Algorithmentheorie

Daniel Neuen (Universität Bremen)

WiSe 2023/24

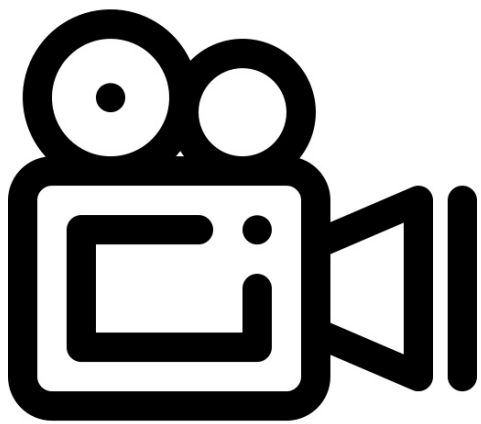
Asymptotische Laufzeit ($\mathcal{O}$-Notation) und Divide-and-Conquer

2. Vorlesung

Aufzeichnung der Vorlesung

Diese Vorlesung wird aufgezeichnet und live gestreamt.

- Aufzeichnungen nur der Lehrenden durch sich selbst.
- Bei Rückfragen aus dem Auditorium und Diskussion bitte deutlich anzeigen, falls das Mikro stumm geschaltet werden soll.



Organisatorisches

- **Tutorien**:
  - Montag 12:15 – 13:45 MZH 5500 (50/50)
  - Dienstag 12:15 – 13:45 MZH 1090 (45/94)
  - Donnerstag 12:15 – 13:45 MZH 5500 (50/50)
  - Donnerstag 12:15 – 13:45 MZH 5600 (56/58)

Organisatorisches

- **Tutorien**:
  - Montag 12:15 – 13:45 MZH 5500 (50/50)
  - Dienstag 12:15 – 13:45 MZH 1090 (45/94)
  - Donnerstag 12:15 – 13:45 MZH 5500 (50/50)
  - Donnerstag 12:15 – 13:45 MZH 5600 (56/58)

- **Übungsgruppen**: kleine Gruppen (< 3) gerne zusammenschließen

Organisatorisches

- **Tutorien**:
  - Montag 12:15 – 13:45 MZH 5500 (50/50)
  - Dienstag 12:15 – 13:45 MZH 1090 (45/94)
  - Donnerstag 12:15 – 13:45 MZH 5500 (50/50)
  - Donnerstag 12:15 – 13:45 MZH 5600 (56/58)
- **Übungsgruppen**: kleine Gruppen (< 3) gerne zusammenschließen
- **Dienstag Feiertag**: keine Übung, Hausaufgaben mit Musterlösung im StudIP

Wiederholung: Algorithmus und Analyse

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

Zu Entwurf und Analyse von Algorithmen gehören:

- Problemspezifikation
- Präzise Beschreibung des Algorithmus (Pseudocode)
- Korrektheitsbeweis
- Laufzeitanalyse

Laufzeitanalyse Insertion Sort

1	`i := 1`	1
2	**while** `i < length(A)` **do**	n
3	`j := i`	$n-1$
4	**while** `j > 0 and A[j − 1] > A[j]` **do**	$\sum_{i=1}^{n-1} t_i$
5	**swap** `A[j]` and `A[j − 1]`	$\sum_{i=1}^{n-1} t_i - 1$
6	`j := j − 1`	$\sum_{i=1}^{n-1} t_i - 1$
7	`i := i + 1`	$n-1$

Worst-Case Laufzeit: c_i Anzahl der Prozessorbefehle für Zeile i

$$c_1 + (c_2 + c_3 + c_7)n + (c_4 + c_5 + c_6)\frac{n(n-1)}{2}.$$

Laufzeitanalyse Insertion Sort

1	`i := 1`	1
2	**while** `i` $<$ `length(A)` **do**	n
3	`j := i`	$n-1$
4	**while** `j` $>$ `0` and `A[j` $-$ `1]` $>$ `A[j]` **do**	$\sum_{i=1}^{n-1} t_i$
5	**swap** `A[j]` and `A[j` $-$ `1]`	$\sum_{i=1}^{n-1} t_i - 1$
6	`j := j` $-$ `1`	$\sum_{i=1}^{n-1} t_i - 1$
7	`i := i` $+$ `1`	$n-1$

Worst-Case Laufzeit: c_i Anzahl der Prozessorbefehle für Zeile i

$$c_1 + (c_2 + c_3 + c_7)n + (c_4 + c_5 + c_6)\frac{n(n-1)}{2}.$$

Ist das aussagekräftig?

Laufzeit, asymptotische Betrachtung, $\mathcal{O}$-Notation

Laufzeit

Die Laufzeit eines Algorithmus $\mathcal{A}$ auf der Eingabe w ist die Anzahl elementarer Rechenoperationen, die $\mathcal{A}$ gestartet auf w ausführt.

Laufzeit

Die Laufzeit eines Algorithmus $\mathcal{A}$ auf der Eingabe w ist die Anzahl elementarer Rechenoperationen, die $\mathcal{A}$ gestartet auf w ausführt.

- Elementare Rechenoperationen:
  - Vergleiche zweier Zahlen, Addition, Multiplikation, etc.

Laufzeit

Die Laufzeit eines Algorithmus $\mathcal{A}$ auf der Eingabe w ist die Anzahl elementarer Rechenoperationen, die $\mathcal{A}$ gestartet auf w ausführt.

- Elementare Rechenoperationen:
  - Vergleiche zweier Zahlen, Addition, Multiplikation, etc.
  - Anzahl gibt eine recht gute Abschätzung des Rechenaufwands

Laufzeit

Die Laufzeit eines Algorithmus $\mathcal{A}$ auf der Eingabe w ist die Anzahl elementarer Rechenoperationen, die $\mathcal{A}$ gestartet auf w ausführt.

- Elementare Rechenoperationen:
  - Vergleiche zweier Zahlen, Addition, Multiplikation, etc.
  - Anzahl gibt eine recht gute Abschätzung des Rechenaufwands
- Wir messen die Laufzeit in Abhängigkeit von der Länge der Eingabe w, abstrahiert von konkreter Eingabe.

Laufzeit

Die Laufzeit eines Algorithmus $\mathcal{A}$ auf der Eingabe w ist die Anzahl elementarer Rechenoperationen, die $\mathcal{A}$ gestartet auf w ausführt.

- Elementare Rechenoperationen:
  - Vergleiche zweier Zahlen, Addition, Multiplikation, etc.
  - Anzahl gibt eine recht gute Abschätzung des Rechenaufwands
- Wir messen die Laufzeit in Abhängigkeit von der Länge der Eingabe w, abstrahiert von konkreter Eingabe.
- (Worst-Case) Laufzeit als Funktion:

$$f : \mathbb{N} \to \mathbb{R}$$

Eingabelänge

(Maximale) Anzahl Rechenschritte auf Eingaben dieser Länge

Asymptotische Betrachtung, Größenordnung

Uns interessieren nicht die exakten Laufzeiten, sondern eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren u. additive Terme ignoriert.

- Laufzeitanalyse von Algorithmen soll unabhängig sein von verwendetem Computer, Programmiersprache, Fähigkeiten des Programmierers usw.

Asymptotische Betrachtung, Größenordnung

Uns interessieren nicht die exakten Laufzeiten, sondern eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren u. additive Terme ignoriert.

- Laufzeitanalyse von Algorithmen soll unabhängig sein von verwendetem Computer, Programmiersprache, Fähigkeiten des Programmierers usw.
- Ein exaktes Zählen der Operationen auf einem abstrakten Rechnermodell wenig aussagekräftig und häufig schwierig.

Asymptotische Betrachtung, Größenordnung

Uns interessieren nicht die exakten Laufzeiten, sondern eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren u. additive Terme ignoriert.

- Laufzeitanalyse von Algorithmen soll unabhängig sein von verwendetem Computer, Programmiersprache, Fähigkeiten des Programmierers usw.
- Ein exaktes Zählen der Operationen auf einem abstrakten Rechnermodell wenig aussagekräftig und häufig schwierig.
- Ein linearer speed-up (um einen konstanten Faktor) läßt sich z.B. durch einen schnelleren Rechner erreichen.

Asymptotische Betrachtung, Größenordnung

Uns interessieren nicht die exakten Laufzeiten, sondern eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren u. additive Terme ignoriert.

- Laufzeitanalyse von Algorithmen soll unabhängig sein von verwendetem Computer, Programmiersprache, Fähigkeiten des Programmierers usw.
- Ein exaktes Zählen der Operationen auf einem abstrakten Rechnermodell wenig aussagekräftig und häufig schwierig.
- Ein linearer speed-up (um einen konstanten Faktor) läßt sich z.B. durch einen schnelleren Rechner erreichen.

Asymptotische Betrachtung, Größenordnung

Uns interessieren nicht die exakten Laufzeiten, sondern eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren u. additive Terme ignoriert.

- Laufzeitanalyse von Algorithmen soll unabhängig sein von verwendetem Computer, Programmiersprache, Fähigkeiten des Programmierers usw.
- Ein exaktes Zählen der Operationen auf einem abstrakten Rechnermodell wenig aussagekräftig und häufig schwierig.
- Ein linearer speed-up (um einen konstanten Faktor) läßt sich z.B. durch einen schnelleren Rechner erreichen.

  $\Rightarrow$ Wir abstrahieren daher von konkreten Konstanten und unterscheiden bspw. nicht zwischen $3n^2$ und $5n^2$.

Asymptotische Betrachtung, Größenordnung

Uns interessieren nicht die exakten Laufzeiten, sondern eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren u. additive Terme ignoriert.

- Laufzeitanalyse von Algorithmen soll unabhängig sein von verwendetem Computer, Programmiersprache, Fähigkeiten des Programmierers usw.
- Ein exaktes Zählen der Operationen auf einem abstrakten Rechnermodell wenig aussagekräftig und häufig schwierig.
- Ein linearer speed-up (um einen konstanten Faktor) läßt sich z.B. durch einen schnelleren Rechner erreichen.

  $\Rightarrow$ Wir abstrahieren daher von konkreten Konstanten und unterscheiden bspw. nicht zwischen $3n^2$ und $5n^2$.
- Auch Supercomputer rettet "schlechten" Algorithmus nicht. Für große Eingaben gewinnt der schnellere Algorithmus auf dem langsameren Computer.

Asymptotische Betrachtung, Größenordnung

Uns interessieren nicht die exakten Laufzeiten, sondern eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren u. additive Terme ignoriert.

- Laufzeitanalyse von Algorithmen soll unabhängig sein von verwendetem Computer, Programmiersprache, Fähigkeiten des Programmierers usw.
- Ein exaktes Zählen der Operationen auf einem abstrakten Rechnermodell wenig aussagekräftig und häufig schwierig.
- Ein linearer speed-up (um einen konstanten Faktor) läßt sich z.B. durch einen schnelleren Rechner erreichen.

  $\Rightarrow$ Wir abstrahieren daher von konkreten Konstanten und unterscheiden bspw. nicht zwischen $3n^2$ und $5n^2$.
- Auch Supercomputer rettet "schlechten" Algorithmus nicht. Für große Eingaben gewinnt der schnellere Algorithmus auf dem langsameren Computer.

Asymptotische Betrachtung, Größenordnung

Uns interessieren nicht die exakten Laufzeiten, sondern eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren u. additive Terme ignoriert.

- Laufzeitanalyse von Algorithmen soll unabhängig sein von verwendetem Computer, Programmiersprache, Fähigkeiten des Programmierers usw.
- Ein exaktes Zählen der Operationen auf einem abstrakten Rechnermodell wenig aussagekräftig und häufig schwierig.
- Ein linearer speed-up (um einen konstanten Faktor) läßt sich z.B. durch einen schnelleren Rechner erreichen.

  $\Rightarrow$ Wir abstrahieren daher von konkreten Konstanten und unterscheiden bspw. nicht zwischen $3n^2$ und $5n^2$.
- Auch Supercomputer rettet "schlechten" Algorithmus nicht. Für große Eingaben gewinnt der schnellere Algorithmus auf dem langsameren Computer.

  $\Rightarrow$ Wir betrachten das Wachstum für $n \to \infty$.

Asymptotische Betrachtung, Größenordnung

Uns interessieren nicht die exakten Laufzeiten, sondern eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren u. additive Terme ignoriert.

- Laufzeitanalyse von Algorithmen soll unabhängig sein von verwendetem Computer, Programmiersprache, Fähigkeiten des Programmierers usw.
- Ein exaktes Zählen der Operationen auf einem abstrakten Rechnermodell wenig aussagekräftig und häufig schwierig.
- Ein linearer speed-up (um einen konstanten Faktor) läßt sich z.B. durch einen schnelleren Rechner erreichen.

  $\Rightarrow$ Wir abstrahieren daher von konkreten Konstanten und unterscheiden bspw. nicht zwischen $3n^2$ und $5n^2$.
- Auch Supercomputer rettet "schlechten" Algorithmus nicht. Für große Eingaben gewinnt der schnellere Algorithmus auf dem langsameren Computer.

  $\Rightarrow$ Wir betrachten das Wachstum für $n \to \infty$.
- D.h. wir fokussieren auf die Größenordnung des Wachstums. Bei großen n sind Terme niedriger Ordnung unwichtig ($10n^3 + 1000n^2 + 30n + 300$).

Größenordnung des Wachstums

$T(n)$ \ n	2	4	8	32
n^2	4			
$n!$	2			

Größenordnung des Wachstums

$T(n)$ \ n	2	4	8	32
n^2	4	16		
$n!$	2	24		

Größenordnung des Wachstums

$T(n)$ \ n	2	4	8	32
n^2	4	16	64	
$n!$	2	24	40320	

Größenordnung des Wachstums

$T(n)$ \ n	2	4	8	32
n^2	4	16	64	1024
$n!$	2	24	40320	$\approx 2.6 \cdot 10^{35}$

Größenordnung des Wachstums

$T(n)$ \ n	2	4	8	32
n^2	4	16	64	1024
$n!$	2	24	40320	$\approx 2.6 \cdot 10^{35}$

Schlimm? Schnellster Supercomputer mit 415 PetaFLOPS $= 415 \cdot 10^{15}$ FLOPS (floating points per second)

Größenordnung des Wachstums

$T(n)$ \ n	2	4	8	32
n^2	4	16	64	1024
$n!$	2	24	40320	$\approx 2.6 \cdot 10^{35}$

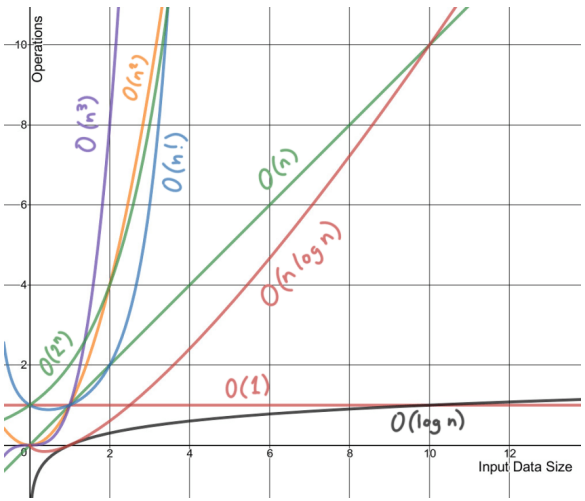
Schlimm? Schnellster Supercomputer mit 415 PetaFLOPS $= 415 \cdot 10^{15}$ FLOPS (floating points per second) braucht bei $n = 32$ etwa $6.3 \cdot 10^{17}$ Jahre.

$\approx$ ein Vielfaches des geschätzten Alters des Universums! ($\approx 1.37 \cdot 10^{10}$ Jahre)

Größenordnung des Wachstums

$T(n)$ \ n	2	4	8	32
n^2	4	16	64	1024
$n!$	2	24	40320	$\approx 2.6 \cdot 10^{35}$

Schlimm? Schnellster Supercomputer mit 415 PetaFLOPS $= 415 \cdot 10^{15}$ FLOPS (floating points per second) braucht bei $n = 32$ etwa $6.3 \cdot 10^{17}$ Jahre.
$\approx$ ein Vielfaches des geschätzten Alters des Universums! ($\approx 1.37 \cdot 10^{10}$ Jahre)

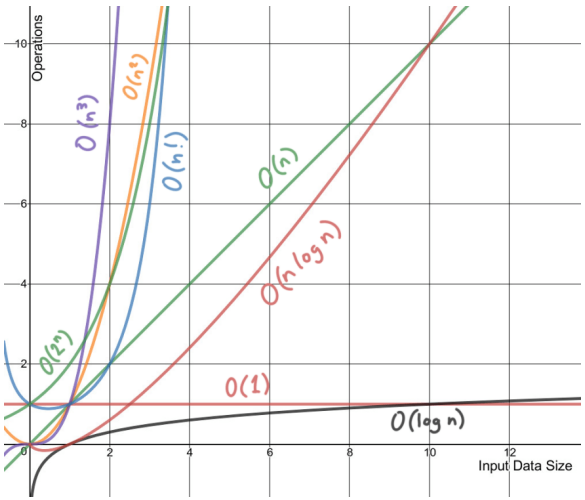


Ziel: Algorithmen deren asymptotische Laufzeit polynomiell in der Eingabegröße ist. Diese nennen wir effizient.

Größenordnung des Wachstums

$T(n)$ \ n	2	4	8	32
n^2	4	16	64	1024
$n!$	2	24	40320	$\approx 2.6 \cdot 10^{35}$

Schlimm? Schnellster Supercomputer mit 415 PetaFLOPS $= 415 \cdot 10^{15}$ FLOPS (floating points per second) braucht bei $n = 32$ etwa $6.3 \cdot 10^{17}$ Jahre.
$\approx$ ein Vielfaches des geschätzten Alters des Universums! ($\approx 1.37 \cdot 10^{10}$ Jahre)



Ziel: Algorithmen deren asymptotische Laufzeit polynomiell in der Eingabegröße ist. Diese nennen wir effizient.

Größenordnung der Laufzeit wird ausgedrückt in der $\mathcal{O}$-Notation, auch Landau-Notation.

$\mathcal{O}$-Notation

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\mathcal{O}(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

$\mathcal{O}$-Notation

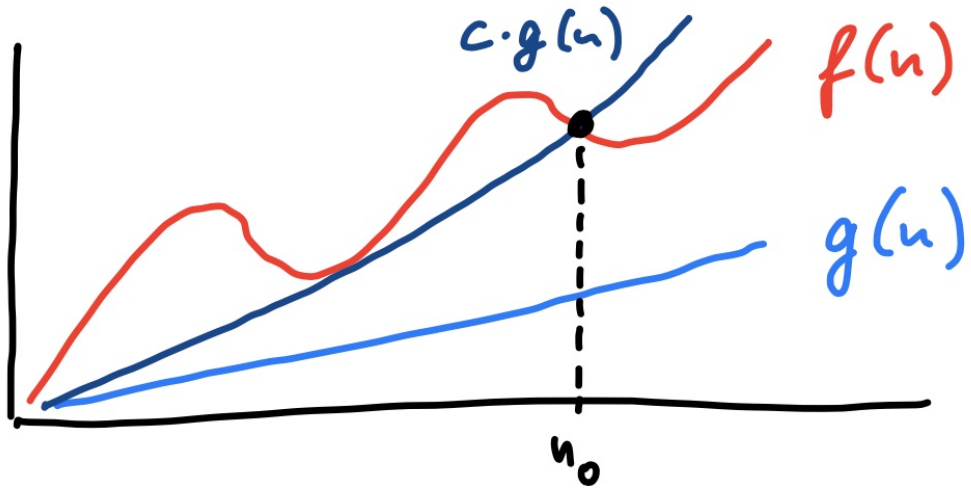
Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\mathcal{O}(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

- $\mathcal{O}(g)$ Menge der Funktionen, die nicht schneller wachsen als g



$\mathcal{O}$-Notation

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\mathcal{O}(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

Äquivalente Charakterisierung

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen. Dann gilt

$$f \in \mathcal{O}(g) \quad \Leftrightarrow \quad \limsup_{n \to \infty} \frac{f(n)}{g(n)} = c$$

für ein $c \in \mathbb{R}_+$.

$\mathcal{O}$-Notation

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\mathcal{O}(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

Äquivalente Charakterisierung

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen. Dann gilt

$$f \in \mathcal{O}(g) \quad \Leftrightarrow \quad \limsup_{n \to \infty} \frac{f(n)}{g(n)} = c$$

für ein $c \in \mathbb{R}_+$.

Groß-$\mathcal{O}$ liefert obere Schranke an die Komplexität einer Funktion.

Limes superior und limes inferior

- Limes superior (inferior) einer Folge $x_1, x_2, \ldots$ reeller Zahlen: größter (kleinster) Häufungspunkt der Folge.

Limes superior und limes inferior

- Limes superior (inferior) einer Folge $x_1, x_2, \ldots$ reeller Zahlen: größter (kleinster) Häufungspunkt der Folge.
- Häufungspunkt: Ein Punkt ist Häufungspunkt, falls in jeder noch so kleinen Umgebung des Punktes unendlich viele Folgenglieder liegen.

Limes superior und limes inferior

- Limes superior (inferior) einer Folge $x_1, x_2, \ldots$ reeller Zahlen: größter (kleinster) Häufungspunkt der Folge.
- Häufungspunkt: Ein Punkt ist Häufungspunkt, falls in jeder noch so kleinen Umgebung des Punktes unendlich viele Folgenglieder liegen. Wir haben einen Häufungspunkt ∞ falls beliebig große Elemente in der Folge auftreten.

Limes superior und limes inferior

- Limes superior (inferior) einer Folge $x_1, x_2, \ldots$ reeller Zahlen: größter (kleinster) Häufungspunkt der Folge.
- Häufungspunkt: Ein Punkt ist Häufungspunkt, falls in jeder noch so kleinen Umgebung des Punktes unendlich viele Folgenglieder liegen. Wir haben einen Häufungspunkt ∞ falls beliebig große Elemente in der Folge auftreten.
- Partieller Ersatz für den Limes, falls dieser nicht existiert.

Limes superior und limes inferior

- Limes superior (inferior) einer Folge $x_1, x_2, \ldots$ reeller Zahlen: größter (kleinster) Häufungspunkt der Folge.
- Häufungspunkt: Ein Punkt ist Häufungspunkt, falls in jeder noch so kleinen Umgebung des Punktes unendlich viele Folgenglieder liegen. Wir haben einen Häufungspunkt ∞ falls beliebig große Elemente in der Folge auftreten.
- Partieller Ersatz für den Limes, falls dieser nicht existiert.

Beispiel:

$$x_n = \begin{cases} \frac{1}{n} & \text{falls } n \text{ ungerade} \\ 1 - \frac{1}{n} & \text{falls } n \text{ gerade} \end{cases}$$

Dann sind 0 und 1 Häufungspunkte und $\limsup_{n\to\infty} x_n = 1$.

$\mathcal{O}$-Notation - Beispiele

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\mathcal{O}(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

- $100n \in \mathcal{O}(n)$

$\mathcal{O}$-Notation - Beispiele

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\mathcal{O}(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

- $100n \in \mathcal{O}(n)$

  Wähle $c = 100$ und $n_0 = 1$.

  Dann gilt $100n \leq cn$ für alle $n \geq n_0$.

$\mathcal{O}$-Notation - Beispiele

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\mathcal{O}(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

- $100n \in \mathcal{O}(n)$

  Wähle $c = 100$ und $n_0 = 1$.

  Dann gilt $100n \leq cn$ für alle $n \geq n_0$.

- $10n^3 + 1000n \in \mathcal{O}(n^3)$

$\mathcal{O}$-Notation - Beispiele

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\mathcal{O}(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

- $100n \in \mathcal{O}(n)$

  Wähle $c = 100$ und $n_0 = 1$.

  Dann gilt $100n \leq cn$ für alle $n \geq n_0$.

- $10n^3 + 1000n \in \mathcal{O}(n^3)$

  Wähle $c = 20$ und $n_0 = 10$.

  Dann gilt $10n^3 + 1000n \leq 10n^3 + 10n^3 = cn^3$ für alle $n \geq n_0$.

$\mathcal{O}$-Notation - Beispiele

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\mathcal{O}(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

- $2n^2 + 10n + 5 \notin \mathcal{O}(n)$

$\mathcal{O}$-Notation - Beispiele

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\mathcal{O}(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

- $2n^2 + 10n + 5 \notin \mathcal{O}(n)$

  Angenommen es gibt $c \in \mathbb{R}_+$ und $n_0 \in \mathbb{N}$ sodass

$$2n^2 + 10n + 5 \leq cn$$

  für alle $n \geq n_0$.

$\mathcal{O}$-Notation - Beispiele

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\mathcal{O}(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

- $2n^2 + 10n + 5 \notin \mathcal{O}(n)$

  Angenommen es gibt $c \in \mathbb{R}_+$ und $n_0 \in \mathbb{N}$ sodass

$$2n^2 + 10n + 5 \leq cn$$

  für alle $n \geq n_0$.
  Betrachte $n = \max(\lceil c \rceil, n_0)$.

$\mathcal{O}$-Notation - Beispiele

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\mathcal{O}(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

- $2n^2 + 10n + 5 \notin \mathcal{O}(n)$

  Angenommen es gibt $c \in \mathbb{R}_+$ und $n_0 \in \mathbb{N}$ sodass

$$2n^2 + 10n + 5 \leq cn$$

  für alle $n \geq n_0$.
  Betrachte $n = \max(\lceil c \rceil, n_0)$.
  Dann ist

$$2n^2 + 10n + 5 \geq 2cn + 10n + 5 > cn,$$

  ein Widerspruch.

Laufzeitanalyse Insertion Sort

```
1 i := 1                                                    1
2 while i < length(A) do                                    n
3     j := i                                                n − 1
4     while j > 0 and A[j − 1] > A[j] do                    ∑_{i=1}^{n−1} t_i
5         swap A[j] and A[j − 1]                            ∑_{i=1}^{n−1} t_i − 1
6         j := j − 1                                        ∑_{i=1}^{n−1} t_i − 1
7     i := i + 1                                            n − 1
```

Worst-Case Laufzeit: c_i Anzahl der Prozessorbefehle für Zeile i

$$c_1 + (c_2 + c_3 + c_7)n + (c_4 + c_5 + c_6)\frac{n(n-1)}{2}.$$

Worst-Case Laufzeit: $\mathcal{O}(n^2)$ (quadratisch)

Rechnen mit $\mathcal{O}$-Notation

Definition

Seien $f, g \colon \mathbb{N} \to \mathbb{R}$ Funktionen.

Rechnen mit $\mathcal{O}$-Notation

Definition

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen. Definiere

- $(f + g)(n) = f(n) + g(n)$ für alle $n \in \mathbb{N}$.

Rechnen mit $\mathcal{O}$-Notation

Definition

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen. Definiere

- $(f + g)(n) = f(n) + g(n)$ für alle $n \in \mathbb{N}$.
- $(f \cdot g)(n) = f(n) \cdot g(n)$ für alle $n \in \mathbb{N}$.

Rechnen mit $\mathcal{O}$-Notation

Definition

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen. Definiere

- $(f + g)(n) = f(n) + g(n)$ für alle $n \in \mathbb{N}$.
- $(f \cdot g)(n) = f(n) \cdot g(n)$ für alle $n \in \mathbb{N}$.

Seien $\mathcal{A}, \mathcal{B}$ Klassen von Funktionen.

Rechnen mit $\mathcal{O}$-Notation

Definition

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen. Definiere

- $(f + g)(n) = f(n) + g(n)$ für alle $n \in \mathbb{N}$.
- $(f \cdot g)(n) = f(n) \cdot g(n)$ für alle $n \in \mathbb{N}$.

Seien $\mathcal{A}, \mathcal{B}$ Klassen von Funktionen. Definiere

- $\mathcal{A} + \mathcal{B} = \{f + g \ : \ f \in \mathcal{A}, g \in \mathcal{B}\}$.

Rechnen mit $\mathcal{O}$-Notation

Definition

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen. Definiere

- $(f + g)(n) = f(n) + g(n)$ für alle $n \in \mathbb{N}$.
- $(f \cdot g)(n) = f(n) \cdot g(n)$ für alle $n \in \mathbb{N}$.

Seien $\mathcal{A}, \mathcal{B}$ Klassen von Funktionen. Definiere

- $\mathcal{A} + \mathcal{B} = \{f + g \ : \ f \in \mathcal{A}, g \in \mathcal{B}\}$.
- $\mathcal{A} \cdot \mathcal{B} = \{f \cdot g \ : \ f \in \mathcal{A}, g \in \mathcal{B}\}$.

Rechnen mit $\mathcal{O}$-Notation

Satz

Seien $f, g\colon \mathbb{N} \to \mathbb{R}$ Funktionen, und $f(n) \geq 0$, $g(n) \geq 1$.

Rechnen mit $\mathcal{O}$-Notation

Satz

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen, und $f(n) \geq 0$, $g(n) \geq 1$. Mit 1 bezeichne die Funktion, die jedes n auf 1 abbildet.

Rechnen mit $\mathcal{O}$-Notation

Satz

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen, und $f(n) \geq 0$, $g(n) \geq 1$. Mit 1 bezeichne die Funktion, die jedes n auf 1 abbildet. Es gilt

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$.

Rechnen mit $\mathcal{O}$-Notation

Satz

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen, und $f(n) \geq 0$, $g(n) \geq 1$. Mit 1 bezeichne die Funktion, die jedes n auf 1 abbildet. Es gilt

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$.

(b) $\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f + g)$.

Rechnen mit $\mathcal{O}$-Notation

Satz

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen, und $f(n) \geq 0$, $g(n) \geq 1$. Mit 1 bezeichne die Funktion, die jedes n auf 1 abbildet. Es gilt

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$.

(b) $\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f + g)$.

(b) $\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$.

Rechnen mit $\mathcal{O}$-Notation

Satz

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen, und $f(n) \geq 0$, $g(n) \geq 1$. Mit 1 bezeichne die Funktion, die jedes n auf 1 abbildet. Es gilt

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$.

(b) $\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f + g)$.

(b) $\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$.

Beweis.

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$:

Rechnen mit $\mathcal{O}$-Notation

Satz

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen, und $f(n) \geq 0$, $g(n) \geq 1$. Mit 1 bezeichne die Funktion, die jedes n auf 1 abbildet. Es gilt

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$.

(b) $\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f + g)$.

(b) $\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$.

Beweis.

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$: Sei $h \in \mathcal{O}(1)$.

Rechnen mit $\mathcal{O}$-Notation

Satz

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen, und $f(n) \geq 0$, $g(n) \geq 1$. Mit 1 bezeichne die Funktion, die jedes n auf 1 abbildet. Es gilt

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$.

(b) $\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f + g)$.

(b) $\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$.

Beweis.

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$: Sei $h \in \mathcal{O}(1)$.

- Dann existieren c und n_0, so dass $h(n) \leq c$ für alle $n \geq n_0$.

Rechnen mit $\mathcal{O}$-Notation

Satz

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen, und $f(n) \geq 0$, $g(n) \geq 1$. Mit 1 bezeichne die Funktion, die jedes n auf 1 abbildet. Es gilt

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$.

(b) $\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f + g)$.

(b) $\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$.

Beweis.

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$: Sei $h \in \mathcal{O}(1)$.

- Dann existieren c und n_0, so dass $h(n) \leq c$ für alle $n \geq n_0$.
- Sei $d = \max\{h(1), \ldots, h(n_0), c\}$. Dann gilt $h(n) \leq d$ f. a. $n \in \mathbb{N}$.

Rechnen mit $\mathcal{O}$-Notation

Satz

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen, und $f(n) \geq 0$, $g(n) \geq 1$. Mit 1 bezeichne die Funktion, die jedes n auf 1 abbildet. Es gilt

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$.

(b) $\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f + g)$.

(b) $\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$.

Beweis.

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$: Sei $h \in \mathcal{O}(1)$.

- Dann existieren c und n_0, so dass $h(n) \leq c$ für alle $n \geq n_0$.
- Sei $d = \max\{h(1), \ldots, h(n_0), c\}$. Dann gilt $h(n) \leq d$ f. a. $n \in \mathbb{N}$.
  - $h \in \mathcal{O}(1)$ heißt h ist absolut beschränkt.

Rechnen mit $\mathcal{O}$-Notation

Satz

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen, und $f(n) \geq 0$, $g(n) \geq 1$. Mit 1 bezeichne die Funktion, die jedes n auf 1 abbildet. Es gilt

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$.

(b) $\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f + g)$.

(b) $\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$.

Beweis.

(a) $\mathcal{O}(1) \subseteq \mathcal{O}(g)$: Sei $h \in \mathcal{O}(1)$.

- Dann existieren c und n_0, so dass $h(n) \leq c$ für alle $n \geq n_0$.
- Sei $d = \max\{h(1), \ldots, h(n_0), c\}$. Dann gilt $h(n) \leq d$ f. a. $n \in \mathbb{N}$.
  - $h \in \mathcal{O}(1)$ heißt h ist absolut beschränkt.
- Da $g(n) \geq 1$ für alle $n \in \mathbb{N}$, gilt $h(n) \leq d \cdot g(n)$, also $h \in \mathcal{O}(g)$.

Rechnen mit $\mathcal{O}$-Notation

(b) $\mathcal{O}(f) + \mathcal{O}(g) \subseteq \mathcal{O}(f + g)$:

Rechnen mit $\mathcal{O}$-Notation

(b) $\mathcal{O}(f) + \mathcal{O}(g) \subseteq \mathcal{O}(f + g)$:

- Sei $h \in \mathcal{O}(f) + \mathcal{O}(g)$, etwa $h = f' + g'$ für $f' \in \mathcal{O}(f)$, $g' \in \mathcal{O}(g)$.

Rechnen mit $\mathcal{O}$-Notation

(b) $\mathcal{O}(f) + \mathcal{O}(g) \subseteq \mathcal{O}(f + g)$:

- Sei $h \in \mathcal{O}(f) + \mathcal{O}(g)$, etwa $h = f' + g'$ für $f' \in \mathcal{O}(f)$, $g' \in \mathcal{O}(g)$.
- Es ex. c_f, c_g und n_f, n_g, so dass $f'(n) \leq c_f f(n)$ und $g'(n) \leq c_g g(n)$ für alle $n \geq n_f$ bzw. $n \geq n_g$.

Rechnen mit $\mathcal{O}$-Notation

(b) $\mathcal{O}(f) + \mathcal{O}(g) \subseteq \mathcal{O}(f + g)$:

- Sei $h \in \mathcal{O}(f) + \mathcal{O}(g)$, etwa $h = f' + g'$ für $f' \in \mathcal{O}(f)$, $g' \in \mathcal{O}(g)$.
- Es ex. c_f, c_g und n_f, n_g, so dass $f'(n) \leq c_f f(n)$ und $g'(n) \leq c_g g(n)$ für alle $n \geq n_f$ bzw. $n \geq n_g$.
- Sei $c = c_f + c_g$ und $n_0 = \max\{n_f, n_g\}$.

Rechnen mit $\mathcal{O}$-Notation

(b) $\mathcal{O}(f) + \mathcal{O}(g) \subseteq \mathcal{O}(f + g)$:

- Sei $h \in \mathcal{O}(f) + \mathcal{O}(g)$, etwa $h = f' + g'$ für $f' \in \mathcal{O}(f)$, $g' \in \mathcal{O}(g)$.
- Es ex. c_f, c_g und n_f, n_g, so dass $f'(n) \leq c_f f(n)$ und $g'(n) \leq c_g g(n)$ für alle $n \geq n_f$ bzw. $n \geq n_g$.
- Sei $c = c_f + c_g$ und $n_0 = \max\{n_f, n_g\}$.
- Dann gilt für alle $n \geq n_0$, dass

Rechnen mit $\mathcal{O}$-Notation

(b) $\mathcal{O}(f) + \mathcal{O}(g) \subseteq \mathcal{O}(f+g)$:

- Sei $h \in \mathcal{O}(f) + \mathcal{O}(g)$, etwa $h = f' + g'$ für $f' \in \mathcal{O}(f)$, $g' \in \mathcal{O}(g)$.
- Es ex. c_f, c_g und n_f, n_g, so dass $f'(n) \leq c_f f(n)$ und $g'(n) \leq c_g g(n)$ für alle $n \geq n_f$ bzw. $n \geq n_g$.
- Sei $c = c_f + c_g$ und $n_0 = \max\{n_f, n_g\}$.
- Dann gilt für alle $n \geq n_0$, dass

$$\begin{aligned} h(n) &\leq f'(n) + g'(n) \leq c_f f(n) + c_g g(n) \\ &\leq (c_f + c_g) f(n) + (c_f + c_g) g(n) \\ &\leq (c_f + c_g)(f(n) + g(n)), \end{aligned}$$

Rechnen mit $\mathcal{O}$-Notation

(b) $\mathcal{O}(f) + \mathcal{O}(g) \subseteq \mathcal{O}(f+g)$:

- Sei $h \in \mathcal{O}(f) + \mathcal{O}(g)$, etwa $h = f' + g'$ für $f' \in \mathcal{O}(f)$, $g' \in \mathcal{O}(g)$.
- Es ex. c_f, c_g und n_f, n_g, so dass $f'(n) \leq c_f f(n)$ und $g'(n) \leq c_g g(n)$ für alle $n \geq n_f$ bzw. $n \geq n_g$.
- Sei $c = c_f + c_g$ und $n_0 = \max\{n_f, n_g\}$.
- Dann gilt für alle $n \geq n_0$, dass

$$\begin{aligned} h(n) &\leq f'(n) + g'(n) \leq c_f f(n) + c_g g(n) \\ &\leq (c_f + c_g) f(n) + (c_f + c_g) g(n) \\ &\leq (c_f + c_g)(f(n) + g(n)), \end{aligned}$$

- Also $h \in \mathcal{O}(f+g)$.

Rechnen mit $\mathcal{O}$-Notation

(b) $\mathcal{O}(f) + \mathcal{O}(g) \subseteq \mathcal{O}(f+g)$:

- Sei $h \in \mathcal{O}(f) + \mathcal{O}(g)$, etwa $h = f' + g'$ für $f' \in \mathcal{O}(f)$, $g' \in \mathcal{O}(g)$.
- Es ex. c_f, c_g und n_f, n_g, so dass $f'(n) \leq c_f f(n)$ und $g'(n) \leq c_g g(n)$ für alle $n \geq n_f$ bzw. $n \geq n_g$.
- Sei $c = c_f + c_g$ und $n_0 = \max\{n_f, n_g\}$.
- Dann gilt für alle $n \geq n_0$, dass

$$\begin{aligned} h(n) &\leq f'(n) + g'(n) \leq c_f f(n) + c_g g(n) \\ &\leq (c_f + c_g) f(n) + (c_f + c_g) g(n) \\ &\leq (c_f + c_g)(f(n) + g(n)), \end{aligned}$$

- Also $h \in \mathcal{O}(f+g)$.

(b) $\mathcal{O}(f+g) \subseteq \mathcal{O}(f) + \mathcal{O}(g)$: Wird analog bewiesen.

(c) Wird analog bewiesen. □

Rechnen mit $\mathcal{O}$-Notation: Transitivität

Satz

Seien $f, g, h\colon \mathbb{N} \to \mathbb{R}$ Funktionen.

Rechnen mit $\mathcal{O}$-Notation: Transitivität

Satz

Seien $f, g, h\colon \mathbb{N} \to \mathbb{R}$ Funktionen. Wenn

$$f \in \mathcal{O}(g) \quad \text{und} \quad g \in \mathcal{O}(h),$$

Rechnen mit $\mathcal{O}$-Notation: Transitivität

Satz

Seien $f, g, h\colon \mathbb{N} \to \mathbb{R}$ Funktionen. Wenn

$$f \in \mathcal{O}(g) \quad \text{und} \quad g \in \mathcal{O}(h),$$

dann

$$f \in \mathcal{O}(h).$$

$\mathcal{O}$-Notation: Schreibweisen in der Literatur

In der Literatur finden sich oft Schreibweisen

- $f = \mathcal{O}(g)$

$\mathcal{O}$-Notation: Schreibweisen in der Literatur

In der Literatur finden sich oft Schreibweisen

- $f = \mathcal{O}(g)$ soll heißen $f \in \mathcal{O}(g)$.

$\mathcal{O}$-Notation: Schreibweisen in der Literatur

In der Literatur finden sich oft Schreibweisen

- $f = \mathcal{O}(g)$ soll heißen $f \in \mathcal{O}(g)$.
- $f = g + \mathcal{O}(h)$

$\mathcal{O}$-Notation: Schreibweisen in der Literatur

In der Literatur finden sich oft Schreibweisen

- $f = \mathcal{O}(g)$ soll heißen $f \in \mathcal{O}(g)$.
- $f = g + \mathcal{O}(h)$ soll heißen $f - g \in \mathcal{O}(h)$.

$\mathcal{O}$-Notation: Schreibweisen in der Literatur

In der Literatur finden sich oft Schreibweisen

- $f = \mathcal{O}(g)$ soll heißen $f \in \mathcal{O}(g)$.
- $f = g + \mathcal{O}(h)$ soll heißen $f - g \in \mathcal{O}(h)$.
- $\mathcal{O}(f) = \mathcal{O}(g)$

$\mathcal{O}$-Notation: Schreibweisen in der Literatur

In der Literatur finden sich oft Schreibweisen

- $f = \mathcal{O}(g)$ soll heißen $f \in \mathcal{O}(g)$.
- $f = g + \mathcal{O}(h)$ soll heißen $f - g \in \mathcal{O}(h)$.
- $\mathcal{O}(f) = \mathcal{O}(g)$ kann $\mathcal{O}(f) \subseteq \mathcal{O}(g)$ heißen.

$\mathcal{O}$-Notation: Schreibweisen in der Literatur

In der Literatur finden sich oft Schreibweisen

- $f = \mathcal{O}(g)$ soll heißen $f \in \mathcal{O}(g)$.
- $f = g + \mathcal{O}(h)$ soll heißen $f - g \in \mathcal{O}(h)$.
- $\mathcal{O}(f) = \mathcal{O}(g)$ kann $\mathcal{O}(f) \subseteq \mathcal{O}(g)$ heißen.

Diese Konventionen sind oft bequem, z.B.

- $2n^2 + 3n + 1 = 2n^2 + \mathcal{O}(n) = \mathcal{O}(n^2)$.

$\mathcal{O}$-Notation: Schreibweisen in der Literatur

In der Literatur finden sich oft Schreibweisen

- $f = \mathcal{O}(g)$ soll heißen $f \in \mathcal{O}(g)$.
- $f = g + \mathcal{O}(h)$ soll heißen $f - g \in \mathcal{O}(h)$.
- $\mathcal{O}(f) = \mathcal{O}(g)$ kann $\mathcal{O}(f) \subseteq \mathcal{O}(g)$ heißen.

Diese Konventionen sind oft bequem, z.B.

- $2n^2 + 3n + 1 = 2n^2 + \mathcal{O}(n) = \mathcal{O}(n^2)$.

ACHTUNG: Man muss bei diesen Schreibweisen aufpassen, z.B., es gilt $\mathcal{O}(n) = \mathcal{O}(n^2)$, aber es gilt NICHT $\mathcal{O}(n^2) = \mathcal{O}(n)$.

Ω-Notation

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\Omega(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists\, c \in \mathbb{R}_+ : n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \geq c \cdot g(n).$$

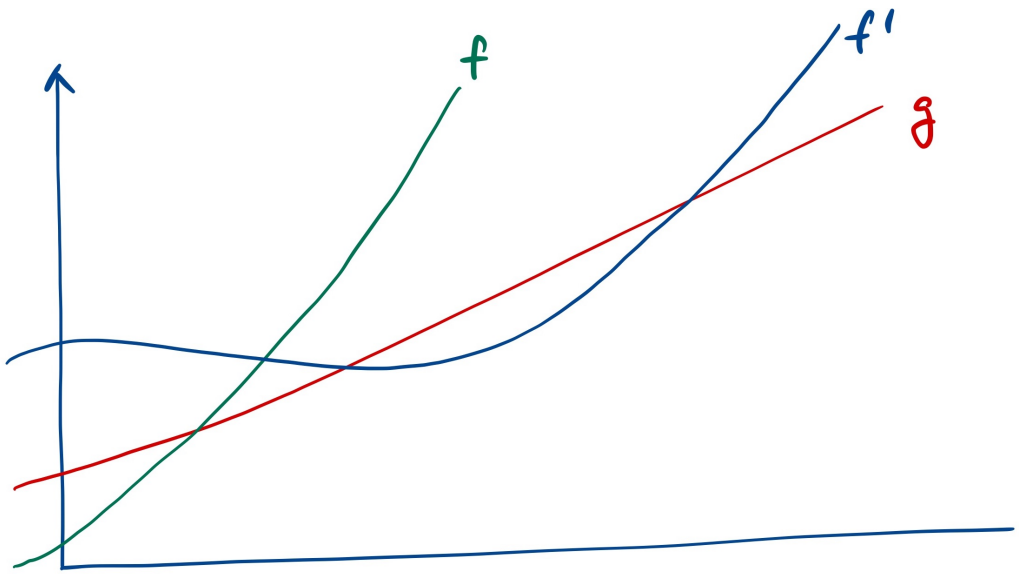
Ω-Notation

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\Omega(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists\, c \in \mathbb{R}_+ : n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \geq c \cdot g(n).$$



Ω-Notation

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\Omega(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists\, c \in \mathbb{R}_+ : n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \geq c \cdot g(n).$$

Äquivalente Charakterisierung

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen. Dann gilt

$$f \in \Omega(g) \quad \Leftrightarrow \quad \liminf_{n \to \infty} \frac{f(n)}{g(n)} > 0.$$

Groß-Ω liefert untere Schranke an die Komplexität einer Funktion.
Große untere Schranken sind nützlich! $\Omega(n^2)$ ist stärker als $\Omega(n)$.

Ω-Notation: Beispiele

Definition

Sei $g : \mathbb{N} \to \mathbb{R}$ eine Funktion.

- Definiere $\Omega(g)$ als die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit

$$\exists\, c \in \mathbb{R}_+ : n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \geq c \cdot g(n).$$

Sei $f(n) = 20 \cdot n^2 + 100 \cdot n + 10$. Dann gilt:

- $f(n) \in \Omega(n)$
- $f(n) \in \Omega(n^2)$
- $f(n) \notin \Omega(n^3)$

Weitere $\mathcal{O}$-Notation, Landau-Notation

- $O(g) := \{f : \mathbb{N} \to \mathbb{R} \mid \exists\, c \in \mathbb{R}_+ : n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$
  (Funktionen die asymptotisch nicht schneller als g wachsen)

Weitere $\mathcal{O}$-Notation, Landau-Notation

- $O(g) := \{f : \mathbb{N} \to \mathbb{R} \mid \exists\, c \in \mathbb{R}_+ : n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$
  (Funktionen die asymptotisch nicht schneller als g wachsen)
- $\Omega(g) := \{f : \mathbb{N} \to \mathbb{R} \mid \exists\, c \in \mathbb{R}_+ : n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$
  (Funktionen die asymptotisch nicht langsamer als g wachsen)

Weitere $\mathcal{O}$-Notation, Landau-Notation

- $O(g) := \{f : \mathbb{N} \to \mathbb{R} \mid \exists\, c \in \mathbb{R}_+ : n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$
  (Funktionen die asymptotisch nicht schneller als g wachsen)
- $\Omega(g) := \{f : \mathbb{N} \to \mathbb{R} \mid \exists\, c \in \mathbb{R}_+ : n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$
  (Funktionen die asymptotisch nicht langsamer als g wachsen)
- $\Theta(g) := O(g) \cap \Omega(g)$
  (Funktionen die asymptotisch das gleiche Wachstum wie g haben)

Weitere $\mathcal{O}$-Notation, Landau-Notation

- $O(g) := \{f : \mathbb{N} \to \mathbb{R} \mid \exists\, c \in \mathbb{R}_+ : n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$
  (Funktionen die asymptotisch nicht schneller als g wachsen)
- $\Omega(g) := \{f : \mathbb{N} \to \mathbb{R} \mid \exists\, c \in \mathbb{R}_+ : n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$
  (Funktionen die asymptotisch nicht langsamer als g wachsen)
- $\Theta(g) := O(g) \cap \Omega(g)$
  (Funktionen die asymptotisch das gleiche Wachstum wie g haben)
- $o(g) := O(g) \backslash \Omega(g) = \{f \mid \forall\, c \in \mathbb{R}_+ : \exists\, n_0 \in \mathbb{N} : \forall\, n \geq n_0 : f(n) \leq c \cdot g(n)\}$
  (Funktionen die asymptotisch langsamer als g wachsen)

Weitere $\mathcal{O}$-Notation, Landau-Notation

- $O(g) := \{f : \mathbb{N} \to \mathbb{R} \mid \exists\, c \in \mathbb{R}_+ : n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$
  (Funktionen die asymptotisch nicht schneller als g wachsen)
- $\Omega(g) := \{f : \mathbb{N} \to \mathbb{R} \mid \exists\, c \in \mathbb{R}_+ : n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$
  (Funktionen die asymptotisch nicht langsamer als g wachsen)
- $\Theta(g) := O(g) \cap \Omega(g)$
  (Funktionen die asymptotisch das gleiche Wachstum wie g haben)
- $o(g) := O(g) \backslash \Omega(g) = \{f \mid \forall\, c \in \mathbb{R}_+ : \exists\, n_0 \in \mathbb{N} : \forall\, n \geq n_0 : f(n) \leq c \cdot g(n)\}$
  (Funktionen die asymptotisch langsamer als g wachsen)
- $\omega(g) := \Omega(g) \backslash O(g) = \{f \mid \forall\, c \in \mathbb{R}_+ : \exists\, n_0 \in \mathbb{N} : \forall\, n \geq n_0 : f(n) \geq c \cdot g(n)\}$
  (Funktionen die asymptotisch schneller als g wachsen)

Weitere $\mathcal{O}$-Notation, Landau-Notation

- $f \in o(g)$: f asymptotisch gegenüber g vernachlässigbar.

$$\limsup_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Weitere $\mathcal{O}$-Notation, Landau-Notation

- $f \in o(g)$: f asymptotisch gegenüber g vernachlässigbar.

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

- $f \in \mathcal{O}(g)$: f ist asymptotisch durch g beschränkt.

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$$

Weitere $\mathcal{O}$-Notation, Landau-Notation

- $f \in \Omega(g)$: f ist asymptotisch von unten durch g beschränkt, $g \in \mathcal{O}(f)$.

$$\liminf_{n \to \infty} \frac{f(n)}{g(n)} > 0.$$

Weitere $\mathcal{O}$-Notation, Landau-Notation

- $f \in \Omega(g)$: f ist asymptotisch von unten durch g beschränkt, $g \in \mathcal{O}(f)$.

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0.$$

- $f \in \omega(g)$: f dominiert g asymptotisch, $g \in o(f)$.

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$

Weitere $\mathcal{O}$-Notation, Landau-Notation

- $f \in \Omega(g)$: f ist asymptotisch von unten durch g beschränkt, $g \in \mathcal{O}(f)$.

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0.$$

- $f \in \omega(g)$: f dominiert g asymptotisch, $g \in o(f)$.

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$

- $f \in \Theta(g)$: f ist asymptotisch scharf beschränkt durch g, sowohl $f \in \mathcal{O}(g)$ als auch $f \in \Omega(g)$.

$\mathcal{O}$-Notation - Zusammenfassung

Die Laufzeit von Algorithmen wird fast ausschließlich in der $\mathcal{O}$-Notation analysiert.

$\mathcal{O}$-Notation - Zusammenfassung

Die Laufzeit von Algorithmen wird fast ausschließlich in der $\mathcal{O}$-Notation analysiert.

Vorteile:

- kompakte und einfache Darstellung der Laufzeit

$\mathcal{O}$-Notation - Zusammenfassung

Die Laufzeit von Algorithmen wird fast ausschließlich in der $\mathcal{O}$-Notation analysiert.

Vorteile:

- kompakte und einfache Darstellung der Laufzeit
- einfacher Vergleich von Laufzeiten

$\mathcal{O}$-Notation - Zusammenfassung

Die Laufzeit von Algorithmen wird fast ausschließlich in der $\mathcal{O}$-Notation analysiert.

Vorteile:

- kompakte und einfache Darstellung der Laufzeit
- einfacher Vergleich von Laufzeiten
- unabhänig von der genauen Definition eines Rechenschritts

$\mathcal{O}$-Notation - Zusammenfassung

Die Laufzeit von Algorithmen wird fast ausschließlich in der $\mathcal{O}$-Notation analysiert.

Vorteile:

- kompakte und einfache Darstellung der Laufzeit
- einfacher Vergleich von Laufzeiten
- unabhänig von der genauen Definition eines Rechenschritts

Beachte:

- Wir verlieren Informationen über die Laufzeit

$\mathcal{O}$-Notation - Zusammenfassung

Die Laufzeit von Algorithmen wird fast ausschließlich in der $\mathcal{O}$-Notation analysiert.

Vorteile:

- kompakte und einfache Darstellung der Laufzeit
- einfacher Vergleich von Laufzeiten
- unabhänig von der genauen Definition eines Rechenschritts

Beachte:

- Wir verlieren Informationen über die Laufzeit
  - Algorithmus $\mathcal{A}_1$ mit Laufzeit $10n + 5 \in \mathcal{O}(n)$
  - Algorithmus $\mathcal{A}_2$ mit Laufzeit $10^9 n + 10^6 \in \mathcal{O}(n)$
  - Beide Algorithmen haben Laufzeit $\mathcal{O}(n)$ in der $\mathcal{O}$-Notation, aber in der Praxis ist Algorithmus $\mathcal{A}_1$ klar besser

Algorithmendesign-Prinzip: Divide-and-Conquer

Divide-and-Conquer (Teile-und-Herrsche)

Divide-and-Conquer (Teile-und-Herrsche) Algorithmen teilen das Problem in mehrere Teilprobleme, die dem Ausgangsproblem ähneln, aber kleiner sind. Sie lösen die Teilprobleme rekursiv und kombinieren sie dann zu einer Lösung des Originalproblems.

Divide-and-Conquer (Teile-und-Herrsche)

Divide-and-Conquer (Teile-und-Herrsche) Algorithmen teilen das Problem in mehrere Teilprobleme, die dem Ausgangsproblem ähneln, aber kleiner sind. Sie lösen die Teilprobleme rekursiv und kombinieren sie dann zu einer Lösung des Originalproblems.

- Divide: Teile eine Probleminstanz in kleinere Teilinstanzen auf.

Divide-and-Conquer (Teile-und-Herrsche)

Divide-and-Conquer (Teile-und-Herrsche) Algorithmen teilen das Problem in mehrere Teilprobleme, die dem Ausgangsproblem ähneln, aber kleiner sind. Sie lösen die Teilprobleme rekursiv und kombinieren sie dann zu einer Lösung des Originalproblems.

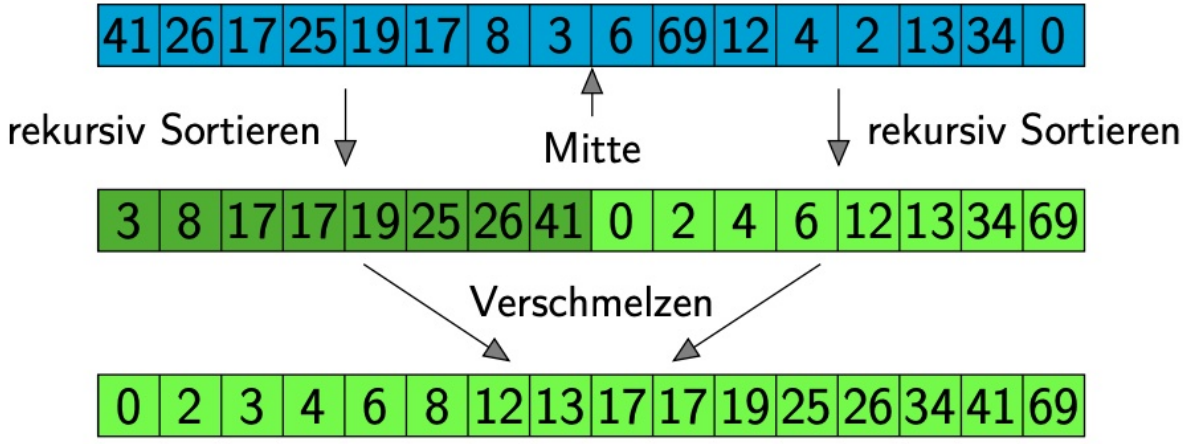
- Divide: Teile eine Probleminstanz in kleinere Teilinstanzen auf.
- Conquer: Löse die Teilinstanzen rekursiv, bis sie so klein sind, dass sie triviale Lösungen erlauben

Divide-and-Conquer (Teile-und-Herrsche)

Divide-and-Conquer (Teile-und-Herrsche) Algorithmen teilen das Problem in mehrere Teilprobleme, die dem Ausgangsproblem ähneln, aber kleiner sind. Sie lösen die Teilprobleme rekursiv und kombinieren sie dann zu einer Lösung des Originalproblems.

- Divide: Teile eine Probleminstanz in kleinere Teilinstanzen auf.
- Conquer: Löse die Teilinstanzen rekursiv, bis sie so klein sind, dass sie triviale Lösungen erlauben
- Combine: Kombiniere die Lösungen der Teilinstanzen zu einer Lösung des Gesamtproblems.

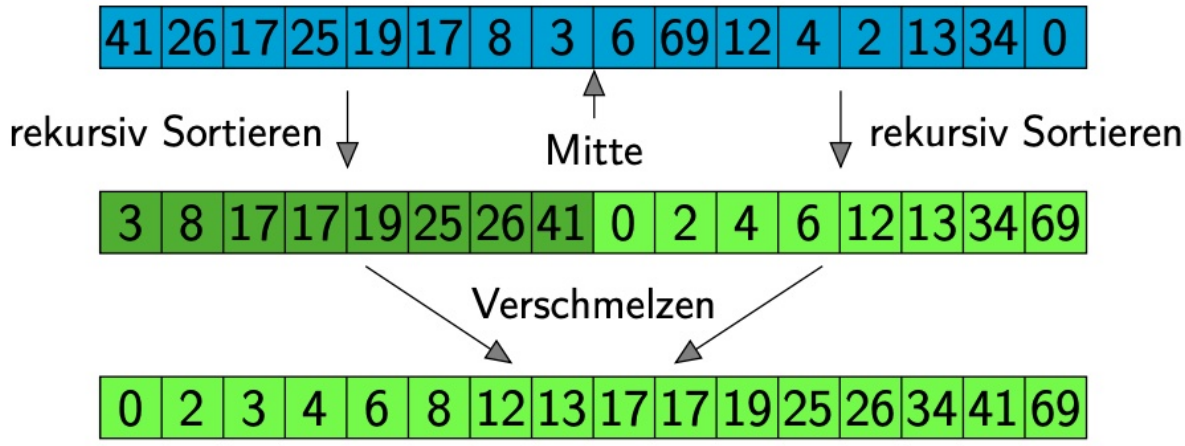
Divide-and-Conquer: Mergesort



© G. Woeginger

Mergesort Strategie:

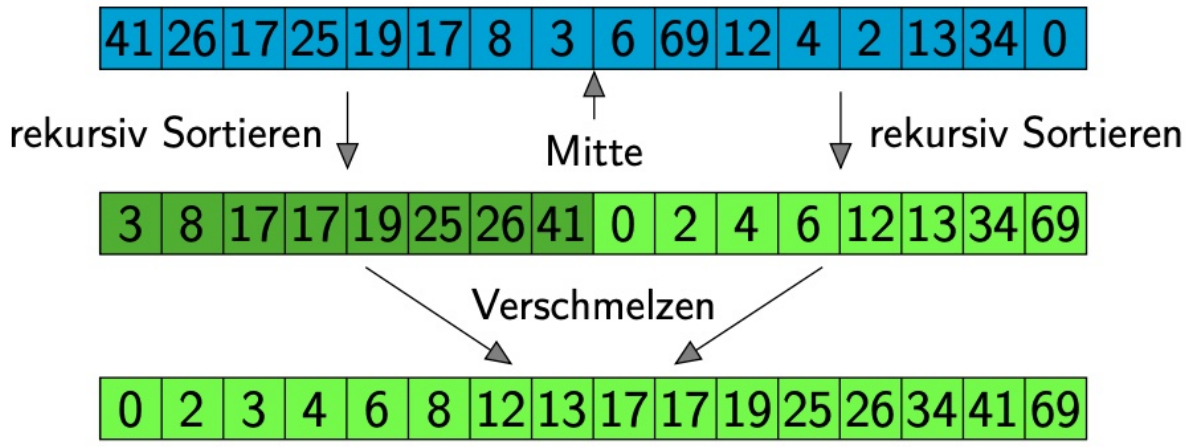
Divide-and-Conquer: Mergesort



Mergesort Strategie:

- Divide: Teile die Eingabefolge in zwei möglichst gleich große Teilfolgen.

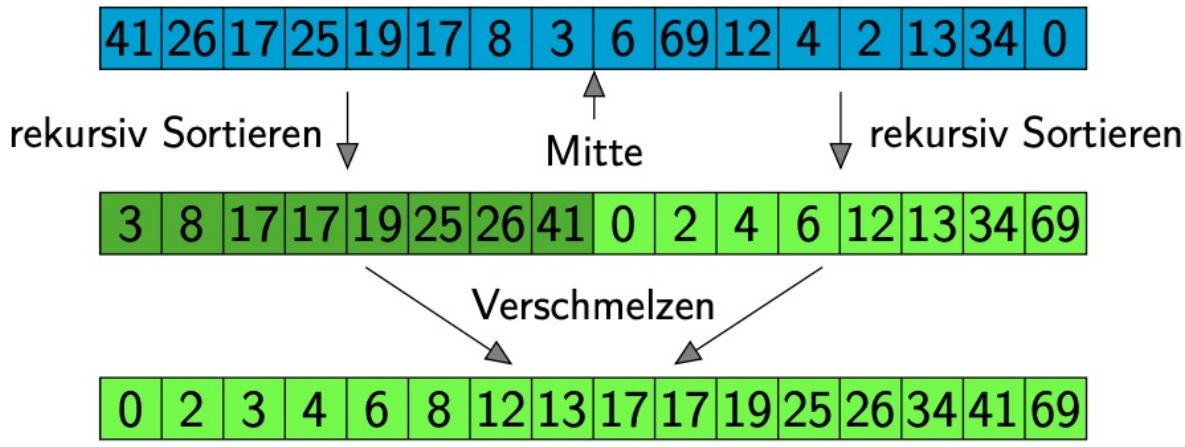
Divide-and-Conquer: Mergesort



Mergesort Strategie:

- Divide: Teile die Eingabefolge in zwei möglichst gleich große Teilfolgen.
- Conquer: Sortiere die Teilfolgen durch rekursive MergeSort Aufrufe, Teilfolgen der Länge 1 sind trivialerweise sortiert.

Divide-and-Conquer: Mergesort



Mergesort Strategie:

- Divide: Teile die Eingabefolge in zwei möglichst gleich große Teilfolgen.
- Conquer: Sortiere die Teilfolgen durch rekursive MergeSort Aufrufe, Teilfolgen der Länge 1 sind trivialerweise sortiert.
- Combine: Verschmelze je zwei sortierte Teilfolgen: Merge.

Mergesort: Algorithmus

```
1 MERGESORT(int[] A, int ℓ, int r)
  /* Eingabe: Feld A[0,...,n − 1], ℓ ≤ r                    */
  /* Nach Ausführung ist A[ℓ,...,r] sortiert.               */
```

Mergesort: Algorithmus

```
1 MERGESORT(int[] A, int ℓ, int r)
  /* Eingabe: Feld A[0,...,n − 1], ℓ ≤ r                    */
  /* Nach Ausführung ist A[ℓ,...,r] sortiert.               */
2 if ℓ < r then
3 |   m = ℓ + (r − ℓ)/2
```

Mergesort: Algorithmus

```
1 MERGESORT(int[] A, int ℓ, int r)
  /* Eingabe: Feld A[0,...,n − 1], ℓ ≤ r                    */
  /* Nach Ausführung ist A[ℓ,...,r] sortiert.               */
2 if ℓ < r then
3 |   m = ℓ + (r − ℓ)/2
4 |   MERGESORT(A, ℓ, m)
  |
  |
```

Mergesort: Algorithmus

```
1 MERGESORT(int[] A, int ℓ, int r)
  /* Eingabe: Feld A[0,...,n − 1], ℓ ≤ r                    */
  /* Nach Ausführung ist A[ℓ,...,r] sortiert.               */
2 if ℓ < r then
3 |   m = ℓ + (r − ℓ)/2
4 |   MERGESORT(A, ℓ, m)
5 |   MERGESORT(A, m + 1, r)
```

Mergesort: Algorithmus

```
1 MERGESORT(int[] A, int ℓ, int r)
  /* Eingabe: Feld A[0,...,n−1], ℓ ≤ r                    */
  /* Nach Ausführung ist A[ℓ,...,r] sortiert.             */
2 if ℓ < r then
3 |   m = ℓ + (r − ℓ)/2
4 |   MERGESORT(A, ℓ, m)
5 |   MERGESORT(A, m + 1, r)
6 |   MERGE(A, ℓ, m, r)
```

Mergesort: Algorithmus

```
1 MERGESORT(int[] A, int ℓ, int r)
  /* Eingabe: Feld A[0,...,n − 1], ℓ ≤ r                    */
  /* Nach Ausführung ist A[ℓ,...,r] sortiert.               */
2 if ℓ < r then
3     m = ℓ + (r − ℓ)/2
4     MERGESORT(A, ℓ, m)
5     MERGESORT(A, m + 1, r)
6     MERGE(A, ℓ, m, r)
```

- Aufruf mit $\textsc{Mergesort}(\mathtt{A}, 0, \mathtt{n}-1)$.

Mergesort: Algorithmus

```
1 MERGESORT(int[] A, int ℓ, int r)
  /* Eingabe: Feld A[0,...,n − 1], ℓ ≤ r                    */
  /* Nach Ausführung ist A[ℓ,...,r] sortiert.               */
2 if ℓ < r then
3     m = ℓ + (r − ℓ)/2
4     MERGESORT(A, ℓ, m)
5     MERGESORT(A, m + 1, r)
6     MERGE(A, ℓ, m, r)
```

- Aufruf mit $\textsc{Mergesort}(\mathtt{A}, 0, \mathtt{n}-1)$.

Mergesort: Algorithmus

```
1 MERGESORT(int[] A, int ℓ, int r)
  /* Eingabe: Feld A[0,...,n−1], ℓ ≤ r                    */
  /* Nach Ausführung ist A[ℓ,...,r] sortiert.             */
2 if ℓ < r then
3     m = ℓ + (r − ℓ)/2
4     MERGESORT(A, ℓ, m)
5     MERGESORT(A, m + 1, r)
6     MERGE(A, ℓ, m, r)
```

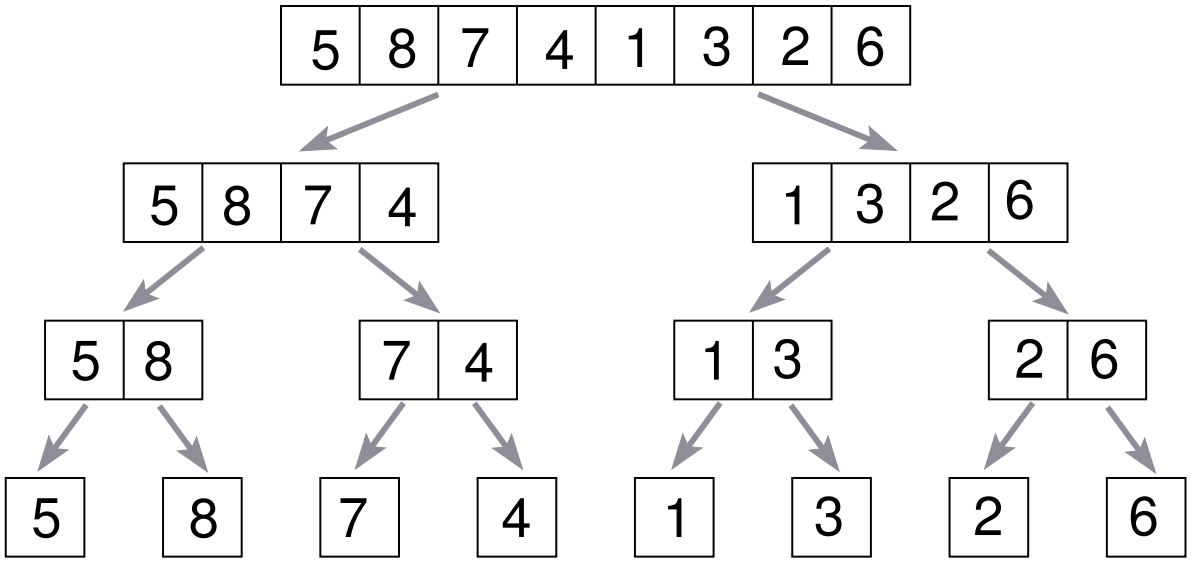
- Aufruf mit $\textsc{Mergesort}(\mathtt{A}, 0, \mathtt{n}-1)$.
- Die Korrektheit folgt per Induktion aus der Korrektheit von $\textsc{Merge}$ (die wir noch beweisen müssen).

Mergesort: Algorithmus

```
1 MERGESORT(int[] A, int ℓ, int r)
  /* Eingabe: Feld A[0,...,n − 1], ℓ ≤ r                    */
  /* Nach Ausführung ist A[ℓ,...,r] sortiert.               */
2 if ℓ < r then
3     m = ℓ + (r − ℓ)/2
4     MERGESORT(A, ℓ, m)
5     MERGESORT(A, m + 1, r)
6     MERGE(A, ℓ, m, r)
```

- Aufruf mit $\textsc{Mergesort}(\mathtt{A}, 0, \mathtt{n}-1)$.
- Die Korrektheit folgt per Induktion aus der Korrektheit von $\textsc{Merge}$ (die wir noch beweisen müssen).
- $\textsc{Merge}$ in Linearzeit! (werden wir zeigen)

MergeSort: Beispiel



Merge

```
1 MERGE(int[] A, int ℓ, int m, int r)
2 n_1 := m − ℓ
```

Merge

```
1 MERGE(int[] A, int ℓ, int m, int r)
2 n_1 := m − ℓ
3 n_2 := r − m − 1
```

Merge

```
1 MERGE(int[] A, int ℓ, int m, int r)
2 n_1 := m − ℓ
3 n_2 := r − m − 1
4 copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1] with
    L[n_1 + 1] := ∞
```

Merge

```
1 MERGE(int[] A, int ℓ, int m, int r)
2 n_1 := m − ℓ
3 n_2 := r − m − 1
4 copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1] with
    L[n_1 + 1] := ∞
5 copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]
    with R[n_2 + 1] := ∞
```

Merge

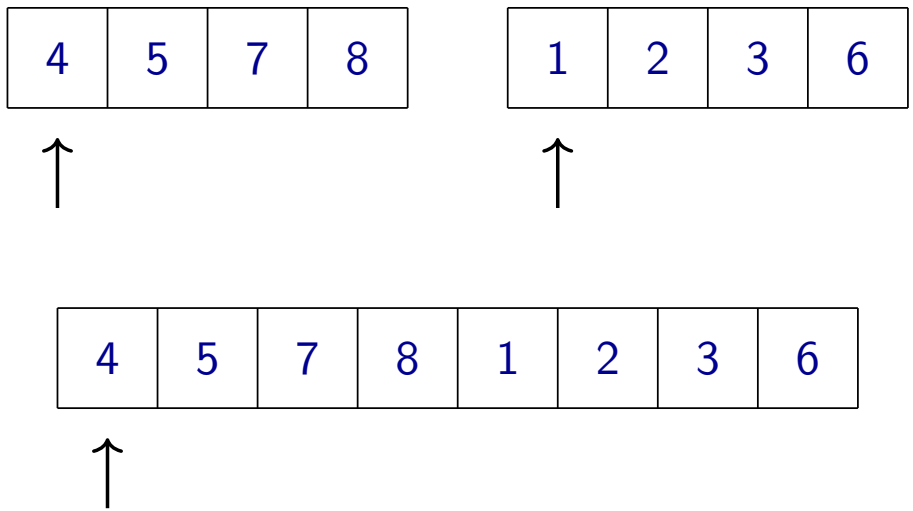
1 MERGE(int[] A, int ℓ, int m, int r)
2 $\mathtt{n}_1 := \mathtt{m} - \ell$
3 $\mathtt{n}_2 := \mathtt{r} - \mathtt{m} - 1$
4 **copy** $\mathtt{A}[\ell, \ldots, \mathtt{m}]$ to new array $\mathtt{L}[0, \ldots, \mathtt{n}_1 + 1]$ with $\mathtt{L}[\mathtt{n}_1 + 1] := \infty$
5 **copy** $\mathtt{A}[\mathtt{m} + 1, \ldots, \mathtt{r}]$ to new array $\mathtt{R}[0, \ldots, \mathtt{n}_2 + 1]$ with $\mathtt{R}[\mathtt{n}_2 + 1] := \infty$
6 $i := 0, j := 0$

Merge

```
1  MERGE(int[] A, int ℓ, int m, int r)
2  n_1 := m − ℓ
3  n_2 := r − m − 1
4  copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1] with
     L[n_1 + 1] := ∞
5  copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]
     with R[n_2 + 1] := ∞
6  i := 0, j := 0
7  for k = ℓ to r do
8      if L[i] ≤ R[j] then
9          A[k] := L[i]
10         i := i + 1
11     else
12         A[k] := R[j]
13         j := j + 1
```

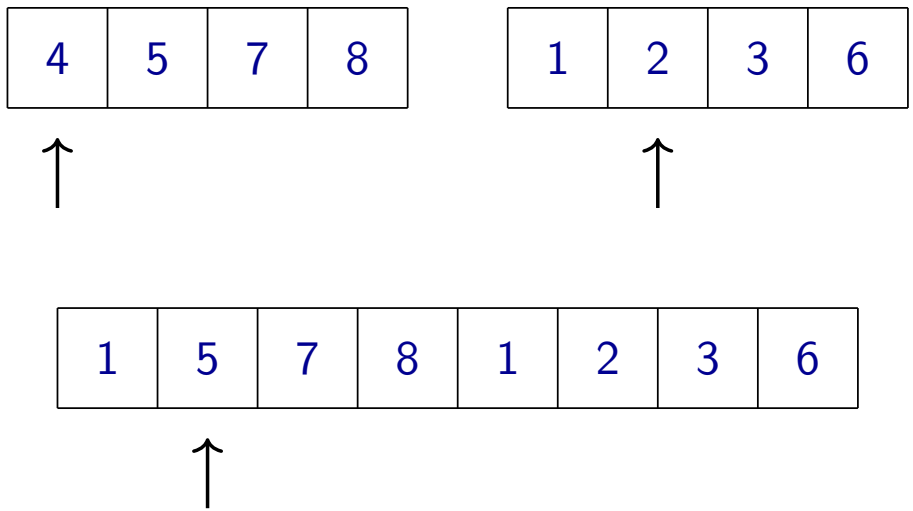
Merge

```
1  MERGE(int[] A, int ℓ, int m, int r)
2  n_1 := m − ℓ
3  n_2 := r − m − 1
4  copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1] with
     L[n_1 + 1] := ∞
5  copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]
     with R[n_2 + 1] := ∞
6  i := 0, j := 0
7  for k = ℓ to r do
8      if L[i] ≤ R[j] then
9          A[k] := L[i]
10         i := i + 1
11     else
12         A[k] := R[j]
13         j := j + 1
```



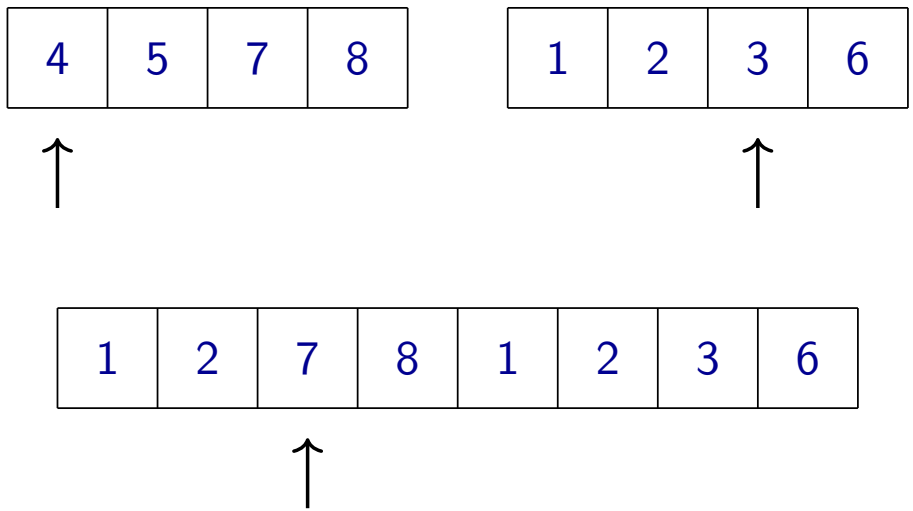
Merge

```
1  MERGE(int[] A, int ℓ, int m, int r)
2  n_1 := m − ℓ
3  n_2 := r − m − 1
4  copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1] with
     L[n_1 + 1] := ∞
5  copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]
     with R[n_2 + 1] := ∞
6  i := 0, j := 0
7  for k = ℓ to r do
8      if L[i] ≤ R[j] then
9          A[k] := L[i]
10         i := i + 1
11     else
12         A[k] := R[j]
13         j := j + 1
```



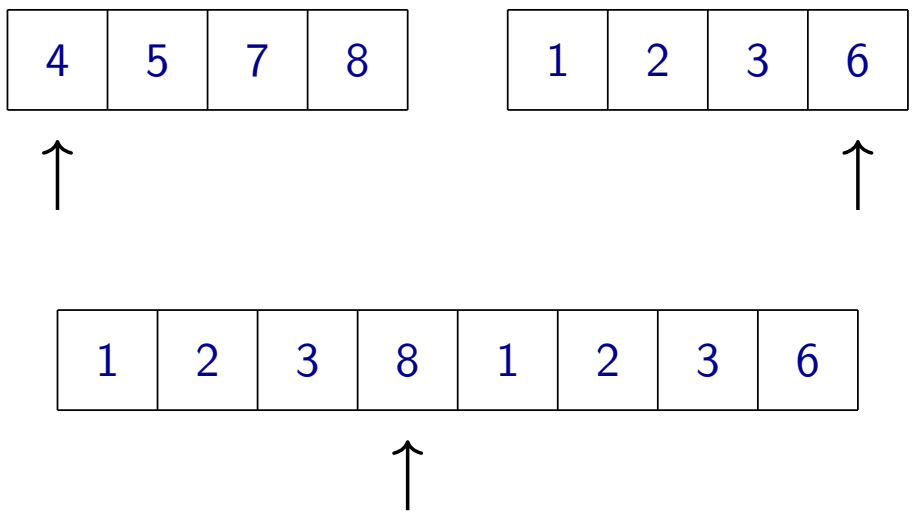
Merge

```
1  MERGE(int[] A, int ℓ, int m, int r)
2  n_1 := m − ℓ
3  n_2 := r − m − 1
4  copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1] with
     L[n_1 + 1] := ∞
5  copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]
     with R[n_2 + 1] := ∞
6  i := 0, j := 0
7  for k = ℓ to r do
8      if L[i] ≤ R[j] then
9          A[k] := L[i]
10         i := i + 1
11     else
12         A[k] := R[j]
13         j := j + 1
```



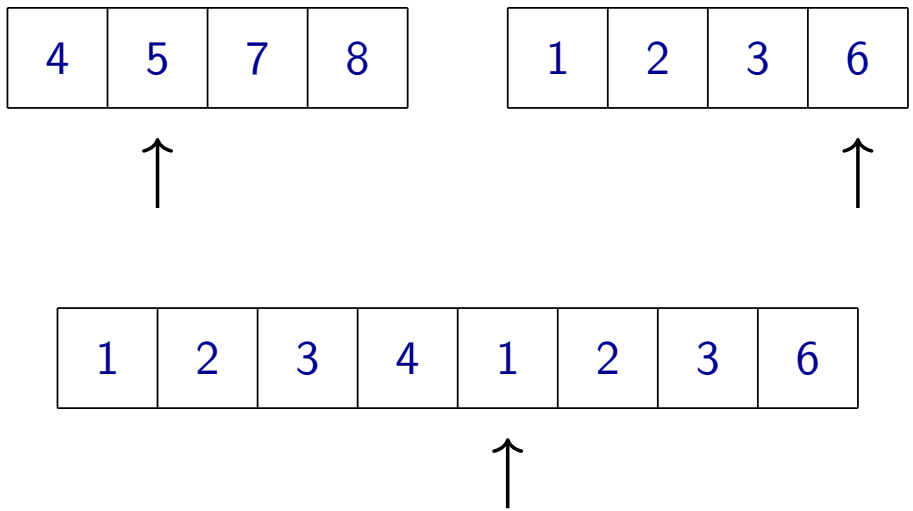
Merge

```
1  MERGE(int[] A, int ℓ, int m, int r)
2  n_1 := m − ℓ
3  n_2 := r − m − 1
4  copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1] with
     L[n_1 + 1] := ∞
5  copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]
     with R[n_2 + 1] := ∞
6  i := 0, j := 0
7  for k = ℓ to r do
8      if L[i] ≤ R[j] then
9          A[k] := L[i]
10         i := i + 1
11     else
12         A[k] := R[j]
13         j := j + 1
```



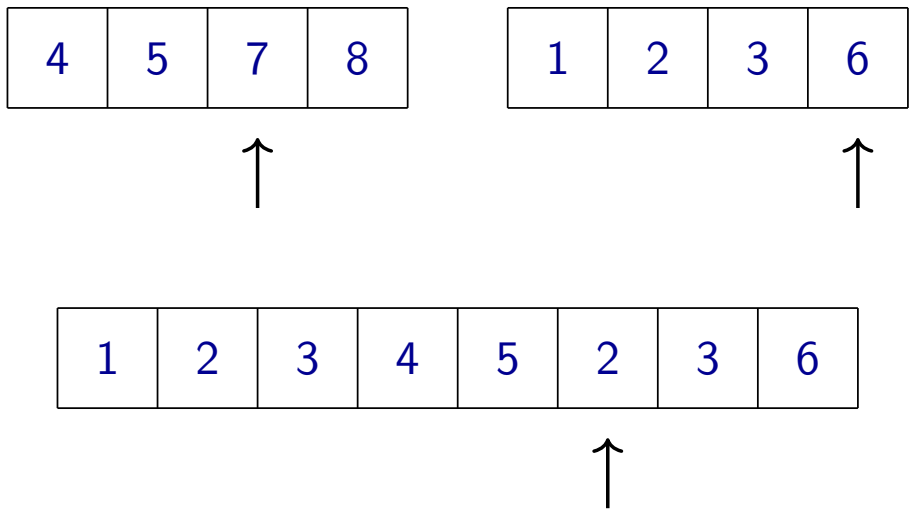
Merge

```
1  MERGE(int[] A, int ℓ, int m, int r)
2  n_1 := m − ℓ
3  n_2 := r − m − 1
4  copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1] with
     L[n_1 + 1] := ∞
5  copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]
     with R[n_2 + 1] := ∞
6  i := 0, j := 0
7  for k = ℓ to r do
8      if L[i] ≤ R[j] then
9          A[k] := L[i]
10         i := i + 1
11     else
12         A[k] := R[j]
13         j := j + 1
```



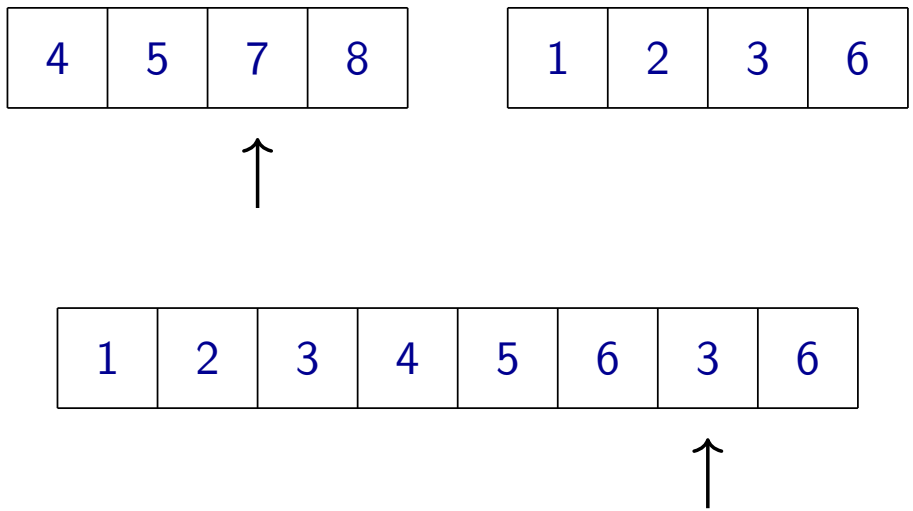
Merge

```
1  MERGE(int[] A, int ℓ, int m, int r)
2  n_1 := m − ℓ
3  n_2 := r − m − 1
4  copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1] with
     L[n_1 + 1] := ∞
5  copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]
     with R[n_2 + 1] := ∞
6  i := 0, j := 0
7  for k = ℓ to r do
8      if L[i] ≤ R[j] then
9          A[k] := L[i]
10         i := i + 1
11     else
12         A[k] := R[j]
13         j := j + 1
```



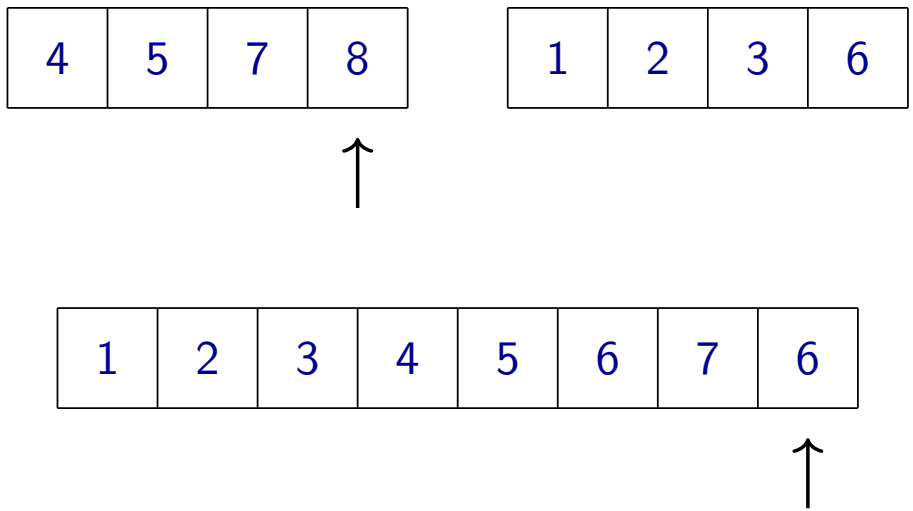
Merge

```
1  MERGE(int[] A, int ℓ, int m, int r)
2  n_1 := m − ℓ
3  n_2 := r − m − 1
4  copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1] with
     L[n_1 + 1] := ∞
5  copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]
     with R[n_2 + 1] := ∞
6  i := 0, j := 0
7  for k = ℓ to r do
8      if L[i] ≤ R[j] then
9          A[k] := L[i]
10         i := i + 1
11     else
12         A[k] := R[j]
13         j := j + 1
```



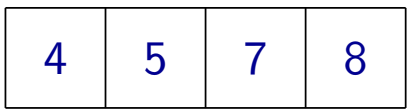
Merge

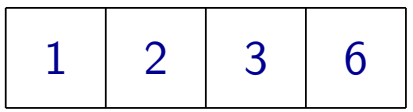
```
1  MERGE(int[] A, int ℓ, int m, int r)
2  n_1 := m − ℓ
3  n_2 := r − m − 1
4  copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1] with
     L[n_1 + 1] := ∞
5  copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]
     with R[n_2 + 1] := ∞
6  i := 0, j := 0
7  for k = ℓ to r do
8      if L[i] ≤ R[j] then
9          A[k] := L[i]
10         i := i + 1
11     else
12         A[k] := R[j]
13         j := j + 1
```

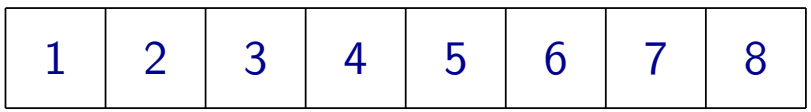


Merge

```
1  MERGE(int[] A, int ℓ, int m, int r)
2  n_1 := m − ℓ
3  n_2 := r − m − 1
4  copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1] with
     L[n_1 + 1] := ∞
5  copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]
     with R[n_2 + 1] := ∞
6  i := 0, j := 0
7  for k = ℓ to r do
8      if L[i] ≤ R[j] then
9          A[k] := L[i]
10         i := i + 1
11     else
12         A[k] := R[j]
13         j := j + 1
```







Merge Korrektheit

For-Schleife:

Merge Korrektheit

For-Schleife: In der **for**-Schleife werden $r - \ell + 1$ Schritte ausgeführt, wobei die folgende Invariante erhalten bleibt.

Merge Korrektheit

For-Schleife: In der **for**-Schleife werden $r - \ell + 1$ Schritte ausgeführt, wobei die folgende Invariante erhalten bleibt.

Invariante: Vor der Iteration enthält $\mathtt{A}[\ell, \ldots, \mathtt{k} - 1]$ (falls $k = \ell$ so ist das Array leer) die $k - \ell$ kleinsten Elemente von $\mathtt{L}$ und $\mathtt{R}$ in sortierter Reihenfolge.

Merge Korrektheit

For-Schleife: In der **for**-Schleife werden $r - \ell + 1$ Schritte ausgeführt, wobei die folgende Invariante erhalten bleibt.

Invariante: Vor der Iteration enthält $\mathtt{A}[\ell, \ldots, \mathtt{k} - 1]$ (falls $k = \ell$ so ist das Array leer) die $k - \ell$ kleinsten Elemente von `L` und `R` in sortierter Reihenfolge. `L[i]` und `R[j]` sind die kleinsten Elemente von `L` und `R`, die noch nicht in `A` eingefügt wurden.

Merge Korrektheit

Initialisierung:

Merge Korrektheit

Initialisierung:

- Vor der ersten Iteration gilt $k = \ell$.

Merge Korrektheit

Initialisierung:

- Vor der ersten Iteration gilt $k = \ell$.
- Also ist $\mathtt{A}[\ell, \ldots, \mathtt{k} - 1]$ leer und enthält die $k - \ell = 0$ kleinsten Elemente von $\mathtt{L}$ und $\mathtt{R}$ in sortierter Reihenfolge.

Merge Korrektheit

Initialisierung:

- Vor der ersten Iteration gilt $k = \ell$.
- Also ist $\texttt{A}[\ell, \ldots, \texttt{k} - 1]$ leer und enthält die $k - \ell = 0$ kleinsten Elemente von `L` und `R` in sortierter Reihenfolge.
- $i = j = 0$. Da `L` und `R` sortiert sind, enthalten `L[i]` und `R[j]` die kleinsten Elemente von `L` und `R`, die noch nicht in `A` eingefügt wurden.

Merge

```
1  MERGE(int[] A, int ℓ, int m, int r)
2  n_1 := m − ℓ
3  n_2 := r − m − 1
4  copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1] with
     L[n_1 + 1] := ∞
5  copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]
     with R[n_2 + 1] := ∞
6  i := 0, j := 0
7  for k = ℓ to r do
8      if L[i] ≤ R[j] then
9          A[k] := L[i]
10         i := i + 1
11     else
12         A[k] := R[j]
13         j := j + 1
```

Merge Korrektheit

Erhaltung der Invariante:

- Angenommen $\texttt{L[i]} \leq \texttt{R[j]}$, dann ist $\texttt{L[i]}$ das kleinste Element, das noch nicht in $\texttt{A}$ einsortiert wurde.
- Nach Annahme enthält $\texttt{A}[\ell, \ldots, \texttt{k} - 1]$ die $k - \ell$ kleinsten Elemente.

Merge Korrektheit

Erhaltung der Invariante:

- Angenommen $\texttt{L[i]} \leq \texttt{R[j]}$, dann ist $\texttt{L[i]}$ das kleinste Element, das noch nicht in $\texttt{A}$ einsortiert wurde.
- Nach Annahme enthält $\texttt{A}[\ell, \ldots, \texttt{k} - 1]$ die $k - \ell$ kleinsten Elemente.
- In Zeile 9 wird $\texttt{L[i]}$ in $\texttt{A[k]}$ kopiert, also enthält $\texttt{A}[\ell, \ldots, \texttt{k}]$ die $k - \ell + 1$ kleinsten Elemente.

Merge Korrektheit

Erhaltung der Invariante:

- Angenommen $\texttt{L[i]} \leq \texttt{R[j]}$, dann ist $\texttt{L[i]}$ das kleinste Element, das noch nicht in $\texttt{A}$ einsortiert wurde.
- Nach Annahme enthält $\texttt{A}[\ell, \ldots, \texttt{k} - 1]$ die $k - \ell$ kleinsten Elemente.
- In Zeile 9 wird $\texttt{L[i]}$ in $\texttt{A[k]}$ kopiert, also enthält $\texttt{A}[\ell, \ldots, \texttt{k}]$ die $k - \ell + 1$ kleinsten Elemente.
- Durch Erhöhung von k (durch die **for**-Schleife) und $\texttt{i}$ ist die Invariante wieder hergestellt.

Merge Korrektheit

Erhaltung der Invariante:

- Angenommen $\texttt{L[i]} \leq \texttt{R[j]}$, dann ist $\texttt{L[i]}$ das kleinste Element, das noch nicht in $\texttt{A}$ einsortiert wurde.
- Nach Annahme enthält $\texttt{A}[\ell, \ldots, \texttt{k} - 1]$ die $k - \ell$ kleinsten Elemente.
- In Zeile 9 wird $\texttt{L[i]}$ in $\texttt{A[k]}$ kopiert, also enthält $\texttt{A}[\ell, \ldots, \texttt{k}]$ die $k - \ell + 1$ kleinsten Elemente.
- Durch Erhöhung von k (durch die **for**-Schleife) und $\texttt{i}$ ist die Invariante wieder hergestellt.
- Der Fall $\texttt{L[i]} > \texttt{R[j]}$ ist analog.

Merge Korrektheit

Terminierung:

Merge Korrektheit

Terminierung:

- Bei Terminierung wird die **for**-Schleife nicht noch einmal ausgeführt, es gilt $k = r + 1$.

Merge Korrektheit

Terminierung:

- Bei Terminierung wird die **for**-Schleife nicht noch einmal ausgeführt, es gilt $k = r + 1$.
- $\mathtt{A}[\ell, \ldots, \mathtt{r}]$ enthält die $r - \ell + 1$ kleinsten Elemente von $\mathtt{L}$ und $\mathtt{R}$ (alle) in sortierter Reihenfolge.

Merge Korrektheit

Terminierung:

- Bei Terminierung wird die **for**-Schleife nicht noch einmal ausgeführt, es gilt $k = r + 1$.
- $\mathtt{A}[\ell, \ldots, \mathtt{r}]$ enthält die $r - \ell + 1$ kleinsten Elemente von `L` und `R` (alle) in sortierter Reihenfolge.

Es folgt die Korrektheit von `Merge`.

Laufzeitanalyse Merge

```
1  MERGE(int[] A, int ℓ, int m, int r)
2  n_1 := m − ℓ
3  n_2 := r − m − 1
4  copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1]
5  copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]
6  i := 0, j := 0
7  for k = ℓ to r do
8      if L[i] ≤ R[j] then
9          A[k] := L[i]
10         i := i + 1
11     else
12         A[k] := R[j]
13         j := j + 1
```

Laufzeitanalyse Merge

```
1  MERGE(int[] A, int ℓ, int m, int r)
2  n_1 := m − ℓ                                                    O(1)
3  n_2 := r − m − 1                                                O(1)
4  copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1]               O(1)
5  copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]           O(m − ℓ)
6  i := 0, j := 0                                                  O(r − m)
7  for k = ℓ to r do                                               O(1)
8      if L[i] ≤ R[j] then                                         O(r − ℓ)
9          A[k] := L[i]                                            O(1)
10         i := i + 1                                              O(1)
11     else                                                        O(1)
12         A[k] := R[j]                                            O(1)
13         j := j + 1                                              O(1)
```

Laufzeitanalyse Merge

```
1  MERGE(int[] A, int ℓ, int m, int r)
2  n_1 := m − ℓ
3  n_2 := r − m − 1
4  copy A[ℓ, ..., m] to new array L[0, ..., n_1 + 1]
5  copy A[m + 1, ..., r] to new array R[0, ..., n_2 + 1]
6  i := 0, j := 0
7  for k = ℓ to r do
8      if L[i] ≤ R[j] then
9          A[k] := L[i]
10         i := i + 1
11     else
12         A[k] := R[j]
13         j := j + 1
```

```
1  O(1)
2  O(1)
3  O(1)
4  O(m − ℓ)
5  O(r − m)
6  O(1)
7  O(r − ℓ)
8  O(1)
9  O(1)
10 O(1)
11 O(1)
12 O(1)
13 O(1)
```

Laufzeit: $\mathcal{O}(m-\ell) + \mathcal{O}(r-m) + \mathcal{O}(r-\ell) \cdot \mathcal{O}(1) = \mathcal{O}(r-\ell) = \mathcal{O}(n)$

Zusammenfassung

1. (Asymptotische) Laufzeitanalyse
2. $\mathcal{O}$-Notation, Landau-Notation
3. Divide-and-Conquer als Algorithmendesign-Prinzip
4. Anwendung: Sortierverfahren MergeSort