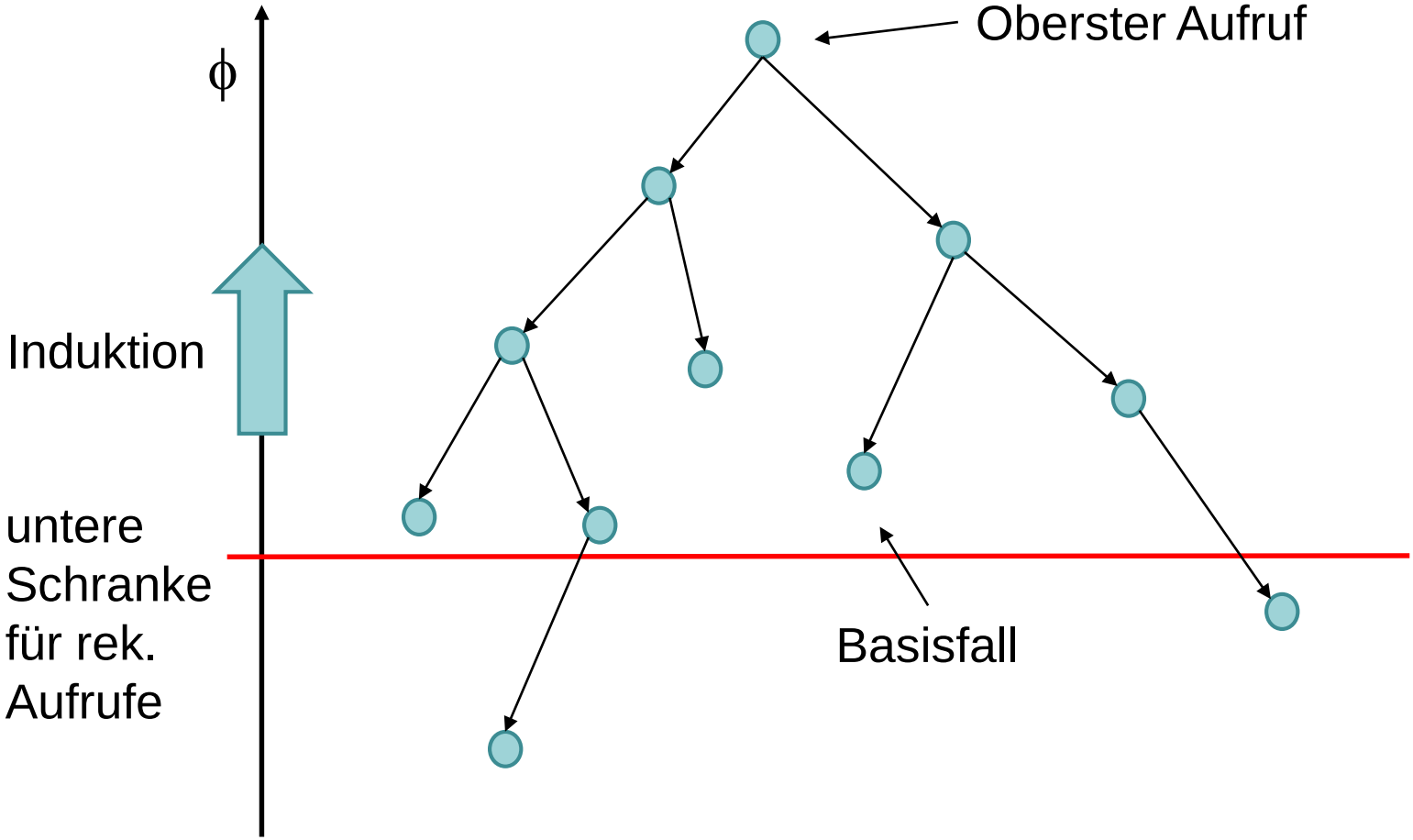
1. der Algorithmus für den Basisfall (keine weiteren rekursiven Aufrufe) korrekt ist (Initialisierung) und
2. falls die rekursiven Aufrufe des Algorithmus korrekt sind, dann auch der aktuell ausgeführte Aufruf korrekt ist (Erhaltung).

Um sicherzustellen, dass die Annahmen in der Erhaltung korrekt sind, muss eine Potenzialfunktion $\phi()$ ein $\delta>0$ gefunden werden, das unabhängig von der Rekursionstiefe ist, so dass für den Fall von rek. Aufrufen

1. ϕ um mindestens δ sinkt (bzw. steigt) und
2. ϕ nach unten (bzw. oben) hin beschränkt ist.

Korrektheit rekursiver Algorithmen

Anschaulich (mon. sinkendes ϕ):



Korrektheit rekursiver Algorithmen

Beispiel: Berechnung der Fakultät

```
Fakultät(n)
  if n=1 then return 1
         else return n*Fakutät(n-1)
```

Behauptung: Fakultät(n)=n!

- Initialisierung: n=1 (keine weiteren rekursiven Aufrufe)
  Fakultät(1) =1=1!
- Erhaltung: Wir nehmen an, dass Fakultät(n-1)=(n-1)!.
  Dann gilt für den Aufruf Fakultät(n):
  Fakultät(n) = n·Fakultät(n-1) = n·(n-1)! = n!

Annahme in der Erhaltung korrekt: betrachte $\phi(n)=n$.

- ϕ wird bei jedem rekursiven Aufruf um 1 vermindert und
- es muss $\phi>1$ sein, damit ein rekursiver Aufruf stattfindet.

Terminierung rekursiver Algorithmen

Analog zu while/repeat Schleifen kann mittels einer Potenzialfunktion auch die Terminierung rekursiver Algorithmen nachgewiesen warden.

Beispiel: Berechnung der Fakultät

```
Fakultät(n)
  if n=1 then return 1
         else return n*Fakutät(n-1)
```

Terminierung: betrachte $\phi(n)=n$.

- ϕ wird bei jedem rekursiven Aufruf um 1 vermindert und
- es muss $\phi>1$ sein, damit ein rekursiver Aufruf stattfindet.

Also ist die Rekursionstiefe bei Aufruf von Fakultät(n) höchstens n. Es bleibt daher nachzuweisen, dass Fakultät(n) unter Annahme der Terminierung der rekursiven Aufrufe terminiert. Das ist aber trivialerweise der Fall, da Fakultät(n) keine while- oder repeat-Schleife verwendet.

Korrektheit von Merge - Sort

Behauptung: Merge-Sort(A,p,r) sortiert A[p,…,r]
Zunächst geeignete Wahl einer Potenzialfunktion.
Initialisierung: $p \geq r$.
Erhaltung: $p<r$, und bei jedem Aufruf wird $r-p$ um mindestens 1 auf einen nichtnegativen Wert reduziert.
Also geeignete Wahl von $\phi(A,p,r)$: $\phi(A,p,r) = r-p$

Merge - Sort(A, p, r)

1 **if** $p < r$
2 **then** $q \leftarrow \lfloor (p+r)/2 \rfloor$
3 Merge - Sort(A, p, q)
4 Merge - Sort$(A, q+1, r)$
5 Merge(A, p, q, r)

Korrektheit von Merge - Sort

Behauptung: Merge-Sort(A,p,r) sortiert A[p…r]
Initialisierung: Für p≥r ist A[p…r] trivialerweise sortiert
Erhaltung: Nach den rekursiven Aufrufen sind A[p…q] und A[q+1…r] sortiert. Mischt also Merge(A,p,q,r) A[p…q] und A[q+1…r] korrekt zu einer sortierten Folge, ist dann auch A[p…r] sortiert.

$\text{Merge-Sort}(A,p,r)$

1 **if** $p < r$
2 **then** $q \leftarrow \lfloor (p+r)/2 \rfloor$
3 $\text{Merge-Sort}(A,p,q)$
4 $\text{Merge-Sort}(A,q+1,r)$
5 $\text{Merge}(A,p,q,r)$

Korrektheit von Merge - Invariante

Lemma 4.2: Erhält Algorithmus Merge(A,p,q,r) als Eingabe ein Teilarray $A[p \ldots r]$, so dass die beiden Teilarrays $A[p \ldots q]$ und $A[q+1 \ldots r]$ sortiert sind, so ist nach Durchführung von Merge das Teilarray $A[p \ldots r]$ ebenfalls sortiert.

Schleifeninvariante I(k): Array $A[p \ldots k\text{-}1]$ enthält die $k\text{-}p$ kleinsten Zahlen aus den Arrays L und R in sortierter Reihenfolge.

Korrektheit von Merge – 3 Schritte

Initialisierung: Vor der Schleife gilt offensichtlich I(p) und damit auch I(k) für k=p.

Erhaltung:

• Angenommen, I(k) gilt zu Anfang des Schleifendurchlaufs.

• Sei o.B.d.A. $L[i] \leq R[j]$. Dann ist $L[i]$ das kleinste noch nicht einsortierte Element. Nach Ausführung der Zeilen 14-15 enthält $A[p..k]$ die $k-p+1$ kleinsten Elemente. Zusammen mit Erhöhung des Zählers i garantiert dies, dass am Ende der Schleife I(k+1) gilt.

Terminierung: Nach Ende der Schleife enthält $A[p..r]$ die $r-p+1$ kleinsten Elemente in sortierter Reihenfolge. Also sind dann alle Elemente sortiert.

Korrektheit von Merge – Formal

```
Merge(A,p,q,r)
1       n1←q-p+1
2       n2←r-q
3       for i←1 to n1 do
4           L[i]←A[p+i-1]
5       for j←1 to n2 do
6           R[j]←A[q+j]
7       L[n1+1]←∞
8       R[n2+1]←∞
9       i←1;j←1
        ▷ I(p)
10      for k←p to r do
            ▷ I(k)
11          if L[i]≤R[j] then
                ▷ I(k) ∧ L[i]≤R[j]
12              A[k]←L[i]; i←i+1
                ▷ I(k+1)
13          else
                ▷ I(k) ∧ L[i]>R[j]
14              A[k]←R[j]; j←j+1
                ▷ I(k+1)
        ▷ I(r+1), d.h. A[p…r] ist sortiert
```

Laufzeit von Merge

Lemma 4.3: Ist die Eingabe von Merge ein Teilarray der Größe n, so ist die Laufzeit von Merge $\Theta(n)$.

		Zeit
	$\text{Merge}(A, p, q, r)$	
1	$n_1 \leftarrow q - p + 1$	$\Theta(1)$
2	$n_2 \leftarrow r - q$	$\Theta(1)$
3	for $i \leftarrow 1$ to n_1	$\Theta(n_1)$
4	do $L[i] \leftarrow A[p + i - 1]$	$\Theta(1)$
5	for $j \leftarrow 1$ to n_2	$\Theta(n_2)$
6	do $R[j] \leftarrow A[q + j]$	$\Theta(1)$
7	$L[n_1 + 1] \leftarrow \infty$	$\Theta(1)$
8	$R[n_2 + 1] \leftarrow \infty$	$\Theta(1)$
9	$i \leftarrow 1$	$\Theta(1)$
10	$j \leftarrow 1$	$\Theta(1)$
11	for $k \leftarrow p$ to r	$\Theta(r-p)$
12	do if $L[i] \leq R[j]$	r-p+1 Durchläufe
13	then $A[k] \leftarrow L[i]$	t_1
14	$i \leftarrow i + 1$	t_1
15	else $A[k] \leftarrow R[j]$	t_2
16	$j \leftarrow j + 1$	t_2

t_1+t_2=r-p+1

Laufzeit von D&C-Algorithmen

Allgemeiner Ansatz:

- $T(n)$: Gesamtlaufzeit bei Eingabegröße n
- a: Anzahl der Teilprobleme durch Teilung
- n/b: Größe der Teilprobleme
- $D(n)$: Zeit für die Teilung (Divide)
- $C(n)$: Zeit für die Kombinierung
- $n \leq u$: Basisfall für Algorithmus, für den dieser Laufzeit $\leq c$ hat

Dann gilt:

$$T(n) \leq \begin{cases} c & \text{falls } n \leq u \\ a \cdot T(n/b) + D(n) + C(n) & \text{sonst} \end{cases}$$

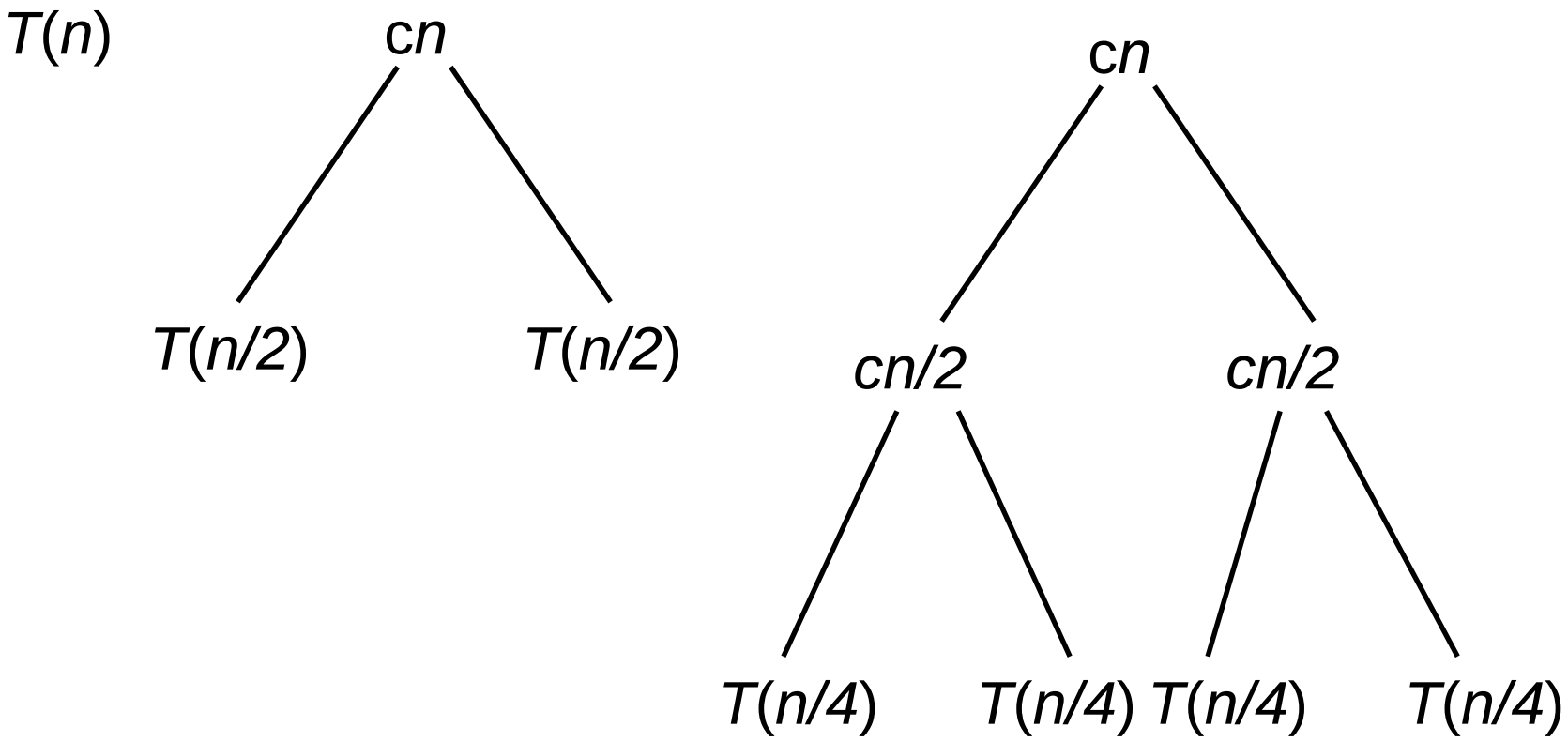
Laufzeit von Merge-Sort (1)

- $u = 1, a = 2, b \approx 2.$
- $D(n) = \Theta(1), C(n) = \Theta(n)$ (Lemma 4.3).
- Sei c so gewählt, dass eine Zahl in Zeit c sortiert werden kann und $D(n) + C(n) \leq cn$ gilt.

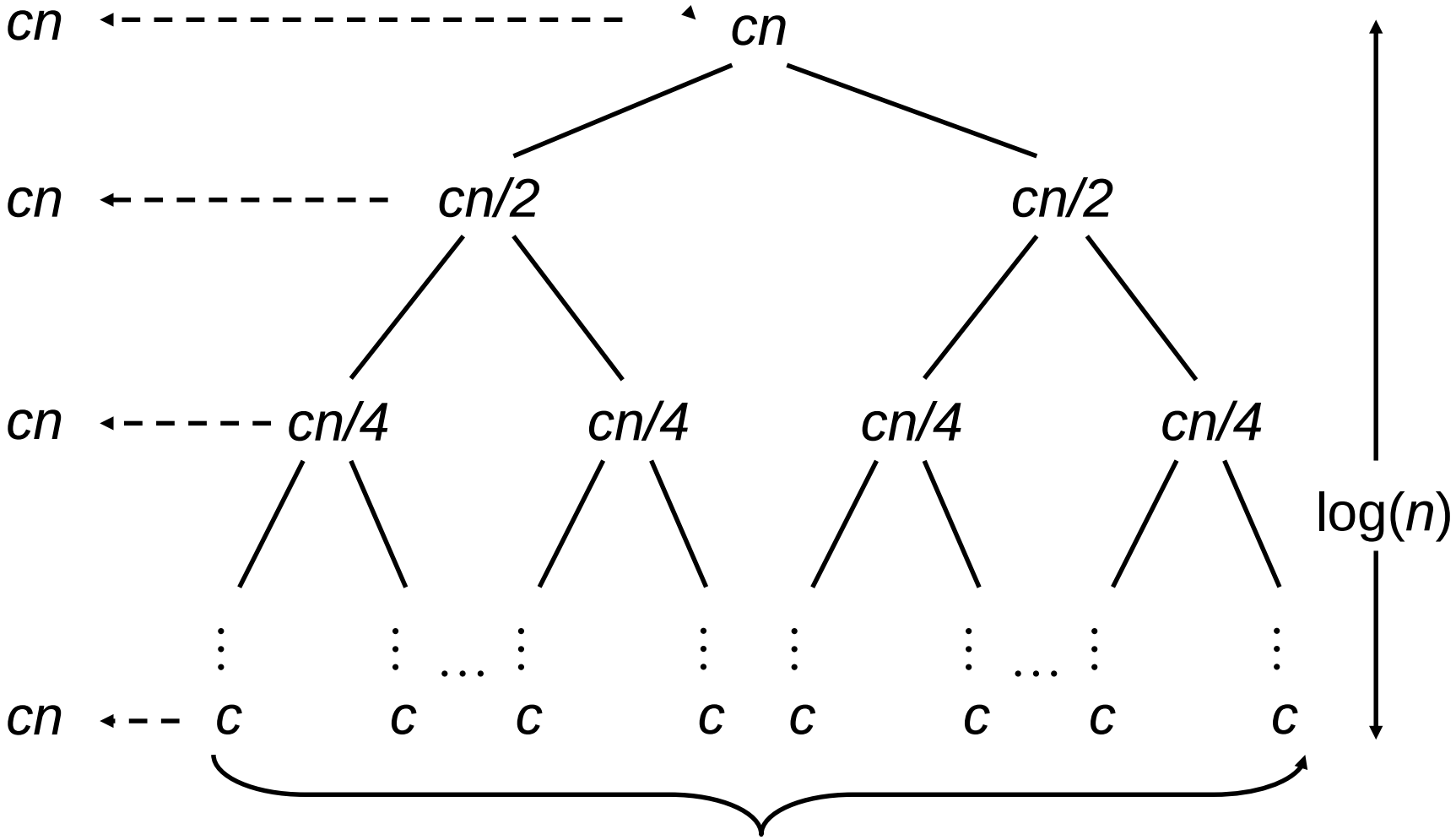
Lemma 4.4: Für die Laufzeit $T(n)$ von Merge-Sort gilt:

$$T(n) \leq \begin{cases} c & \text{falls } n \leq 1 \\ 2T(n/2) + cn & \text{sonst} \end{cases}$$

Laufzeit von Merge-Sort (2)



Laufzeit von Merge-Sort (3)



Zusammen: $cn \log(n)+cn$

Laufzeit von Merge-Sort (3)

Satz 4.5: Merge-Sort besitzt Laufzeit $\Theta(n\log(n))$.

Zum Beweis muss gezeigt werden:

1. Es gibt ein c_1, so dass die Laufzeit von Merge-Sort bei allen Eingaben der Größe n immer höchstens $c_2 n\log(n)$ ist.

2. Es gibt ein c_1, so dass für alle n eine Eingabe I_n der Größe n existiert bei der Merge-Sort mindestens Laufzeit $c_1 n\log(n)$ besitzt.

Laufzeit von Merge-Sort (4)

Satz 4.5: Merge-Sort besitzt Laufzeit $\Theta(n \log n)$.

Beweis:

Wir nehmen vereinfachend an, dass $n=2^k$ für ein $k \in \mathbb{N}$ ist. Aufgrund von Lemma 4.3 gibt es Konstanten $c_1>0$ und $c_2>0$ mit

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

und

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

Laufzeit von Merge-Sort (5)

Satz 4.5: Merge-Sort besitzt Laufzeit $\Theta(n \log n)$.

Beweis:

Aus Ungleichung (1) ergibt sich eine Laufzeit von $O(n \log n)$ und aus Ungleichung (2) ergibt sich eine Laufzeit von $\Omega(n \log n)$ für die Menge aller $n \in \mathbb{N}$ mit $n=2^k$ für ein $k \in \mathbb{N}$.

Im Allgemeinen muss gezeigt werden:

- $O(n \log n)$: es gibt ein $c_1>0$, so dass für alle $n \in \mathbb{N}$ die Laufzeit von Merge-Sort für alle Eingaben der Größe n höchstens $c_1 \cdot n \log n$ ist.
- $\Omega(n \log n)$: es gibt ein $c_2>0$, so dass für alle $n \in \mathbb{N}$ eine Einstanz I der Größe n existiert, für die Merge-Sort eine Laufzeit von mindestens $c_2 \cdot n \log n$ hat.

Laufzeit von Merge-Sort (6)

Eingabegröße n

Laufzeit	10	100	1,000	10,000	100,000
n^2	100	10,000	1,000,000	100,000,000	10,000,000,000
n log n	33	664	9,965	132,877	166,096

Beobachtung:

- n^2 wächst viel stärker als n log n
- Selbst bei großen Konst. wäre MergeSort schnell besser
- Konstanten spielen kaum eine Rolle
  $\rightarrow$ Θ-Notation ist entscheidend für große n

Average-Case Laufzeit

Average-case Laufzeit:

- Betrachten alle Permutationen der *n* Eingabezahlen.
- Berechnen für jede Permutation Laufzeit des Algorithmus bei dieser Permutation.
- Average-case Laufzeit ist dann der Durchschnitt über all diese Laufzeiten.

Definition 4.6: Eine Permutation ist eine bijektive Abbildung einer endlichen Menge auf sich selbst.

Alternativ: Eine Permutation ist eine Anordnung der Elemente einer endlichen Menge in einer geordneten Folge.

Average-Case Laufzeit

Lemma 4.7: Zu einer n-elementigen Menge gibt es genau $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$ Permutationen.

Beweis: Induktion über *n*.

(I.A.) *n*=1 klar.

(I.V.) Der Satz gilt für *n*.

(I.S.) *n*+1: An letzter Stelle steht die *i*-te Zahl. Es gibt *n!* unterschiedliche Anordnungen der restlichen *n* Zahlen. Da *i* jeden Wert zwischen 1 und *n*+1 annehmen kann, gibt es (*n*+1) *n!* = (*n*+1)*!* Anordnungen der *n*+1 Zahlen.

Beispiel: Menge $\{2,3,6\}$

Permutationen : $(2,3,6),(2,6,3),(3,2,6),(3,6,2),$
$(6,2,3),(6,3,2).$

Average-Case Laufzeit

Average-case Laufzeit:

- Wir betrachten alle Permutationen der *n* Eingabezahlen.
- Wir berechnen für jede Permutation Laufzeit des Algorithmus bei dieser Permutation.
- Average-case Laufzeit ist dann der Durchschnitt über all diese Laufzeiten.
- Average-case Laufzeit ist gleich der erwarteten Laufzeit einer zufällig und gleichverteilt gewählten Permutation aus der Menge aller Permutationen der *n* Eingabezahlen. Folgt aus Definition des Erwartungswerts (wird hier nicht behandelt).

Average-Case Laufzeit

Satz 4.7: Insertion-Sort besitzt average-case Laufzeit $\Theta(n^2)$.

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

- $L_{n/2}$: Menge der n/2 kleinsten Zahlen.
- Mit Wahrscheinlichkeit ½ gibt es mindestens $n/4$ Elemente A[j] in $L_{n/2}$ mit $j \geq n/2$.
- Sei $j \geq n/2$ und A[j] in $L_{n/2}$. Dann wird die while-Schleife mindestens $n/4$ mal durchlaufen.
  $\Rightarrow \geq n^2/16$ Vergleiche