

SS 2019

# Algorithmen und Datenstrukturen

Harald Räche

Fakultät für Informatik  
TU München

<http://www14.in.tum.de/lehre/2019SS/ad/>

Sommersemester 2019

# Teil I

## Organisatorisches

# Personen

## Vorlesung

Prof. Harald Räcke (raecke@in.tum.de)  
Lehrstuhl für Algorithmen & Komplexität  
(Prof. Albers)



## Zentralübung

Sebastian Weiß (sebastian13.weiss@tum.de)  
Doktorand am  
Lehrstuhl für Grafik und Visualisierung  
(Prof. Westermann)



# Personen

## Tutoren

Lizichong Li

Lisa Roßgoderer

Rojda Hicsanmaz

Jan Hünemann

Nadia Masmoudi

Islam Benshaban

Aaron Kutzner

Stefan Kammermeier

Martin Zimmermann

Selin Kesler

# Termine

## Vorlesung

Montag, 8:00 - 9:30, Raum 1200

Mittwoch, 15:00 - 16:30, Raum 1200

## Zentralübung

Dienstag, 9:45 - 11:15, Raum 1200

## Terminplan

Bitte Terminplan auf Vorlesungswebseite beachten!

# Termine

## Vorlesung

Montag, 8:00 - 9:30, Raum 1200

Mittwoch, 15:00 - 16:30, Raum 1200

## Zentralübung

Dienstag, 9:45 - 11:15, Raum 1200

## Terminplan

Bitte Terminplan auf Vorlesungswebseite beachten!

# Termine

## Tutorübungen

- 1 Fr 9:45-11:15 0509.EG.999 (0999)
- 2 Fr 9:45-11:15 0504.EG.406 (0406)
- 3 Do 8:00- 9:30 0504.EG.406 (0406)
- 4 Fr 11:30-13:00 0509.EG.999 (0999)
- 5 Di 15:00-16:30 0509.EG.999 (0999)
- 6 Di 15:00-16:30 0103.05.325 (N5325)
- 7 Mi 11:30-13:00 0504.EG.406 (0406)
- 8 Fr 11:30-13:00 0504.EG.406 (0406)
- 9 Do 8:00- 9:30 0103.05.325 (N5325)
- 10 Di 15:00-16:30 0504.EG.406 (0406)

## Moodle

<https://www.moodle.tum.de/course/view.php?id=45840>

- ▶ sämtliche Vorlesungsmaterialien (Folien, Übungsblätter, Übungsklausuren)
- ▶ aktuelle Nachrichten
- ▶ Diskussions-Forum



# Kontakt und Feedback

## Für Fragen:

- ▶ Persönlich
  - ▶ Sprechstunde nach Vereinbarung
  - ▶ in der Tutorübung
  - ▶ in der Zentralübung
- ▶ Email (inhaltliche Fragen ins Forum)
- ▶ Diskussions-Forum in Moodle

## Feedback

Feedback zur Vorlesung/Übung ist jederzeit willkommen (bitte nicht erst in den Evaluierungsbögen)!

# Kontakt und Feedback

## Für Fragen:

- ▶ Persönlich
  - ▶ Sprechstunde nach Vereinbarung
  - ▶ in der Tutorübung
  - ▶ in der Zentralübung
- ▶ Email (inhaltliche Fragen ins Forum)
- ▶ Diskussions-Forum in Moodle

## Feedback

Feedback zur Vorlesung/Übung ist jederzeit willkommen (bitte nicht erst in den Evaluierungsbögen)!

# Ablauf: Vorlesung

## Folienvortrag

- ▶ mit gelegentlichen Annotationen
- ▶ kein Skript!
- ▶ Folien vor Vorlesung als PDF zum Download

Eigene Notizen sind hilfreich!

# Ablauf: Zentralübung

- ▶ Keine Zentralübung am **Dienstag, 22.04.2019**
- ▶ Erste Zentralübung: **Dienstag, 30.04.2018**

# Ablauf: Zentralübung

- ▶ Keine Zentralübung am **Dienstag, 22.04.2019**
- ▶ Erste Zentralübung: **Dienstag, 30.04.2018**
  
- ▶ Beispielaufgaben zu ausgewählten Themen der Vorlesung
- ▶ Begleitetes Programmieren in C/C++

# Ablauf: Zentralübung

- ▶ Keine Zentralübung am **Dienstag, 22.04.2019**
- ▶ Erste Zentralübung: **Dienstag, 30.04.2018**
  
- ▶ Beispielaufgaben zu ausgewählten Themen der Vorlesung
- ▶ Begleitetes Programmieren in C/C++
  
- ▶ Beantwortung von ausgewählten Fragen
- ▶ nur Fragen zur Vorlesung (Fragen zu Übungsblättern in Tutorübungsgruppen)

# Ablauf: Tutorübungen

Keine Tutorübungen in der ersten Vorlesungswoche

Erste Tutorübung ab **Montag, 6.05.2019**

Jede Woche ein Übungsblatt

- ▶ 3-5 Aufgaben
- ▶ zur Anwendung und Vertiefung der Vorlesung

In der **Tutorübung**

- ▶ Besprechung der Aufgaben
- ▶ Individuelle Beantwortung von Fragen
- ▶ keine Korrektur der Aufgaben

Eigene Bearbeitung der Übungsblätter dringend empfohlen!

- ▶ z.B. auch in kleinen Gruppen

# Ablauf: Tutorübungen

Keine Tutorübungen in der ersten Vorlesungswoche

Erste Tutorübung ab **Montag, 6.05.2019**

Jede Woche ein Übungsblatt

- ▶ 3-5 Aufgaben
- ▶ zur Anwendung und Vertiefung der Vorlesung

In der **Tutorübung**

- ▶ Besprechung der Aufgaben
- ▶ Individuelle Beantwortung von Fragen
- ▶ keine Korrektur der Aufgaben

Eigene Bearbeitung der Übungsblätter dringend empfohlen!

- ▶ z.B. auch in kleinen Gruppen



# Ablauf: Tutorübungen

Keine Tutorübungen in der ersten Vorlesungswoche

Erste Tutorübung ab **Montag, 6.05.2019**

Jede Woche ein Übungsblatt

- ▶ 3-5 Aufgaben
- ▶ zur Anwendung und Vertiefung der Vorlesung

In der **Tutorübung**

- ▶ Besprechung der Aufgaben
- ▶ Individuelle Beantwortung von Fragen
- ▶ keine Korrektur der Aufgaben

Eigene Bearbeitung der Übungsblätter dringend empfohlen!

- ▶ z.B. auch in kleinen Gruppen

# Ablauf: Tutorübungen

Keine Tutorübungen in der ersten Vorlesungswoche

Erste Tutorübung ab **Montag, 6.05.2019**

Jede Woche ein Übungsblatt

- ▶ 3-5 Aufgaben
- ▶ zur Anwendung und Vertiefung der Vorlesung

In der **Tutorübung**

- ▶ Besprechung der Aufgaben
- ▶ Individuelle Beantwortung von Fragen
- ▶ keine Korrektur der Aufgaben

Eigene Bearbeitung der Übungsblätter dringend empfohlen!

- ▶ z.B. auch in kleinen Gruppen

# Leistungsnachweis

Klausur am 6.08.2019

- ▶ Schriftliche Prüfung
- ▶ Dauer: 120 Minuten
- ▶ erlaubte Hilfsmittel: handbeschriebenes DIN A4 Blatt

Vorbereitung durch **aktive** Teilnahme und Bearbeitung des Übungs-Programms

Keine Probeklausuren

# Leistungsnachweis

Klausur am 6.08.2019

- ▶ Schriftliche Prüfung
- ▶ Dauer: 120 Minuten
- ▶ erlaubte Hilfsmittel: handbeschriebenes DIN A4 Blatt

Vorbereitung durch **aktive** Teilnahme und Bearbeitung des Übungs-Programms

Keine Probeklausuren

# Leistungsnachweis

Klausur am 6.08.2019

- ▶ Schriftliche Prüfung
- ▶ Dauer: 120 Minuten
- ▶ erlaubte Hilfsmittel: handbeschriebenes DIN A4 Blatt

Vorbereitung durch **aktive** Teilnahme und Bearbeitung des Übungs-Programms

Keine Probeklausuren

# Allgemeine Regeln

Wir dulden keine Ruhestörung!

- ▶ Weder in Vorlesung und Übung, noch in Tutorübungen
- ▶ Es besteht keine Anwesenheitspflicht!




Fragen nach den Veranstaltungen!

- ▶ ausserhalb des Hörsaals!
- ▶ Fragen sonst gerne während den Veranstaltungen, per Email/Diskussionsforum oder persönlich nach Vereinbarung

Transferleistung zu Computertechnik bzw. GOP dringend empfohlen

- ▶ Begleitetes Programmieren während der Zentralübung aktiv wahrnehmen
- ▶ Hilft enorm für besseres Verständnis der Algorithmen

# 1 Literatur

-  Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman:  
*The design and analysis of computer algorithms*,  
Addison-Wesley Publishing Company: Reading (MA), 1974
-  Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest,  
Clifford Stein:  
*Introduction to algorithms*,  
McGraw-Hill, 1990
-  Michael T. Goodrich, Roberto Tamassia:  
*Algorithm design: Foundations, analysis, and internet  
examples*,  
John Wiley & Sons, 2002

# 1 Literatur



Ronald L. Graham, Donald E. Knuth, Oren Patashnik:  
*Concrete Mathematics*,  
2. Auflage, Addison-Wesley, 1994



Volker Heun:  
*Grundlegende Algorithmen: Einführung in den Entwurf und die Analyse effizienter Algorithmen*,  
2. Auflage, Vieweg, 2003



Jon Kleinberg, Eva Tardos:  
*Algorithm Design*,  
Addison-Wesley, 2005



Donald E. Knuth:  
*The art of computer programming. Vol. 1: Fundamental Algorithms*,  
3. Auflage, Addison-Wesley, 1997



# 1 Literatur



Uwe Schöning:

*Algorithmik,*

Spektrum Akademischer Verlag, 2001



Steven S. Skiena:

*The Algorithm Design Manual,*

Springer, 1998

# Teil II

## Grundlagen

# Ziele der Vorlesung

## Wissen:

- ▶ Algorithmische Prinzipien verstehen und anwenden
- ▶ Grundlegende Algorithmen kennen lernen
- ▶ Grundlegende Datenstrukturen kennen lernen
- ▶ Bewertung von Effizienz und Korrektheit

## Methodenkompetenz:

- ▶ für Entwurf von effizienten und korrekten Algorithmen
- ▶ zur Analyse von Algorithmen
- ▶ zur Umsetzung auf dem Computer

# Ziele der Vorlesung

## Wissen:

- ▶ Algorithmische Prinzipien verstehen und anwenden
- ▶ Grundlegende Algorithmen kennen lernen
- ▶ Grundlegende Datenstrukturen kennen lernen
- ▶ Bewertung von Effizienz und Korrektheit

## Methodenkompetenz:

- ▶ für Entwurf von effizienten und korrekten Algorithmen
- ▶ zur Analyse von Algorithmen
- ▶ zur Umsetzung auf dem Computer

# Übersicht der Inhalte

Grundlagen:

- 1. Einführung in Algorithmen und Datenstrukturen**  
Motivation, Definitionen, Einordnung
- 2. Grundlagen von Algorithmen**  
Darstellung, elementare Bausteine, Pseudocode
- 3. Grundlagen von Datenstrukturen**  
Primitive Datentypen, Felder, abstrakte Datentypen
- 4. Grundlagen der Korrektheit von Algorithmen**  
Verifikation, Testen, Sortieren
- 5. Grundlagen der Effizienz von Algorithmen**  
Komplexitätsanalyse, Sortieren
- 6. Grundlagen des Algorithmen-Entwurfs**  
Entwurfs-Prinzipien

# Übersicht der Inhalte

Fortgeschrittene Algorithmen und Datenstrukturen:

## 7. Fortgeschrittene Datenstrukturen

Bäume, Graphen, Priority-Queue

## 8. Such-Algorithmen

Elementare Suchmethoden, Suchbäume

## 9. Graph-Algorithmen

Elementare Algorithmen, kürzeste Pfade, Spannbaum

## 10. Numerische Algorithmen

Matrizen-Operationen, Fast Fourier Transform

# Übersicht der Inhalte

Ausgewählte Themen (je nach verfügbarer Zeit):

## 11. Datenkompression

Huffmann-Codes, JPEG

## 12. Kryptographie

symmetrische und asymmetrische  
Verschlüsselungsverfahren

# Was ist ein Algorithmus?

## Duden online:

„Rechenvorgang nach einem bestimmten (sich wiederholenden) Schema“

Beispiele für Algorithmen bereits in der Antike, etwa der **Euklidische Algorithmus** zur Berechnung des ggT:

„Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das kleinere vom größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst.“

aus *Euklid: Die Elemente, Buch VII (Clemens Thaer)*



# Was ist ein Algorithmus?

## Duden online:

„Rechenvorgang nach einem bestimmten (sich wiederholenden) Schema“

Beispiele für Algorithmen bereits in der Antike, etwa der **Euklidsche Algorithmus** zur Berechnung des ggT:

„Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das kleinere vom größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst.“

aus *Euklid: Die Elemente, Buch VII (Clemens Thaer)*

# Was ist ein Algorithmus?

## M. Broy: Informatik: Eine grundlegende Einführung

„Ein Algorithmus ist ein Verfahren

- ▶ mit einer **präzisen** (d.h. in einer genau festgelegten Sprache abgefassten),
- ▶ **endlichen** Beschreibung,
- ▶ unter Verwendung
  - ▶ **effektiver** (d.h. tatsächlich ausführbarer),
  - ▶ **elementarer** (Verarbeitungs-) Schritte.“

# Was ist ein Algorithmus?

H. Rogers:

Theory of Recursive Functions and Effective Computability

„Ein Algorithmus ist eine

- ▶ **deterministische** Handlungsvorschrift,
- ▶ die auf eine bestimmte Klasse von **Eingaben** angewendet werden kann,
- ▶ und für jede dieser Eingaben eine korrespondierende **Ausgabe** liefert.“

Im weiteren Verlauf des Buches wird mathematische Theorie zur  
↑**Berechenbarkeit** entwickelt

↑**theoretische Informatik**

# Was ist ein Algorithmus?

## Mathematische Definition Algorithmus

Eine Berechnungsvorschrift zur Lösung eines Problems heißt **Algorithmus** genau dann, wenn

- ▶ eine zu dieser Berechnungsvorschrift äquivalente **Turingmaschine** existiert,
- ▶ die für jede **Eingabe**, die eine **Lösung** besitzt, **terminiert**.

Alan Turing (1936): **Turingmaschine** als mathematisches Modell eines Computers

↑theoretische Informatik

## 3 Einführung

Ein **Problem** besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:

Ein Problem beschreibt eine Funktion  $f : E \rightarrow A$ , mit  $E =$  zulässige Eingaben und  $A =$  mögliche Ausgaben.

Beispiele:

• Addition

• Primfaktorzerlegung

• Suchen eines Elements in einer Menge

• Suchen eines Elements in einer Menge (Zug in Position)

## 3 Einführung

Ein **Problem** besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

### **mathematisch:**

Ein Problem beschreibt eine Funktion  $f : E \rightarrow A$ , mit  $E =$  zulässige Eingaben und  $A =$  mögliche Ausgaben.

### Beispiele:

## 3 Einführung

Ein **Problem** besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

### mathematisch:

Ein Problem beschreibt eine Funktion  $f : E \rightarrow A$ , mit  $E =$  zulässige Eingaben und  $A =$  mögliche Ausgaben.

### Beispiele:

- ▶ Addition:  $f : \mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{Q}$
- ▶ Primzahltest:  $f : \mathbb{N} \rightarrow \{\text{yes, no}\}$
- ▶ Schach:  $f : \mathcal{P} \rightarrow \mathcal{Z}$ , wobei  $\mathcal{P}$  die Menge aller Schachpositionen ist, und  $f(P)$ , der beste Zug in Position  $P$ .

## 3 Einführung

Ein **Problem** besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

### mathematisch:

Ein Problem beschreibt eine Funktion  $f : E \rightarrow A$ , mit  $E =$  zulässige Eingaben und  $A =$  mögliche Ausgaben.

### Beispiele:

- ▶ Addition:  $f : \mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{Q}$
- ▶ Primzahltest:  $f : \mathbb{N} \rightarrow \{\text{yes, no}\}$
- ▶ Schach:  $f : \mathcal{P} \rightarrow \mathcal{Z}$ , wobei  $\mathcal{P}$  die Menge aller Schachpositionen ist, und  $f(P)$ , der beste Zug in Position  $P$ .



## 3 Einführung

Ein **Problem** besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

### mathematisch:

Ein Problem beschreibt eine Funktion  $f : E \rightarrow A$ , mit  $E =$  zulässige Eingaben und  $A =$  mögliche Ausgaben.

### Beispiele:

- ▶ Addition:  $f : \mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{Q}$
- ▶ Primzahltest:  $f : \mathbb{N} \rightarrow \{\text{yes, no}\}$
- ▶ Schach:  $f : \mathcal{P} \rightarrow \mathcal{Z}$ , wobei  $\mathcal{P}$  die Menge aller Schachpositionen ist, und  $f(P)$ , der beste Zug in Position  $P$ .

# Algorithmus

Ein **Algorithmus** ist ein **exaktes Verfahren** zur Lösung eines Problems, d.h. zur Bestimmung der gewünschten Resultate.

Man sagt auch ein Algorithmus **berechnet** eine Funktion  $f$ .



Ausschnitt aus Briefmarke, Soviet Union 1983  
Public Domain [↗](#)

Abu Abdallah  
Muhamed ibn Musa  
al-Chwarizmi, ca.  
780–835

## Beobachtung:

Nicht jedes Problem läßt sich durch einen Algorithmus lösen  
(↑**Berechenbarkeitstheorie**).

## Beweisidee:

- ▶ es gibt überabzählbar unendlich viele Probleme
- ▶ es gibt abzählbar unendlich viele Algorithmen

## Beobachtung:

Nicht jedes Problem läßt sich durch einen Algorithmus lösen  
(↑**Berechenbarkeitstheorie**).

## Beweisidee:

- ▶ es gibt **überabzählbar unendlich** viele Probleme
- ▶ es gibt **abzählbar unendlich** viele Algorithmen

# Algorithmus

Das **exakte Verfahren** besteht i.a. darin, eine Abfolge von **elementaren Einzelschritten** der Verarbeitung festzulegen.

**Beispiel:** Alltagsalgorithmen

<i>Resultat</i>	<i>Algorithmus</i>	<i>Einzelschritte</i>
Pullover	Strickmuster	eine links, eine rechts, eine fallen lassen
Kuchen	Rezept	nimm 3 Eier ...
Konzert	Partitur	Noten

# Beispiel: Euklidischer Algorithmus

**Problem:** geg.  $a, b \in \mathbb{N}, a, b \neq 0$ . Bestimme  $\text{ggT}(a, b)$ .

**Algorithmus:**

1. Falls  $a = b$ , brich Berechnung ab. Es gilt  $\text{ggT}(a, b) = a$ .  
Ansonsten gehe zu Schritt 2.
2. Falls  $a > b$ , ersetze  $a$  durch  $a - b$  und setze Berechnung in Schritt 1 fort. Ansonsten gehe zu Schritt 3.
3. Es gilt  $a < b$ . Ersetze  $b$  durch  $b - a$  und setze Berechnung in Schritt 1 fort.

# Beispiel: Euklidischer Algorithmus

**Warum geht das?**

Wir zeigen, für  $a > b$ :  $\text{ggT}(a, b) = \text{ggT}(a - b, b)$ .

Seien  $g = \text{ggT}(a, b)$ ,  $g' = \text{ggT}(a - b, b)$ .

Dann gilt:

# Beispiel: Euklidischer Algorithmus

## Warum geht das?

Wir zeigen, für  $a > b$ :  $\text{ggT}(a, b) = \text{ggT}(a - b, b)$ .

Seien  $g = \text{ggT}(a, b)$ ,  $g' = \text{ggT}(a - b, b)$ .

Dann gilt:

$$\begin{array}{lcl} a & = & q_a \cdot g \\ b & = & q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{lcl} a - b & = & q'_{a-b} \cdot g' \\ b & = & q'_b \cdot g' \end{array}$$



# Beispiel: Euklidischer Algorithmus

## Warum geht das?

Wir zeigen, für  $a > b$ :  $\text{ggT}(a, b) = \text{ggT}(a - b, b)$ .

Seien  $g = \text{ggT}(a, b)$ ,  $g' = \text{ggT}(a - b, b)$ .

Dann gilt:

$$\begin{array}{l} a = q_a \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a - b = q'_{a-b} \cdot g' \\ b = q'_b \cdot g' \end{array}$$

$$\begin{array}{l} a - b = (q_a - q_b) \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a = (q'_{a-b} + q'_b) \cdot g' \\ b = q'_b \cdot g' \end{array}$$

# Beispiel: Euklidischer Algorithmus

## Warum geht das?

Wir zeigen, für  $a > b$ :  $\text{ggT}(a, b) = \text{ggT}(a - b, b)$ .

Seien  $g = \text{ggT}(a, b)$ ,  $g' = \text{ggT}(a - b, b)$ .

Dann gilt:

$$\begin{array}{l} a = q_a \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a - b = q'_{a-b} \cdot g' \\ b = q'_b \cdot g' \end{array}$$

$$\begin{array}{l} a - b = (q_a - q_b) \cdot g \\ b = q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{l} a = (q'_{a-b} + q'_b) \cdot g' \\ b = q'_b \cdot g' \end{array}$$

Das heißt  $g$  ist Teiler von  $a - b, b$  und  $g'$  ist Teiler von  $a, b$ .

Daraus folgt  $g \leq g'$  und  $g' \leq g$ , also  $g = g'$ .

# Eigenschaften

**(statische) Finitheit.** Die Beschreibung des Algorithmus besitzt endliche Länge. (↑**nichtuniforme Algorithmen**)

**(dynamische) Finitheit.** Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

**Terminiertheit.** Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen **terminierend**. (↑**Betriebssysteme**, ↑**reaktive Systeme**)

**Determiniertheit.** Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

**Determinismus.** Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

# Eigenschaften

**(statische) Finitheit.** Die Beschreibung des Algorithmus besitzt endliche Länge. (↑**nichtuniforme Algorithmen**)

**(dynamische) Finitheit.** Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

**Terminiertheit.** Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen **terminierend**. (↑**Betriebssysteme, reaktive Systeme**)

**Determiniertheit.** Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑**randomisierte Algorithmen, nicht-deterministische Algorithmen**)

**Determinismus.** Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑**randomisierte Algorithmen, nicht-deterministische Algorithmen**)

# Eigenschaften

**(statische) Finitheit.** Die Beschreibung des Algorithmus besitzt endliche Länge. (↑**nichtuniforme Algorithmen**)

**(dynamische) Finitheit.** Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

**Terminiertheit.** Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen **terminierend**. (↑**Betriebssysteme**, ↑**reaktive Systeme**)

**Determiniertheit.** Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

**Determinismus.** Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

# Eigenschaften

**(statische) Finitheit.** Die Beschreibung des Algorithmus besitzt endliche Länge. (↑**nichtuniforme Algorithmen**)

**(dynamische) Finitheit.** Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

**Terminiertheit.** Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen **terminierend**. (↑**Betriebssysteme**, ↑**reaktive Systeme**)

**Determiniertheit.** Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

**Determinismus.** Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

# Eigenschaften

**(statische) Finitheit.** Die Beschreibung des Algorithmus besitzt endliche Länge. (↑**nichtuniforme Algorithmen**)

**(dynamische) Finitheit.** Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

**Terminiertheit.** Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen **terminierend**. (↑**Betriebssysteme**, ↑**reaktive Systeme**)

**Determiniertheit.** Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

**Determinismus.** Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑**randomisierte Algorithmen**, ↑**nicht-deterministische Algorithmen**)

## Entscheidende Fragestellungen:

- ▶ **Darstellung** → Kapitel 2
- ▶ **Robustheit** und **Korrektheit** → Kapitel 4
- ▶ **Effizienz** und **Komplexität** → Kapitel 5
- ▶ **Entwurfstechniken** → Kapitel 6



# Definition Datenstruktur

## Definition Datenstruktur (nach Prof. Eckert)

Eine Datenstruktur ist eine

- ▶ **logische Anordnung** von Datenobjekten,
- ▶ die **Informationen repräsentieren**,
- ▶ den **Zugriff** auf die repräsentierte Information über **Operationen** auf Daten ermöglichen und
- ▶ die Information **verwalten**.

# Beispiel Datenstruktur

**Stapel** (oder Englisch: **Stack**), z.B. Pizza-Stapel

Operationen:

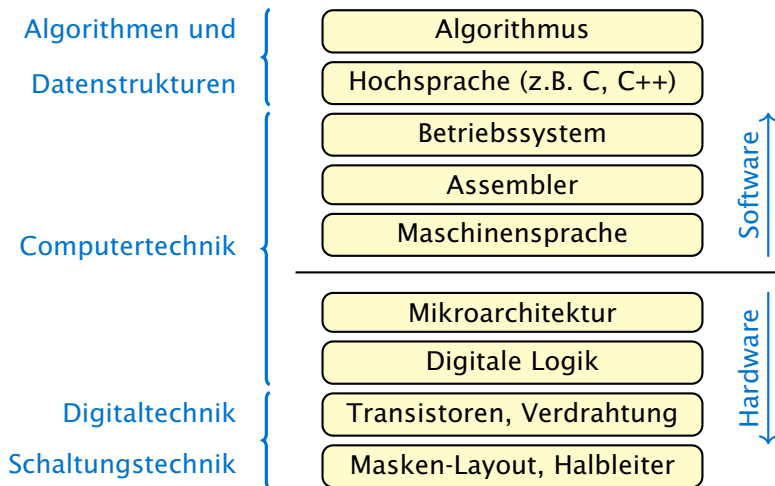
- ▶ Element auf Stapel legen – **push**
- ▶ Element von Stapel nehmen – **pop**

Operationen jeweils nur auf oberstem Element!

# Weitere Beispiele von Datenstrukturen

- ▶ **Felder, Listen, Stack, Queue** → Kapitel 3
- ▶ **Bäume, Graphen** → Kapitel 7, 8, 9

# Wie funktioniert ein Computer?



Schema nach Prof. Diepold: Grundlagen der Informatik.

# Einordnung Algorithmen und Datenstrukturen

## Beispiel-Problem Navigationssystem Auto

Finde kürzesten Weg von Berlin nach München.

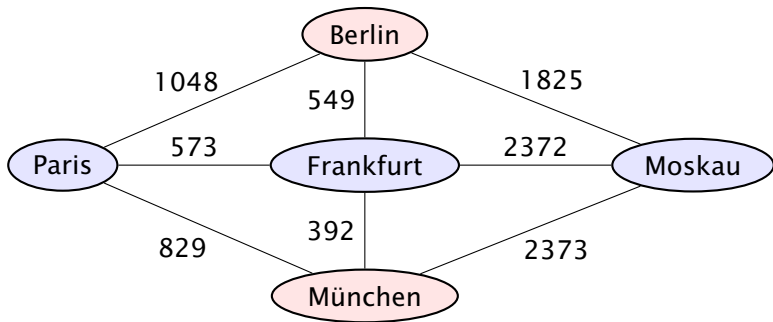


# Einordnung Algorithmen und Datenstrukturen

## Beispiel-Problem Navigationssystem Auto

Finde kürzesten Weg von Berlin nach München.

- **Datenstruktur:** gewichteter Graph (→ Kapitel 7)

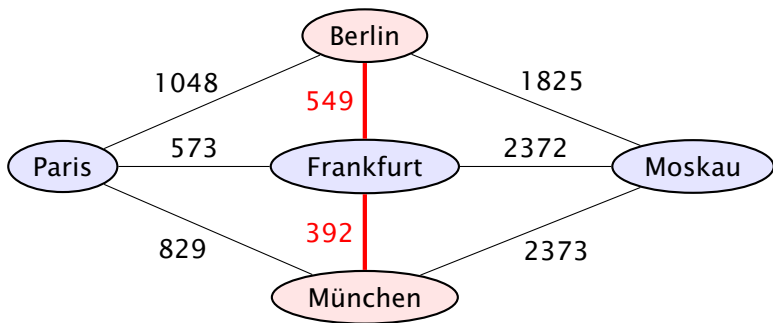


# Einordnung Algorithmen und Datenstrukturen

## Beispiel-Problem Navigationssystem Auto

Finde kürzesten Weg von Berlin nach München.

- ▶ **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- ▶ **Algorithmus:** kürzester Pfad (→ Kapitel 9)



# Einordnung Algorithmen und Datenstrukturen

## Beispiel-Problem Navigationssystem Auto

**Finde kürzesten Weg von Berlin nach München.**

- ▶ **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- ▶ **Algorithmus:** kürzester Pfad (→ Kapitel 9)
- ▶ **Algorithmus-Beschreibung:** Programmiersprache (z.B. C)



# Einordnung Algorithmen und Datenstrukturen

## Beispiel-Problem Navigationssystem Auto

**Finde kürzesten Weg von Berlin nach München.**

- ▶ **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- ▶ **Algorithmus:** kürzester Pfad (→ Kapitel 9)
- ▶ **Algorithmus-Beschreibung:** Programmiersprache (z.B. C)
- ▶ **Übersetzung in Maschinensprache:** Compiler (z.B. GCC)

# Einordnung Algorithmen und Datenstrukturen

## Beispiel-Problem Navigationssystem Auto

**Finde kürzesten Weg von Berlin nach München.**

- ▶ **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- ▶ **Algorithmus:** kürzester Pfad (→ Kapitel 9)
- ▶ **Algorithmus-Beschreibung:** Programmiersprache (z.B. C)
- ▶ **Übersetzung in Maschinensprache:** Compiler (z.B. GCC)
- ▶ **Aufruf des Programms:** Betriebssystem (z.B. Linux)

# Einordnung Algorithmen und Datenstrukturen

## Beispiel-Problem Navigationssystem Auto

**Finde kürzesten Weg von Berlin nach München.**

- ▶ **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- ▶ **Algorithmus:** kürzester Pfad (→ Kapitel 9)
- ▶ **Algorithmus-Beschreibung:** Programmiersprache (z.B. C)
- ▶ **Übersetzung in Maschinensprache:** Compiler (z.B. GCC)
- ▶ **Aufruf des Programms:** Betriebssystem (z.B. Linux)
- ▶ **Ausführung des Programms:** Computer (z.B. Laptop)

# Einordnung Algorithmen und Datenstrukturen

Algorithmen und  
Datenstrukturen



Algorithmus

Hochsprache (z.B. C)

Betriebssystem

Assembler

Maschinensprache

Software ↑

Mikroarchitektur

Digitale Logik

Transistoren, Verdrahtung

Masken-Layout, Halbleiter

Hardware ↓

*Schema nach Prof. Diepold: Grundlagen der Informatik.*

# Wie beschreibt man Algorithmen?

## Algorithmus: bestimme Maximum von zwei Zahlen

- ▶ Input: Zahlen  $a, b$
- ▶ Output: Zahl  $x = \max(a, b)$

**Problem:** präzise Beschreibung der Schritte

# Wie beschreibt man Algorithmen?

**Algorithmus: bestimme Maximum von zwei Zahlen**

- ▶ Input: Zahlen  $a, b$
- ▶ Output: Zahl  $x = \max(a, b)$

**Problem:** präzise Beschreibung der Schritte

**Lösung:** Pseudocode

Algorithmus:  $\max(a, b)$

Input:  $a, b$

$x = a$

Falls  $b > a$  dann

$x = b$

Ende Falls

Output:  $x$

# Darstellung von Algorithmen I

## Pseudocode

- ▶ informelle Veranschaulichung von Algorithmus
- ▶ nicht von Rechner ausführbar
- ▶ nicht standardisiert

Algorithmus:  $\max(a,b)$

Input:  $a,b$

$x=a$

Falls  $b>a$  dann

$x=b$

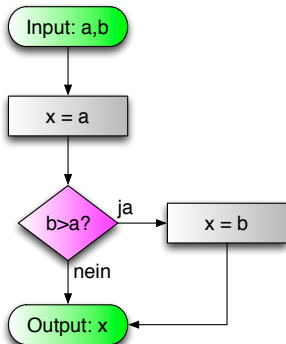
Ende Falls

Output:  $x$

# Darstellung von Algorithmen II

## Flussdiagramm

- ▶ graphische Darstellung als Ablaufdiagramm, nicht ausführbar
- ▶ normiert als DIN 66001

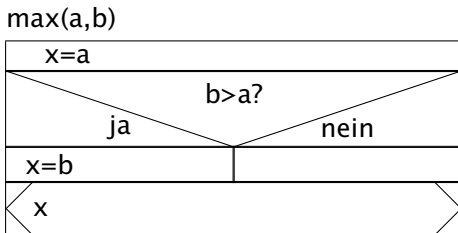




# Darstellung von Algorithmen III

## Struktogramm

- ▶ Diagramm zur Strukturdarstellung, nicht ausführbar
- ▶ eingeführt von Nassi/Shneiderman 1973, normiert als DIN 66261



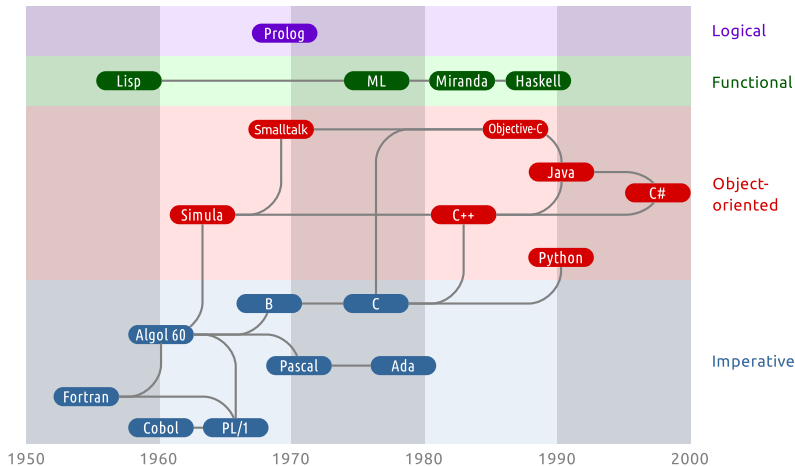
# Darstellung von Algorithmen IV

## Programmiersprache

- ▶ formale Sprache zur Beschreibung von Algorithmen
- ▶ fest definierte Syntax
- ▶ ein Compiler/Interpreter wandelt Programm in ausführbare Form für Rechner um
- ▶ Beispiele: Assembler, C, Java
  
- ▶ Algorithmus in C:

```
1 int max(int a, int b) {  
2     int x = a;  
3     if (b > a)  
4         x = b;  
5     return x;  
6 }
```

# Programmiersprachen Übersicht



Grafik von Alexandru Dului.

# Äquivalenz von Algorithmen-Beschreibungen

## Churchsche These

Alle „vernünftigen“ Definitionen von Algorithmen sind äquivalent.

- ▶ alle gängigen Programmiersprachen leisten dasselbe
- ▶ jeder Computer ist äquivalent
  
- ▶ formal: berechenbare Funktionen, formale Sprachen, Automaten, Turing-Maschinen

↑theoretische Informatik

# Äquivalenz von Algorithmen-Beschreibungen

## Churchsche These

Alle „vernünftigen“ Definitionen von Algorithmen sind äquivalent.

- ▶ alle gängigen Programmiersprachen leisten dasselbe
- ▶ jeder Computer ist äquivalent
  
- ▶ formal: berechenbare Funktionen, formale Sprachen, Automaten, Turing-Maschinen

↑theoretische Informatik

# Bausteine von Algorithmen

## Elementare Bausteine

„Normale“ Algorithmen lassen sich mit

### **vier elementaren Bausteinen**

darstellen:

1. Elementarer Verarbeitungsschritt (z.B. Zuweisung an Variable)
2. Sequenz (elementare Schritte nacheinander)
3. Bedingter Verarbeitungsschritt (z.B. if/else)
4. Wiederholung (z.B. while-Schleife)

# 1. Elementarer Verarbeitungsschritt

## Beispiele

- ▶ `a = a - b // weist Variable a den Wert a-b zu`
- ▶ `return a // liefert den Wert von a zurueck`

**Achtung:** manche Verarbeitungsschritte sehen elementar aus, sind es aber nicht!

- ▶ `sortiere Liste L // nicht elementar`
- ▶ `finde kuerzesten Pfad in G // nicht elementar`

## 2. Sequenz

Sequenz ist eine **Aneinanderreihung** von elementaren Verarbeitungsschritten

Abgrenzung der Schritte mittels **Semikolon (;)**

### Beispiel

- ▶ `x = 5; // Zuweisung von Wert 5 an Variable x`
- ▶ `x = x + 2; // Wert von x ist nun 7`

Um Ausnahmen zu vermeiden, wird Semikolon auch verwendet, wenn kein weiterer Schritt folgt



## 3. Bedingter Verarbeitungsschritt

Ausführung des Verarbeitungsschrittes nur wenn **Bedingung** erfüllt ist

### Beispiele:

- ▶ `if (a > b) // Bedingung wird in Klammern notiert`  
    `a = a - b;`
- ▶ `if (a > b)`  
    `a = a - b;`  
    `else // falls Bedingung nicht erfuehlt`  
    `b = b - a;`

**Einrückung** verdeutlicht logische Ebenen

## 3. Bedingter Verarbeitungsschritt

falls mehr als ein Verarbeitungsschritt bedingt ausgeführt werden soll, Markierung durch einen Block `{ ... }` mit geschweiften Klammern

### Beispiel

```
if (x == 0) {  
    x = 5;  
    x = x + 2;  
} // if Block ist hier zu Ende  
else {  
    x = x - 1;  
} // else Block ist hier zu Ende
```

auch einzelne Schritte können in einen Block gefasst werden

## 4. Wiederholung

wiederholte Ausführung von Verarbeitungsschritt/Block solange Bedingung erfüllt ist (auch **while Schleife** genannt)

### Beispiele

- ▶ `while (x != 0) // Bedingung in Klammern`  
    `x = x - 1;`
- ▶ `while (b > 0) { // Block fuer mehrere Schritte`  
    `if (a > b)`  
        `a = a - b;`  
    `else`  
        `b = b - a;`  
    `} // while Block ist hier zu Ende`

## 4. Wiederholung

Es gibt auch andere Schleifentypen: **do-while Schleife**:

```
▶ do {  
    x = x - 1;  
} while (x != 0); // Vorsicht by floats!!!
```

**for-Schleife**:

```
▶ for i=1 to 10  
    print(i); // gibt Wert von i aus
```

Achtung, Syntax der **for-Schleife** ist in C komplexer!

```
▶ for (i=1; i <= 10; i++) // echte C Syntax  
    print(i);
```

# Beispiel: Euklidischer Algorithmus

- ▶ Einrücken **oder** geschweifte Klammern `{, }` kennzeichnen **Blockstruktur**

```
1  euklid(a,b)
2      if (a == 0)
3          return b;
4      while (b > 0) {
5          if (a > b)
6              a = a - b;
7          else
8              b = b - a;
9      }
10     return a;
```

Euklidischer Algorithmus

# Beispiel: Euklidischer Algorithmus

- ▶ Einrücken **oder** geschweifte Klammern `{, }` kennzeichnen **Blockstruktur**
- ▶ In C so nicht möglich

```
1  euklid(a,b)
2      if (a == 0)
3          return b;
4      while (b > 0)
5          if (a > b)
6              a = a - b;
7          else
8              b = b - a;
9      return a;
```

Euklidischer Algorithmus

# Euklidischer Algorithmus

**Input:** Natürliche Zahlen  $a, b$

**Output:**  $\text{ggT}(a, b)$

1. Falls  $a = 0$  liefere  $b$  zurück
2. Solange  $b > 0$  wiederhole  
    Falls  $a > b$  setze  $a = a - b$   
    sonst setze  $b = b - a$
3. Liefere  $a$  zurück

# Euklidischer Algorithmus

**Input:** Natürliche Zahlen  $a, b$

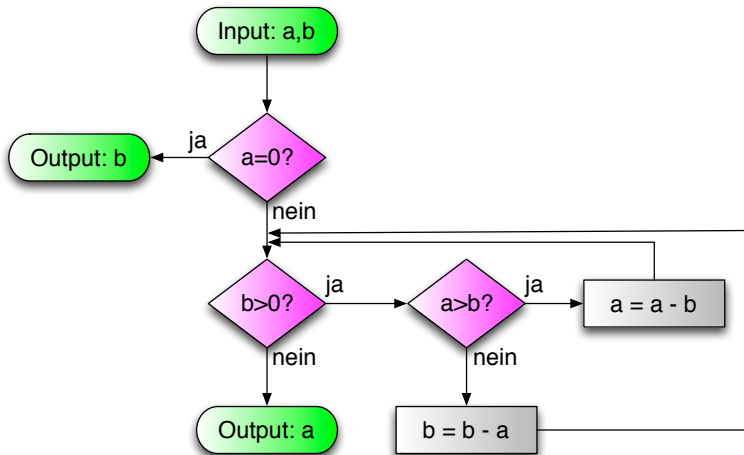
**Output:**  $\text{ggT}(a, b)$

1. Falls  $a = 0$  liefere  $b$  zurück
2. Solange  $b > 0$  wiederhole  
    Falls  $a > b$  setze  $a = a - b$   
    sonst setze  $b = b - a$
3. Liefere  $a$  zurück

→ das ist Pseudocode!

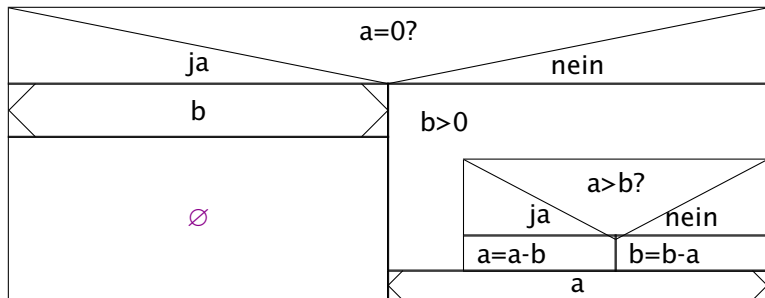


# Euklidischer Algorithmus als Flussdiagramm



# Euklidischer Algorithmus als Struktogramm

ggT(a,b)



# Euklidischer Algorithmus als Pseudocode

```
1 Input: natuerliche Zahlen a, b
2 Output: ggT(a,b)
3 euklid(a,b)
4     if (a == 0)
5         return b;
6     while (b > 0) { // Hauptschleife
7         if (a > b)
8             a = a - b;
9         else
10            b = b - a;
11    }
12    return a;
```

## Euklidischer Algorithmus

# Euklidischer Algorithmus als C

```
1 int ggT(int a, int b)
2 {
3     if (a==0)
4         return b;
5     while (b>0) {
6         if (a>b)
7             a = a - b;
8         else
9             b = b - a;
10    }
11    return a;
12 }
```

# Euklidischer Algorithmus als Python

```
1 def ggT(a, b):
2     if a == 0:
3         return b
4     while b > 0:
5         if a > b:
6             a = a - b
7         else:
8             b = b - a
9     return a
```

# Darstellung von Algorithmen in der Vorlesung

viele Möglichkeiten der Darstellung!

- ▶ alle vernünftigen Darstellungen sind äquivalent
- ▶ jede Darstellung hat Vor- und Nachteile

für die Vorlesung: **Pseudocode im C Stil**

Zusatzmaterial für viele Beispiele aus der Vorlesung:

- ▶ <http://www.brpreiss.com/books/opus7/>
- ▶ Beispiele in:
  - ▶ Python
  - ▶ C++
  - ▶ Java
  - ▶ C#
  - ▶ und vieles mehr...

# Beispiel: Fibonacci Zahlen

## Fibonacci Folge

Die **Fibonacci Folge** ist eine Folge natürlicher Zahlen

$f_1, f_2, f_3, \dots$ , für die gilt

$$f_n = f_{n-1} + f_{n-2} \quad \text{für } n \geq 3$$

mit Anfangswerten  $f_1 = 1, f_2 = 1$ .

- ▶ eingesetzt von Leonardo Fibonacci zur Beschreibung von Wachstum einer Kaninchenpopulation
- ▶ Folge lautet: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- ▶ berechenbar z.B. via Rekursion



# Beispiel: Fibonacci Funktion

**Input:** Index  $n$  der Fibonaccifolge

**Output:** Wert  $f_n$

```
fib(n)
  if (n == 1 || n == 2) {
    return 1;
  }
  else {
    // rekursiver Aufruf
    return fib(n-1) + fib(n-2);
  }
```



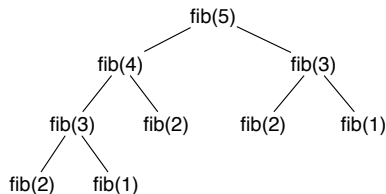
# Beispiel: Fibonacci Funktion

**Input:** Index  $n$  der Fibonaccifolge

**Output:** Wert  $f_n$

```
fib(n)
  if (n == 1 || n == 2) {
    return 1;
  }
  else {
    // rekursiver Aufruf
    return fib(n-1) + fib(n-2);
  }
```

Aufrufstruktur für fib(5):



# George Boole



Englischer Mathematiker (1815-1864)

## Boolesche Logik: Logik mit zwei Werten

Repräsentationen:

- ▶ 1 und 0
- ▶ W und F (in Englisch: T und F)
- ▶ L und O

Mengensymbol  $\mathbb{B}$

- ▶  $\mathbb{B} = \{0, 1\} = \{F, W\} = \{O, L\}$

# Logische Werte und Verknüpfungen

„Grundrechenarten“ mit logischen Werten:

**Konjunktion:**  $\wedge : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ ähnlich zu Multiplikation bei Zahlen
- ▶ auch bezeichnet als **UND** bzw. **AND**

**Disjunktion:**  $\vee : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ ähnlich zu Addition bei Zahlen
- ▶ auch bezeichnet als **ODER** bzw. **OR**

**Negation:**  $\neg : \mathbb{B} \rightarrow \mathbb{B}$

- ▶ auch bezeichnet als **NICHT** bzw. **NOT**

# Logische Werte und Verknüpfungen

„Grundrechenarten“ mit logischen Werten:

**Konjunktion:**  $\wedge : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ ähnlich zu Multiplikation bei Zahlen
- ▶ auch bezeichnet als **UND** bzw. **AND**

**Disjunktion:**  $\vee : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ ähnlich zu Addition bei Zahlen
- ▶ auch bezeichnet als **ODER** bzw. **OR**

**Negation:**  $\neg : \mathbb{B} \rightarrow \mathbb{B}$

- ▶ auch bezeichnet als **NICHT** bzw. **NOT**

**Wahrheitstabelle:**

$a$	$b$	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

# Logische Werte und Verknüpfungen

„Grundrechenarten“ mit logischen Werten:

**Konjunktion:**  $\wedge : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ ähnlich zu Multiplikation bei Zahlen
- ▶ auch bezeichnet als **UND** bzw. **AND**

**Disjunktion:**  $\vee : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ ähnlich zu Addition bei Zahlen
- ▶ auch bezeichnet als **ODER** bzw. **OR**

**Negation:**  $\neg : \mathbb{B} \rightarrow \mathbb{B}$

- ▶ auch bezeichnet als **NICHT** bzw. **NOT**

**Wahrheitstabelle:**

$a$	$b$	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

$a$	$b$	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

# Logische Werte und Verknüpfungen

„Grundrechenarten“ mit logischen Werten:

**Konjunktion:**  $\wedge : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ ähnlich zu Multiplikation bei Zahlen
- ▶ auch bezeichnet als **UND** bzw. **AND**

**Disjunktion:**  $\vee : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ ähnlich zu Addition bei Zahlen
- ▶ auch bezeichnet als **ODER** bzw. **OR**

**Negation:**  $\neg : \mathbb{B} \rightarrow \mathbb{B}$

- ▶ auch bezeichnet als **NICHT** bzw. **NOT**

**Wahrheitstabelle:**

$a$	$b$	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

$a$	$b$	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

$a$	$\neg a$
0	1
1	0

## Weitere Verknüpfungen I

**NAND:**  $\uparrow: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶  $a \uparrow b = \neg(a \wedge b)$
- ▶ mit NAND lassen sich NOT, OR, AND erzeugen

**NOR:**  $\downarrow: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶  $a \downarrow b = \neg(a \vee b)$
- ▶ mit NOR lassen sich ebenso NOT, OR, AND erzeugen

**XOR:**  $\oplus: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ auch **exklusiv oder** genannt
- ▶ erzeugbar aus  $\neg(a \wedge b) \wedge (a \vee b)$   
(siehe Übung)

**Wahrheitstabelle:**

$a$	$b$	$a \uparrow b$
0	0	1
0	1	1
1	0	1
1	1	0



# Weitere Verknüpfungen I

**NAND:**  $\uparrow: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶  $a \uparrow b = \neg(a \wedge b)$
- ▶ mit NAND lassen sich NOT, OR, AND erzeugen

**NOR:**  $\downarrow: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶  $a \downarrow b = \neg(a \vee b)$
- ▶ mit NOR lassen sich ebenso NOT, OR, AND erzeugen

**XOR:**  $\oplus: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ auch **exklusiv oder** genannt
- ▶ erzeugbar aus  $\neg(a \wedge b) \wedge (a \vee b)$   
(siehe Übung)

**Wahrheitstabelle:**

$a$	$b$	$a \uparrow b$
0	0	1
0	1	1
1	0	1
1	1	0

$a$	$b$	$a \downarrow b$
0	0	1
0	1	0
1	0	0
1	1	0

# Weitere Verknüpfungen I

**NAND:**  $\uparrow: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶  $a \uparrow b = \neg(a \wedge b)$
- ▶ mit NAND lassen sich NOT, OR, AND erzeugen

**NOR:**  $\downarrow: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶  $a \downarrow b = \neg(a \vee b)$
- ▶ mit NOR lassen sich ebenso NOT, OR, AND erzeugen

**XOR:**  $\oplus: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ auch **exklusiv oder** genannt
- ▶ erzeugbar aus  $\neg(a \wedge b) \wedge (a \vee b)$   
(siehe Übung)

**Wahrheitstabelle:**

$a$	$b$	$a \uparrow b$
0	0	1
0	1	1
1	0	1
1	1	0

$a$	$b$	$a \downarrow b$
0	0	1
0	1	0
1	0	0
1	1	0

$a$	$b$	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

## Weitere Verknüpfungen II

**Implikation:**  $\Rightarrow: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ oft verwendet für mathematische Sätze:  
„ $a$  impliziert  $b$ “, „aus  $a$  folgt  $b$ “
- ▶ Beispiel: „aus  $n < 3$  folgt  $n < 5$ “
- ▶ erzeugbar aus  $\neg a \vee b$

**Wahrheitstabelle:**

$a$	$b$	$a \Rightarrow b$
0	0	1
0	1	1
1	0	0
1	1	1

**Äquivalenz:**  $\Leftrightarrow: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ oft verwendet für mathematische Sätze:  
„ $a$  gilt genau dann, wenn  $b$  gilt“,  
„ $a$  und  $b$  sind äquivalent“
- ▶ Beispiel: „ $f$  ist bijektiv genau dann,  
wenn  $f$  injektiv und surjektiv ist“
- ▶ erzeugbar aus  $(a \wedge b) \vee (\neg a \wedge \neg b)$

## Weitere Verknüpfungen II

**Implikation:**  $\Rightarrow: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ oft verwendet für mathematische Sätze:  
„ $a$  impliziert  $b$ “, „aus  $a$  folgt  $b$ “
- ▶ Beispiel: „aus  $n < 3$  folgt  $n < 5$ “
- ▶ erzeugbar aus  $\neg a \vee b$

**Wahrheitstabelle:**

$a$	$b$	$a \Rightarrow b$
0	0	1
0	1	1
1	0	0
1	1	1

**Äquivalenz:**  $\Leftrightarrow: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- ▶ oft verwendet für mathematische Sätze:  
„ $a$  gilt genau dann, wenn  $b$  gilt“,  
„ $a$  und  $b$  sind äquivalent“
- ▶ Beispiel: „ $f$  ist bijektiv genau dann,  
wenn  $f$  injektiv und surjektiv ist“
- ▶ erzeugbar aus  $(a \wedge b) \vee (\neg a \wedge \neg b)$

$a$	$b$	$a \Leftrightarrow b$
0	0	1
0	1	0
1	0	0
1	1	1

# Rangfolge und Rechenregeln

## Rangfolge:

- ▶ NICHT vor UND
- ▶ UND vor ODER

## Beispiel

$$\neg 0 \vee 1 \wedge 0 = (\neg 0) \vee (1 \wedge 0) = 1 \vee 0 = 1$$

## De Morgan-Gesetze:

- ▶  $\neg(a \wedge b) = \neg a \vee \neg b$
- ▶  $\neg(a \vee b) = \neg a \wedge \neg b$

# Logische Ausdrücke in Pseudocode und C

- ▶ logische Variablen: `bool a,b;`
- ▶ logische Werte: `true` und `false`
- ▶ NOT Operator: `!a`
- ▶ AND Operator: `a && b`
- ▶ OR Operator: `a || b`

## Beispiele:

- ▶ `( (2 == 2) && (3 < 1) )`  
ergibt `(true && false)`, also `false`
- ▶ `( !(2 == 2) || (3 > 1) )`  
ergibt `(false || true)`, also `true`
- ▶ Kurzform für `!(2 == 2)` ist `(2 != 2)`

# Logische Ausdrücke in Pseudocode und C

- ▶ logische Variablen: `bool a,b;`
- ▶ logische Werte: `true` und `false`
- ▶ NOT Operator: `!a`
- ▶ AND Operator: `a && b`
- ▶ OR Operator: `a || b`

## Beispiele:

- ▶ `( (2 == 2) && (3 < 1) )`  
ergibt `(true && false)`, also `false`
- ▶ `( !(2 == 2) || (3 > 1) )`  
ergibt `(false || true)`, also `true`
- ▶ Kurzform für `!(2 == 2)` ist `(2 != 2)`

# Logische Ausdrücke in Pseudocode und C

- ▶ logische Variablen: `bool a,b;`
- ▶ logische Werte: `true` und `false`
- ▶ NOT Operator: `!a`
- ▶ AND Operator: `a && b`
- ▶ OR Operator: `a || b`

## Beispiele:

- ▶ `( (2 == 2) && (3 < 1) )`  
ergibt `(true && false)`, also `false`
- ▶ `( !(2 == 2) || (3 > 1) )`  
ergibt `(false || true)`, also `true`
- ▶ Kurzform für `!(2 == 2)` ist `(2 != 2)`



# Logische Ausdrücke in Pseudocode und C

- ▶ logische Variablen: `bool a,b;`
- ▶ logische Werte: `true` und `false`
- ▶ NOT Operator: `!a`
- ▶ AND Operator: `a && b`
- ▶ OR Operator: `a || b`

## Beispiele:

- ▶ `( (2 == 2) && (3 < 1) )`  
ergibt `(true && false)`, also `false`
- ▶ `( !(2 == 2) || (3 > 1) )`  
ergibt `(false || true)`, also `true`
- ▶ Kurzform für `!(2 == 2)` ist `(2 != 2)`

# Was sind primitive Datentypen?

## Primitive Datentypen

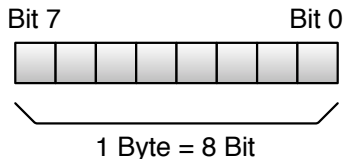
Wir bezeichnen grundlegende, in Programmiersprachen eingebaute Datentypen als **primitive Datentypen**.

Durch Kombination von primitiven Datentypen lassen sich **zusammengesetzte Datentypen** bilden.

Beispiele für primitive Datentypen in C:

- ▶ **int** für ganze Zahlen
- ▶ **float** für floating point Zahlen
- ▶ **bool** für logische Werte

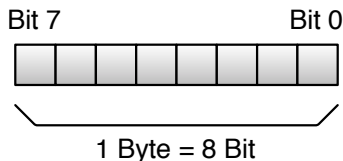
# Bits und Bytes



**Bytes** als Maßeinheit für Speichergrößen (nach IEC, **traditionell**):

- ▶  $2^{10}$  Bytes = 1024 Bytes = 1 KiB, ein **Kilo Byte** (Kibi Byte)
- ▶  $2^{20}$  Bytes = 1 MiB, ein **Mega Byte** (bzw. MebiByte)
- ▶  $2^{30}$  Bytes = 1 GiB, ein **Giga Byte** (bzw. GibiByte)
- ▶  $2^{40}$  Bytes = 1 TiB, ein **Tera Byte** (bzw. TebiByte)
- ▶  $2^{50}$  Bytes = 1 PiB, ein **Peta Byte** (bzw. PebiByte)
- ▶  $2^{60}$  Bytes = 1 EiB, ein **Exa Byte** (bzw. ExbiByte)

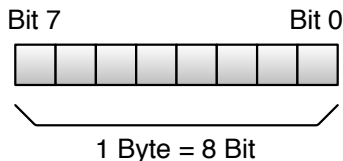
# Bits und Bytes



**Bytes** als Maßeinheit für Speichergrößen (nach IEC, **metrisch**):

- ▶  $10^3$  Bytes = 1000 Bytes = 1 kB, ein **kilo Byte** (großes B)
- ▶  $10^6$  Bytes = 1 MB, ein **Mega Byte**
- ▶  $10^9$  Bytes = 1 GB, ein **Giga Byte**
- ▶  $10^{12}$  Bytes = 1 TB, ein **Tera Byte**
- ▶  $10^{15}$  Bytes = 1 PB, ein **Peta Byte**
- ▶  $10^{18}$  Bytes = 1 EB, ein **Exa Byte**

# Bits und Bytes



**Bytes** als Maßeinheit für Speichergrößen (nach IEC, **metrisch**):

- ▶  $10^3$  Bytes = 1000 Bytes = 1 kB, ein **kilo Byte** (großes B)
- ▶  $10^6$  Bytes = 1 MB, ein **Mega Byte**
- ▶  $10^9$  Bytes = 1 GB, ein **Giga Byte**
- ▶  $10^{12}$  Bytes = 1 TB, ein **Tera Byte**
- ▶  $10^{15}$  Bytes = 1 PB, ein **Peta Byte**
- ▶  $10^{18}$  Bytes = 1 EB, ein **Exa Byte**

**Hinweis:** auch Bits werden als Maßangabe verwendet, z.B. 16 Mbit oder 16 Mb (kleines b).

# Primitive Datentypen in C-ähnlichen Sprachen

Wir betrachten im Detail **primitive Datentypen** für:

1. natürliche Zahlen (*unsigned integers*)
2. ganze Zahlen (*signed integers*)
3. floating point Zahlen (*floats*)

# Zahldarstellung

## Dezimalsystem:

- ▶ Basis  $b = 10$
- ▶ Koeffizienten  $c_n \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- ▶ Beispiel:  $123_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$

## Binärsystem:

- ▶ Basis  $b = 2$
- ▶ Koeffizienten  $c_n \in \{0, 1\}$
- ▶ Beispiel:  $1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{10}$

# Zahldarstellung

## Dezimalsystem:

- ▶ Basis  $b = 10$
- ▶ Koeffizienten  $c_n \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- ▶ Beispiel:  $123_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$

## Binärsystem:

- ▶ Basis  $b = 2$
- ▶ Koeffizienten  $c_n \in \{0, 1\}$
- ▶ Beispiel:  $1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{10}$



# Zahldarstellung

## Oktalsystem:

- ▶ Basis  $b = 8 (= 2^3)$
- ▶ Koeffizienten  $c_n \in \{0, 1, 2, 3, 4, 5, 6, 7\}$
- ▶ Beispiel:  $173_8 = 1 \cdot 8^2 + 7 \cdot 8^1 + 3 \cdot 8^0 = 123_{10}$

## Hexadezimalsystem:

- ▶ Basis  $b = 16 (= 2^4)$
- ▶ Koeffizienten  $c_n \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$
- ▶ Beispiel:  $7B_{16} = 7 \cdot 16^1 + B \cdot 16^0 = 123_{10}$

# Wie viele Ziffern pro Zahl?

## Problem

Gegeben Zahl  $z \in \mathbb{N}$ , wie viele Ziffern  $m$  werden bezüglich Basis  $b$  benötigt?

**Lösung:**  $m = \lfloor \log_b(z) \rfloor + 1$

**Erläuterung:** ( $a \in \mathbb{R}$ )

- ▶  $\lfloor a \rfloor = \text{floor}(a) =$  größte ganze Zahl kleiner gleich  $a$
- ▶  $\lceil a \rceil = \text{ceil}(a) =$  kleinste ganze Zahl größer gleich  $a$

$$a - 1 < \lfloor a \rfloor \leq a \leq \lceil a \rceil < a + 1$$

- ▶  $\log_b(z) = \frac{\ln(z)}{\ln(b)}$ , wobei „ln“ der natürliche Logarithmus ist

# Wie viele Ziffern pro Zahl?

**Lösung:**  $m = \lfloor \log_x(z) \rfloor + 1$

**Beispiele:**  $z = 123$

- ▶ Basis  $b = 10$ :  $m = \lfloor \log_{10}(123) \rfloor + 1 = \lfloor 2.0899\dots \rfloor + 1 = 3$
- ▶ Basis  $b = 2$ :  $m = \lfloor \log_2(123) \rfloor + 1 = \lfloor 6.9425\dots \rfloor + 1 = 7$
- ▶ Basis  $b = 8$ :  $m = \lfloor \log_8(123) \rfloor + 1 = \lfloor 2.3141\dots \rfloor + 1 = 3$
- ▶ Basis  $b = 16$ :  $m = \lfloor \log_{16}(123) \rfloor + 1 = \lfloor 1.7356\dots \rfloor + 1 = 2$

# Größte Zahl pro Anzahl Ziffern?

## Problem

Gegeben Basis  $b$  und  $m$  Ziffern, was ist die größte darstellbare Zahl?

**Lösung:**  $z_{max} = x^m - 1$

## Beispiele:

- ▶  $b = 2, m = 4: z_{max} = 2^4 - 1 = 15 = 1111_2$
- ▶  $b = 2, m = 8: z_{max} = 2^8 - 1 = 255 = 11111111_2$
- ▶  $b = 16, m = 2: z_{max} = 16^2 - 1 = 255 = FF_{16}$

# Natürliche Zahlen in C-ähnlichen Sprachen

## Natürliche Zahlen

In Computern verwendet man **Binärdarstellung** mit einer fixen Anzahl Ziffern (genannt **Bits**).

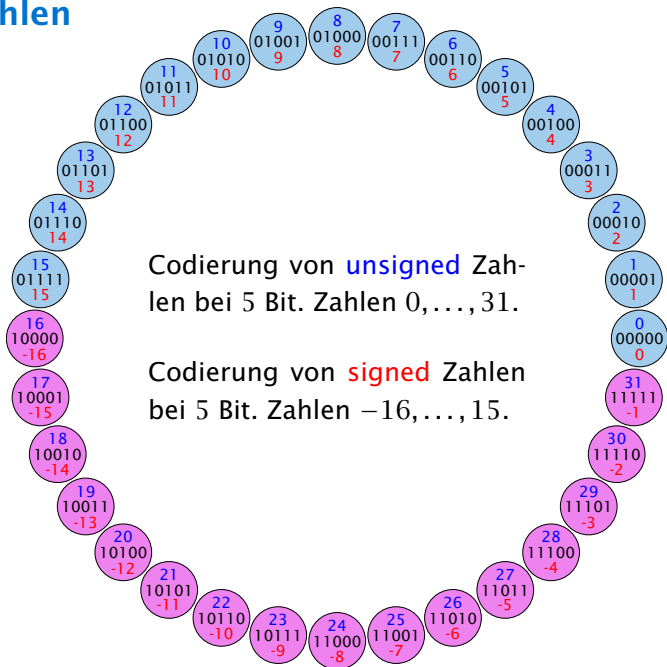
Die **primitiven Datentypen** für **natürliche Zahlen** sind:

- ▶ **8 Bits** (ein **Byte**), darstellbare Zahlen:  $\{0, \dots, 255\}$   
in C: **unsigned char**
- ▶ **16 Bits**, darstellbare Zahlen:  $\{0, \dots, 65\,535\}$   
in C: **unsigned short**
- ▶ **32 Bits**, darstellbare Zahlen:  $\{0, \dots, 4\,294\,967\,295\}$   
in C: **unsigned long**
- ▶ **64 Bits**, darstellbare Zahlen:  $\{0, \dots, 2^{64} - 1\}$   
in C: **unsigned long long**

# Negative Zahlen



# Negative Zahlen



Codierung von **unsigned** Zahlen bei 5 Bit. Zahlen 0, ..., 31.

Codierung von **signed** Zahlen bei 5 Bit. Zahlen -16, ..., 15.

# Negative Zahlen

## Bitfolge

$$x = \langle x_{n-1}, \dots, x_0 \rangle$$

$$\begin{array}{c} \xrightarrow{f} \\ \xleftarrow{f^{-1}} \end{array}$$

## Zahl

$$\sum_{i=0}^{n-1} x_i 2^i$$

$$x = \langle x_{n-1}, \dots, x_0 \rangle$$

$$\begin{array}{c} \xrightarrow{f_{\text{ZK}}} \\ \xleftarrow{f_{\text{ZK}}^{-1}} \end{array}$$

$$-x_{n-1} 2^n + \sum_{i=0}^{n-1} x_i 2^i$$



# Negative Zahlen

## Definition

In **2-Komplement Darstellung** mit  $n$  bits repräsentiert die Bitfolge

$$x = \langle x_{n-1}, x_{n-2}, \dots, x_1, x_0 \rangle$$

die Zahl  $f_{\text{ZK}}(x) = -x_{n-1}2^n + \sum_{i=0}^{n-1} x_i 2^i$ .

## Beobachtungen

- ▶ Zahlen mit  $x_{n-1} = 1$  sind negativ; andere positiv (Vorzeichenbit)
- ▶ positive Zahlen:  $0, \dots, 2^{n-1} - 1$   
negative Zahlen:  $-1, \dots, -2^{n-1}$
- ▶  $f_{\text{ZK}}(x) = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$ .

# Negative Zahlen

## Vorzeichenwechsel

Sei  $x = \langle x_{n-1}, \dots, x_0 \rangle$  ein Bitfolge mit  $\langle x_{n-2}, \dots, x_0 \rangle \neq \langle 0, \dots, 0 \rangle$ .

Die Repräsentation für die Zahl  $-f_{\text{ZK}}(x)$  im **2er Komplement** (d.h.  $f_{\text{ZK}}^{-1}(-f_{\text{ZK}}(x))$ ) erhält man durch

$$f^{-1}(f(\bar{x}) + 1)$$

wobei  $\bar{x} = \langle \bar{x}_{n-1}, \dots, \bar{x}_0 \rangle$  die invertierte Bitfolge bezeichnet.

D.h. man invertiert die Bitfolge und addiert **1** auf die sich ergebende Zahl.

# Negative Zahlen

## Beweis

1. Fall:  $x_{n-1} = 1$ , d.h.  $f_{ZK}(x)$  negativ

$$\begin{aligned} -f_{ZK}(x) &= 2^n - \sum_{i=0}^{n-1} x_i 2^i = 1 + \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} x_i 2^i \\ &= 1 + \sum_{i=0}^{n-1} (1 - x_i) 2^i = 1 + \sum_{i=0}^{n-1} \bar{x}_i 2^i \\ &= 1 + f(\bar{x}) \end{aligned}$$

Da  $1 + f(\bar{x}) < 2^{n-1}$  liefert Anwendung von  $f^{-1}$  oder  $f_{ZK}^{-1}$  die gleiche Bitfolge.

# Negative Zahlen

## Beweis

2. Fall:  $x_{n-1} = 0$ , d.h.  $f_{ZK}(x)$  strikt positiv

$$\begin{aligned} -f_{ZK}(x) &= -\sum_{i=0}^{n-1} x_i 2^i = \sum_{i=0}^{n-1} (1 - x_i) 2^i - \sum_{i=0}^{n-1} 2^i - 1 + 1 \\ &= 1 + \sum_{i=0}^{n-1} \bar{x}_i 2^i - 2^n = 1 + f(\bar{x}) - 2^n \end{aligned}$$

Für eine Zahl  $z$  die das höchstwertige Bit  $n - 1$  gesetzt hat gilt  $f_{ZK}^{-1}(z - 2^n) = f^{-1}(z)$ . Dies gilt für  $1 + f(\bar{x})$ .

# Negative Zahlen

## Definition

Die Restklasse  $[a]_m$  enthält alle  $z \in \mathbb{Z}$  die bei Division durch  $m$  den gleichen Rest lassen.

$a$  heißt Repräsentant der Restklasse. Eine Restklasse hat viele verschiedene Repräsentanten.

Für eine Restklasse  $M \subseteq \mathbb{Z}$  nennen wir  $a \in M$  mit  $0 \leq a < m$  den Standardrepräsentanten der Restklasse.

## Beispiel

►  $[2]_8 = [42]_8 = [-78]_8$

# Negative Zahlen

## Rechnen mit Restklassen

Man kann mit Restklassen rechnen. Die Multiplikation / Addition / Subtraktion etc. wird **repräsentantenweise** ausgeführt. **Die Wahl des Repräsentanten ist unwichtig!!!!!!!!!!**

## Beispiele:

- ▶  $[2]_8 \cdot [7]_8 = [2 \cdot 7]_8 = [6]_8$
- ▶  $[-6]_8 \cdot [23]_8 = [-6 \cdot 23]_8 = [-138]_8 = [-18]_8 = [6]_8$
- ▶  $[7]_8 + [8]_8 = [15]_8 = [-1]_8$

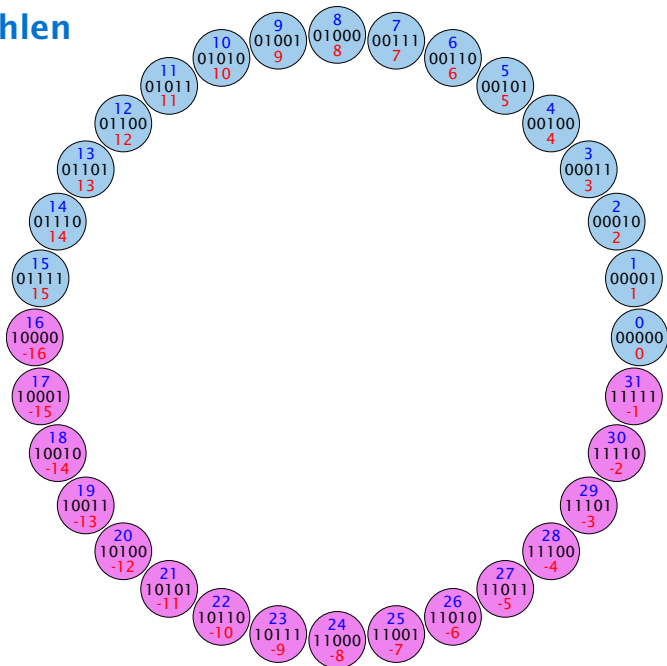
# Negative Zahlen

Die Hardware Implementierung von Addition / Multiplikation etc. implementiert eigentlich eine Operation auf Restklassen modulo  $2^n$ , wobei  $n$  der Bitlänge entspricht.

Im Prinzip wird eine Operation (Addition / Subtraktion / Multiplikation / Ganzzahldivision) ausgeführt, und dann werden überzählige Bits verworfen (d.h., dass Ergebnis wird modulo  $2^n$  genommen).

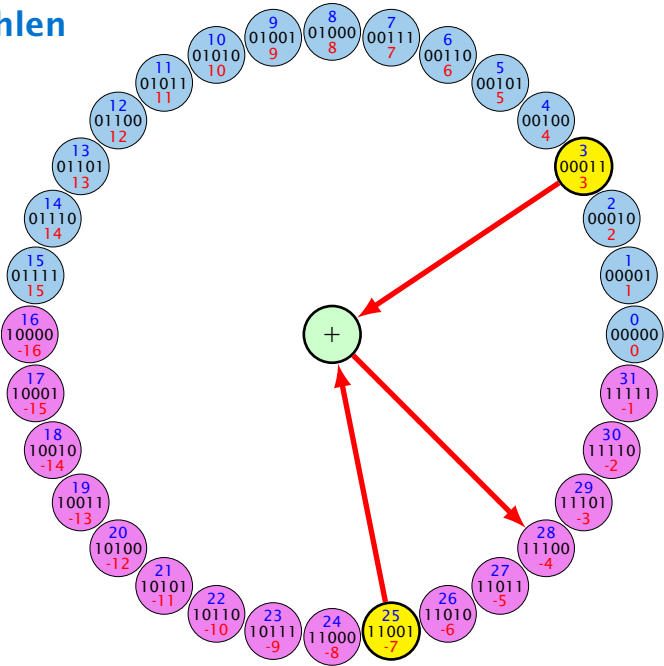
Durch das Verwenden des 2er-Komplements kann man für signed und unsigned Datentypen, (im wesentlichen) die gleiche Hardware benutzen.

# Negative Zahlen

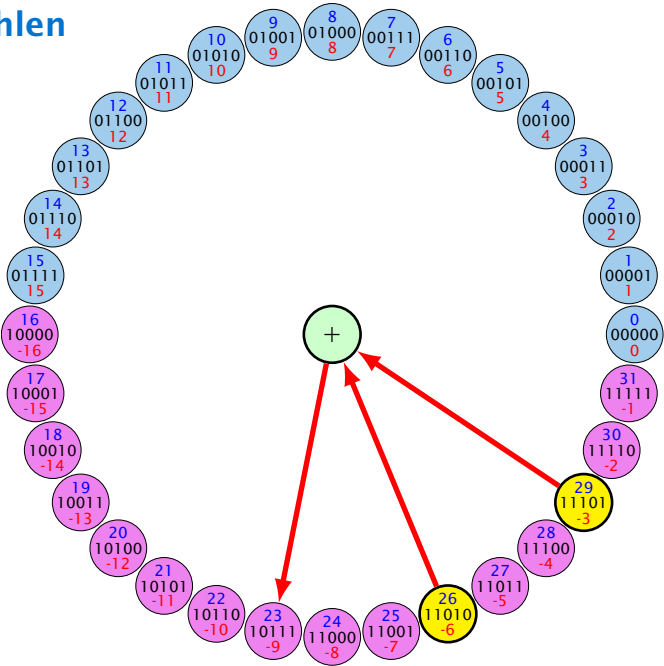




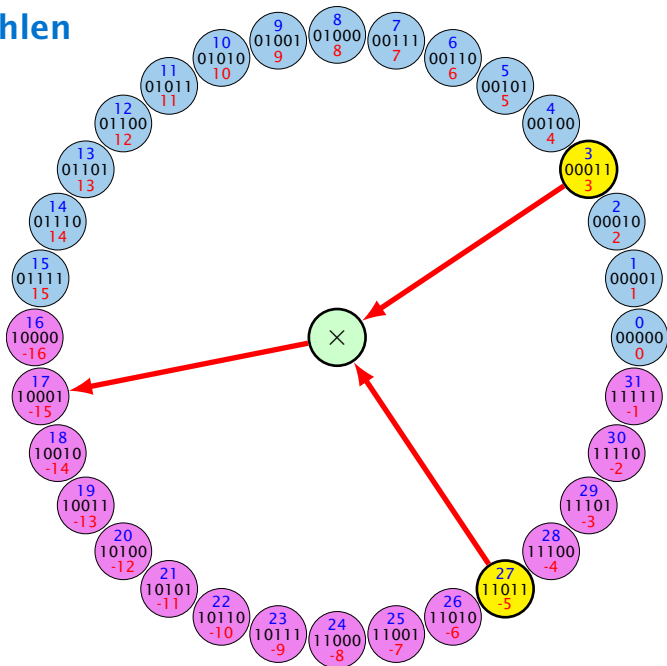
# Negative Zahlen



# Negative Zahlen



# Negative Zahlen



# Ganze Zahlen in C-ähnlichen Sprachen

## Ganze Zahlen:

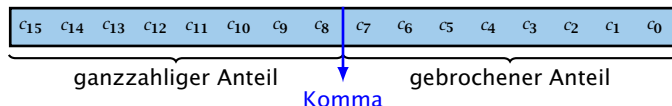
Die **primitiven Datentypen** für **ganze Zahlen** sind:

- ▶ **8 Bits:** unsigned char {0,...,255}  
signed char {-128,...,127}
- ▶ **16 Bits:** unsigned short {0,...,65535}  
signed short {-32768,...,32767}
- ▶ **32 Bits:** unsigned long {0,..., $2^{32} - 1$ }  
signed long {- $2^{31}$ ,..., $2^{31} - 1$ }
- ▶ **64 Bits:** unsigned long long {0,..., $2^{64} - 1$ }  
signed long long {- $2^{63}$ ,..., $2^{63} - 1$ }
- ▶ **signed** kann weggelassen werden (ausser bei **char**!)
- ▶ **unsigned int** und **signed int** sind je nach System 16, 32 oder 64 Bit

# Rationale Zahlen I

**Festkommadarstellung:** Komma an fester Stelle in Zahl

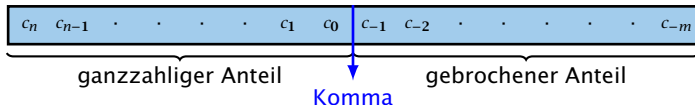
Beispiel mit  $n = 16$ :



Nachteile:

- ▶ weniger große Zahlen darstellbar
- ▶ feste Genauigkeit der Nachkommastellen

# Rationale Zahlen II



**Interpretation** für  $r \in \mathbb{Q}$ :

$$r = c_n \cdot 2^n + \dots + c_0 \cdot 2^0 + c_{-1} 2^{-1} + \dots + c_{-m} \cdot 2^{-m}$$

mit  $n + 1$  Vorkomma- und  $m$  Nachkommaziffern

**Beispiel:**

$$\begin{aligned} 11.01_2 &= 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} \\ &= 2 + 1 + 0 + \frac{1}{4} = 3.25_{10} \end{aligned}$$

# Floating Point Zahlen I

**Wissenschaftliche Notation:**  $x = a \cdot 10^b$  für  $x \in \mathbb{R}$ , wobei:

- ▶  $a \in \mathbb{R}$  mit  $1 \leq |a| < 10$
- ▶  $b \in \mathbb{Z}$

## Beispiele:

- ▶  $-2.7315 \cdot 10^2$  °C      absoluter Nullpunkt
- ▶  $1.5 \cdot 10^9$  Hz              Taktfrequenz A8X Prozessor

## Drei Bestandteile:

- ▶ Vorzeichen
- ▶ Mantisse  $|a|$  (bestimmt die Genauigkeit)
- ▶ Exponent  $b$  (bestimmt Größe des Wertebereichs)

**Problem:** bei fester Länge der Mantisse (z.B. 3 Ziffern)

- ▶ zwischen  $1.23 \cdot 10^4 = 12300$  und  $1.24 \cdot 10^4 = 12400$  keine Zahl darstellbar!

# Floating Point Zahlen II



- ▶ wissenschaftliche Darstellung mit Basis 2

$$f = (-1)^V \cdot (1 + M) \cdot 2^{E-bias}$$

- ▶ Vorzeichen Bit  $V$
- ▶ Mantisse  $M$  hat immer die Form  $1.abc$ , also wird erste Stelle weggelassen („hidden bit“)
- ▶ Exponent  $E$  wird vorzeichenlos abgespeichert, verschoben um  $bias$ 
  - ▶ bei 32 bit:  $bias = 127$ , bei 64 bit:  $bias = 1023$



# Floating Point Zahlen III

## Übliche Floating Point Formate:

<i>Bit</i>	<i>Vorz.</i>	<i>Exp.</i>	<i>Mant.</i>	<i>Dezimal stellen</i>	<i>Bereich</i>
32	1 Bit	8 Bit	23 Bit	~ 7	$\pm 2 \cdot 10^{-38}$ bis $\pm 2 \cdot 10^{38}$
64	1 Bit	11 Bit	52 Bit	~ 15	$\pm 2 \cdot 10^{-308}$ bis $\pm 2 \cdot 10^{308}$
80	1 Bit	15 Bit	64 Bit	~ 19	$\pm 1 \cdot 10^{-4932}$ bis $\pm 1 \cdot 10^{4932}$

In C:

`float` (32 Bit), `double` (64 Bit), `long double` (80 Bit)

# Vorsicht mit Floating Point!

Floating Point Zahlen sind bequem, aber **Vorsicht!**

- ▶ Viele Dezimalzahlen haben keine Floating Point Darstellung
  - ▶ Beispiel:  $0.1_{10} = 0.0001100110011\dots_2$  (periodisch)
- ▶ Durch feste Länge der Mantisse sind ebenfalls viele Zahlen nicht darstellbar
  - ▶ Beispiel: mit 3 Ziffern Mantisse ist zwischen  $1.23 \cdot 10^4 = 12300$  und  $1.24 \cdot 10^4 = 12400$  keine Zahl darstellbar!
- ▶ Kritisch sind Vergleiche von Floating Point Zahlen
  - ▶ Beispiel:  $(0.1 + 0.2 == 0.3)$  ist meist **FALSE!**
- ▶ Zins-Berechnungen und dergleichen **NIE** mit Floating Point Zahlen!
  - ▶ Stattdessen: spezielle Bibliotheken wie GMP

# Vorsicht mit Floating Point!

Floating Point Zahlen sind bequem, aber **Vorsicht!**

- ▶ Viele Dezimalzahlen haben keine Floating Point Darstellung
  - ▶ Beispiel:  $0.1_{10} = 0.0001100110011\dots_2$  (periodisch)
- ▶ Durch feste Länge der Mantisse sind ebenfalls viele Zahlen nicht darstellbar
  - ▶ Beispiel: mit 3 Ziffern Mantisse ist zwischen  $1.23 \cdot 10^4 = 12300$  und  $1.24 \cdot 10^4 = 12400$  keine Zahl darstellbar!
- ▶ Kritisch sind Vergleiche von Floating Point Zahlen
  - ▶ Beispiel:  $(0.1 + 0.2 == 0.3)$  ist meist **FALSE!**
- ▶ Zins-Berechnungen und dergleichen **NIE** mit Floating Point Zahlen!
  - ▶ Stattdessen: spezielle Bibliotheken wie GMP

# Vorsicht mit Floating Point!

Floating Point Zahlen sind bequem, aber **Vorsicht!**

- ▶ Viele Dezimalzahlen haben keine Floating Point Darstellung
  - ▶ Beispiel:  $0.1_{10} = 0.0001100110011\dots_2$  (periodisch)
- ▶ Durch feste Länge der Mantisse sind ebenfalls viele Zahlen nicht darstellbar
  - ▶ Beispiel: mit 3 Ziffern Mantisse ist zwischen  $1.23 \cdot 10^4 = 12300$  und  $1.24 \cdot 10^4 = 12400$  keine Zahl darstellbar!
- ▶ Kritisch sind Vergleiche von Floating Point Zahlen
  - ▶ Beispiel:  $(0.1 + 0.2 == 0.3)$  ist meist **FALSE!**
- ▶ Zins-Berechnungen und dergleichen **NIE** mit Floating Point Zahlen!
  - ▶ Stattdessen: spezielle Bibliotheken wie GMP

# Vorsicht mit Floating Point!

Floating Point Zahlen sind bequem, aber **Vorsicht!**

- ▶ Viele Dezimalzahlen haben keine Floating Point Darstellung
  - ▶ Beispiel:  $0.1_{10} = 0.0001100110011\dots_2$  (periodisch)
- ▶ Durch feste Länge der Mantisse sind ebenfalls viele Zahlen nicht darstellbar
  - ▶ Beispiel: mit 3 Ziffern Mantisse ist zwischen  $1.23 \cdot 10^4 = 12300$  und  $1.24 \cdot 10^4 = 12400$  keine Zahl darstellbar!
- ▶ Kritisch sind Vergleiche von Floating Point Zahlen
  - ▶ Beispiel:  $(0.1 + 0.2 == 0.3)$  ist meist **FALSE!**
- ▶ Zins-Berechnungen und dergleichen **NIE** mit Floating Point Zahlen!
  - ▶ Stattdessen: spezielle Bibliotheken wie GMP

# Definition Datenstruktur

## Definition Datenstruktur (nach Prof. Eckert)

Eine Datenstruktur ist eine

- ▶ **logische Anordnung** von Datenobjekten,
- ▶ die **Informationen** repräsentieren,
- ▶ den **Zugriff** auf die repräsentierte Information über **Operationen** auf Daten ermöglichen und
- ▶ die Information **verwalten**.

Zwei Hauptbestandteile:

- ▶ **Datenobjekte**  
z.B. definiert über primitive Datentypen
- ▶ **Operationen** auf den Objekten  
z.B. definiert als Funktionen

# Primitive Datentypen in C

Natürliche Zahlen, z.B. `unsigned short`, `unsigned long`

- ▶ Wertebereich: bei  $n$  Bit von 0 bis  $2^n - 1$
- ▶ Operationen: `+`, `-`, `*`, `/`, `%`, `<`, `==`, `!=`, `>`

Ganze Zahlen, z.B. `int`, `long`

- ▶ Wertebereich: bei  $n$  Bit von  $-2^{n-1}$  bis  $2^{n-1} - 1$
- ▶ Operationen: `+`, `-`, `*`, `/`, `%`, `<`, `==`, `!=`, `>`

Floating Point Zahlen, z.B. `double`, `float`

- ▶ Wertebereich: abhängig von Größe
- ▶ Operationen: `+`, `-`, `*`, `/`, `<`, `==`, `!=`, `>`

Logische Werte, `bool`

- ▶ Wertebereich: `true`, `false`
- ▶ Operationen: `&&`, `||`, `!`, `==`, `!=`

# Definition Feld

## Definition Feld

Ein **Feld**  $F$  ist eine Folge von  $n$  Datenelementen  $(d_i)_{i=1,\dots,n}$ ,

$$F = d_1, d_2, \dots, d_n$$

mit  $n \in \mathbb{N}_0$ .

Die Datenelemente  $d_i$  sind beliebige Datentypen (z.B. primitive).

## Beispiele:

- ▶  $F$  sind die natürlichen Zahlen von 1 bis 10, aufsteigend geordnet:

$$F = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

- ▶ Ist  $n = 0$ , so ist das Feld leer.



# Definition Feld

## Operationen

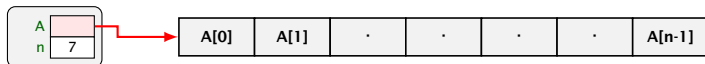
- ▶ `F.initialize()`: initialisiere leeres Feld `A`
- ▶ `F.elementAt(i)`: Zugriff auf `i`-tes Element von `A`.  
`i` muss zwischen `1` und `F.size()` liegen.
- ▶ `F.insert(d, i)`: füge Element `d` an Position `i` in Feld `A` ein.  
`i` muss zwischen `1` und `F.size()+1` liegen
- ▶ `F.erase(i)`: entferne `i`-tes Element aus Feld `A`.
- ▶ `F.size()`: gibt die Anzahl der Elemente des Feldes zurück

Ein abstrakter Datentyp wird im wesentlichen durch die auf ihn anwendbaren Operationen definiert.

# Feld als Array

## Repräsentation von Feld durch Array der Länge $n$

- ▶ Datenelemente werden in Array gespeichert
- ▶ einfacher Zugriff über index-operator ( $A[i]$ )
- ▶ Hinzufügen/Löschen schwierig...



**Achtung:** Indizierung des Arrays startet bei 0!  
 $A[i]$  enthält Element  $i+1$  des Feldes

# Eigenschaften von Arrays

Feld  $F$  mit Länge  $n$  als Array

## Vorteile:

- ▶ direkter Zugriff auf Elemente in konstanter Zeit mittels  $A[i]$
- ▶ sequentielles Durchlaufen sehr einfach

## Nachteile:

- ▶ Verlängern des Feldes aufwendig
- ▶ Hinzufügen und Löschen von Elementen aufwendig

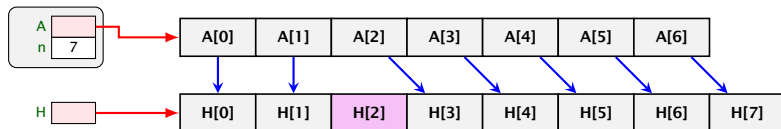
# Hinzufügen eines Elementes

**Gegeben:** Feld  $F$ , Länge  $n$ , via Array implementiert

**Gewünscht:** Feld  $F$ , zusätzliches Element  $d_i$  an Position  $i$ ;  
Elemente an Positionen  $\geq i$  werden auf nächsthöhere Position  
verschoben

- ▶ neuen Speicher der Größe  $n+1$  reservieren
- ▶ altes Array in neuen Speicher kopieren

**Beispiel:**  $F.insert(12, 3)$  (einfügen an Position 3)



## Implementierung: Feld via Array

```
3 class Feld {
4     int n;
5     int* A;
6
7     public:
8     Feld() {
9         n = 0;
10        A = new int[n];
11    }
```

## Implementierung: Feld via Array

```
13 void insert(int d, int i) {
14     int* H = new int[n+1];
15     for (int j=0; j<i-1; j++) {
16         H[j] = A[j];
17     }
18     H[i-1] = d;
19     for (int j=i-1; j<n; j++) {
20         H[j+1] = A[j];
21     }
22     delete[] A;
23     A = H;
24     n++;
25 }
```

## Implementierung: Feld via Array

```
27 void erase(int i) {
28     int* H = new int[n-1];
29     for (int j=0; j<i-1; j++) {
30         H[j] = A[j];
31     }
32     for (int j=i; j<n; j++) {
33         H[j-1] = A[j];
34     }
35     delete[] A;
36     A = H;
37     n--;
38 }
```

## Implementierung: Feld via Array

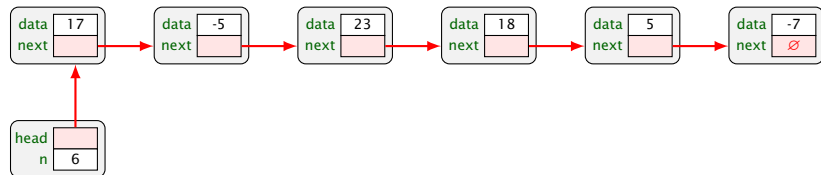
```
40     int elementAt(int i) {
41         return A[i-1];
42     }
43
44     int size() {
45         return n;
46     }
47 };
```



# Feld als einfach verkettete Liste

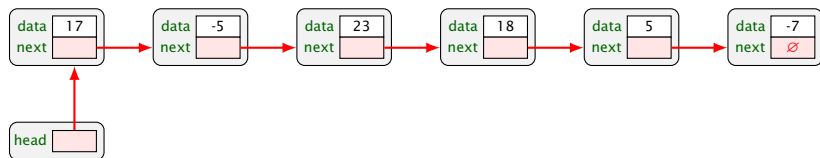
## Repräsentation von Feld als verkettete Liste

- ▶ dynamische Anzahl von Datenelementen
- ▶ in linearer Reihenfolge gespeichert (nicht notwendigerweise zusammenhängend!)
- ▶ mit Referenzen oder Zeigern verkettet



auf Englisch: *linked list*

# Verkettete Liste



Folge von miteinander verbundenen Elementen

jedes Element besteht aus

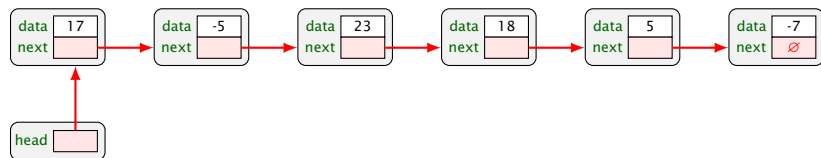
- ▶ **data:** Wert des Feldes an Position  $i$
- ▶ **next:** Referenz auf das nächste Element

head ist Referenz auf erstes Element der Liste

letztes Element hat keinen Nachfolger

- ▶ symbolisiert durch **null**-Referenz

# Verkettete Liste



Folge von miteinander verbundenen Elementen

jedes Element besteht aus

- ▶ **data:** Wert des Feldes an Position  $i$
- ▶ **next:** Referenz auf das nächste Element

**head** ist Referenz auf erstes Element der Liste

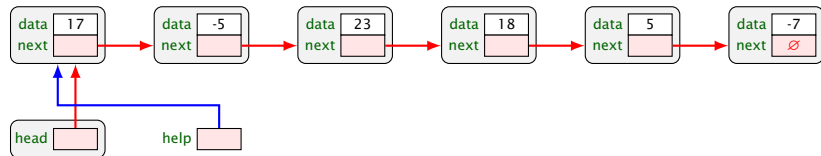
letztes Element hat keinen Nachfolger

- ▶ symbolisiert durch **null**-Referenz

# Verketteter Liste – Zugriff auf Element

## Zugriff auf Element i:

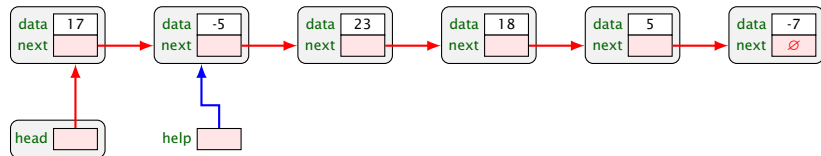
- ▶ beginne bei **head**-Referenz
- ▶ “vorhangeln” entlang **next**-Referenzen bis zum **i**-ten Element



# Verketteter Liste – Zugriff auf Element

## Zugriff auf Element i:

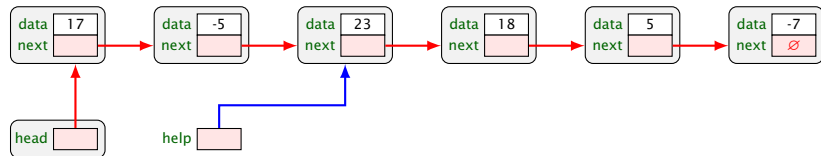
- ▶ beginne bei **head**-Referenz
- ▶ “vorhangeln” entlang **next**-Referenzen bis zum **i**-ten Element



# Verketteter Liste – Zugriff auf Element

## Zugriff auf Element i:

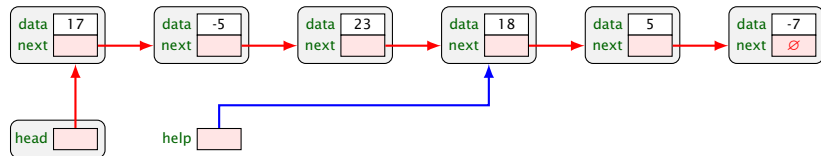
- ▶ beginne bei **head**-Referenz
- ▶ “vorhangeln” entlang **next**-Referenzen bis zum **i**-ten Element



# Verketteter Liste – Zugriff auf Element

## Zugriff auf Element i:

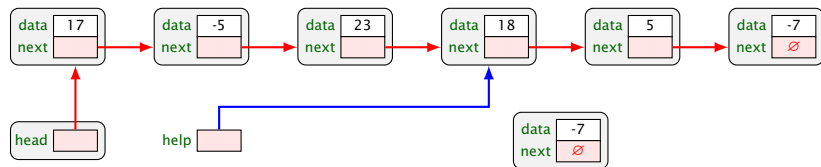
- ▶ beginne bei **head**-Referenz
- ▶ “vorhangeln” entlang **next**-Referenzen bis zum **i**-ten Element



# Verketteter Liste – Einfügen nach Referenz

## Einfügen nach Element i:

- ▶ beginne bei **head**-Referenz
- ▶ “vorhangeln” entlang **next**-Referenzen bis zum **i-1**-ten Element
- ▶ Referenzen umsetzen

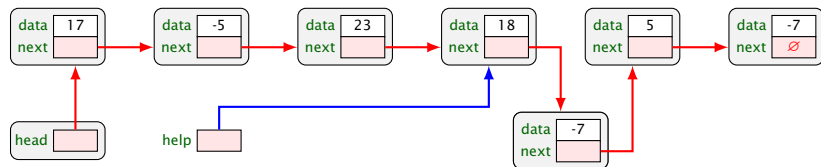




# Verketteter Liste – Einfügen nach Referenz

## Einfügen nach Element $i$ :

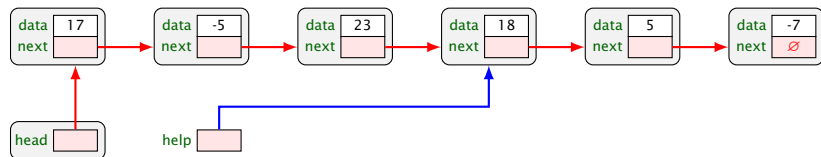
- ▶ beginne bei **head**-Referenz
- ▶ “vorhangeln” entlang **next**-Referenzen bis zum  $i-1$ -ten Element
- ▶ Referenzen umsetzen



# Verketteter Liste – Löschen nach Referenz

## Einfügen nach Element $i$ :

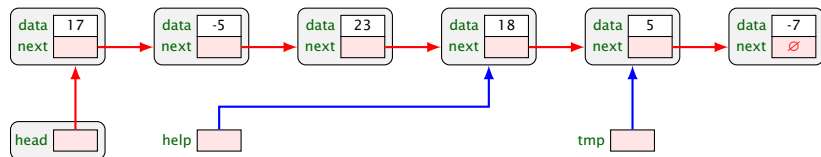
- ▶ beginne bei **head**-Referenz
- ▶ “vorhangeln” entlang **next**-Referenzen bis zum  $i-1$ -ten Element
- ▶ Referenzen umsetzen + Speicher freigeben



# Verketteter Liste – Löschen nach Referenz

## Einfügen nach Element $i$ :

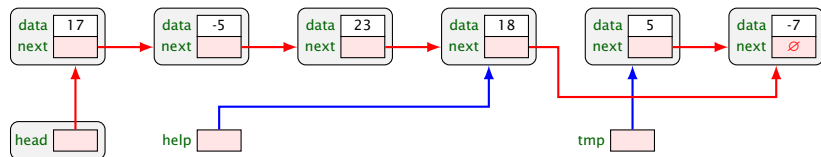
- ▶ beginne bei **head**-Referenz
- ▶ “vorhangeln” entlang **next**-Referenzen bis zum  $i-1$ -ten Element
- ▶ Referenzen umsetzen + Speicher freigeben



# Verketteter Liste – Löschen nach Referenz

## Einfügen nach Element $i$ :

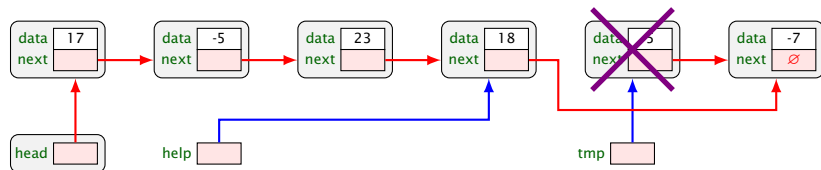
- ▶ beginne bei **head**-Referenz
- ▶ “vorhangeln” entlang **next**-Referenzen bis zum  $i-1$ -ten Element
- ▶ Referenzen umsetzen + Speicher freigeben



# Verketteter Liste – Löschen nach Referenz

## Einfügen nach Element i:

- ▶ beginne bei **head**-Referenz
- ▶ “vorhangeln” entlang **next**-Referenzen bis zum **i-1**-ten Element
- ▶ Referenzen umsetzen + Speicher freigeben



## Implementierung: Feld via Liste

```
4 struct Node {
5     int data;
6     Node *next;
7
8     Node(int d, Node* n) {
9         data = d;
10        next = n;
11    }
12};
```

## Implementierung: Feld via Liste

```
14 class Feld {
15     int n;
16     Node *head;
17 public:
18
19     Feld() { // erzeuge leeres Feld
20         n = 0;
21         head = NULL;
22     }
23
24     int size() { return n; }
25
26     int elementAt(int i) {
27         Node* h = head;
28         while (i-- > 1)
29             h = h->next;
30         return h->data;
31     }
```

## Implementierung: Feld via Liste

```
33     void insert(int d, int i) {
34         Node* tmp = new Node(d, NULL);
35         n++;
36         if (i == 1) {
37             tmp->next = head;
38             head      = tmp;
39             return;
40         }
41         Node* h = head;
42         i--;
43         while (i-- > 1)
44             h = h->next;
45         tmp->next = h->next;
46         h->next  = tmp;
47     }
```



## Implementierung: Feld via Liste

```
49     void erase(int i) {
50         n--;
51         if (i == 1) {
52             Node* tmp = head;
53             head = head->next;
54             delete tmp;
55             return;
56         }
57         Node* h = head;
58         i--;
59         while (i-- > 1)
60             h = h->next;
61         Node* tmp = h->next;
62         h->next = h->next->next;
63         delete tmp;
64     }
65 }; // end class
```

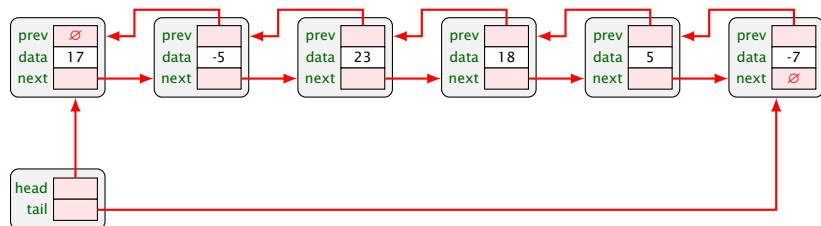
# Gegenüberstellung Array und verkettete Liste

Array	Verkettete Liste
⊕ Direkter Zugriff auf i-tes Element	⊖ Zugriff auf i-tes Element erfordert i Iterationen
⊕ sequentielles Durchlaufen sehr einfach	⊕ sequentielles Durchlaufen sehr einfach
⊖ statische Länge, kann Speicher verschwenden	⊕ dynamische Länge
	⊖ zusätzlicher Speicher für Zeiger benötigt
⊖ Einfügen/Löschen erfordert erheblich Kopieraufwand	⊕ Einfügen/Löschen einfach

# Feld als doppelt verkettete Liste

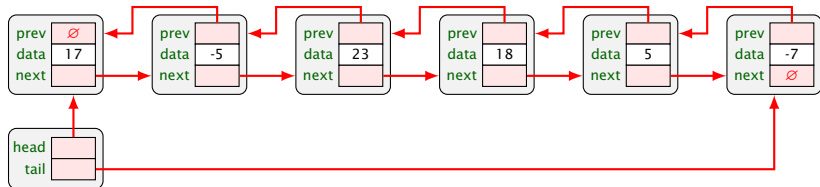
## Repräsentation von Feld A als doppelt verkettete Liste

- ▶ verkettete Liste
- ▶ jedes Element mit Referenzen **doppelt** verkettet



auf Englisch: *doubly linked list*

## Doppelt verkettete Liste



Folge von miteinander verbundenen Elementen

Jedes Element besteht aus

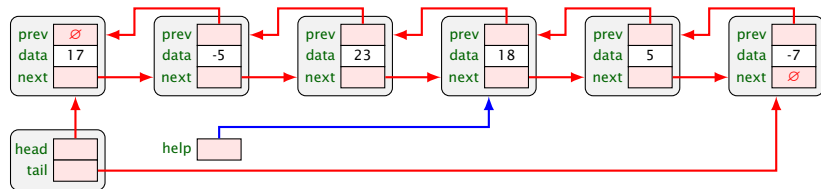
- ▶ **data**: Wert des Feldes
- ▶ **next**: Referenz auf das nächste Element
- ▶ **prev**: Referenz auf das vorherige Element

**head/tail** sind Referenzen auf erstes/letztes Element; diese haben keinen Nachfolger bzw. keinen Vorgänger.

# Operationen auf doppelt verketteter Liste

## Löschen von Element $i$ :

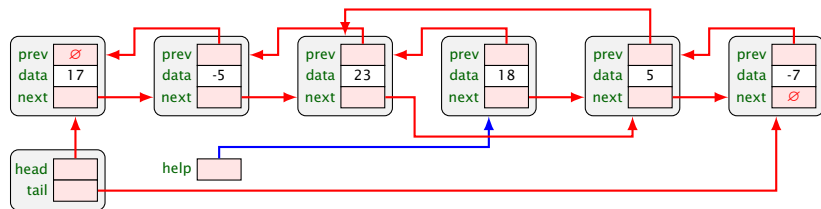
- ▶ Zugriff auf Element  $i$
- ▶ umhängen von Referenzen
- ▶ Speicherplatz freigeben



# Operationen auf doppelt verketteter Liste

## Löschen von Element $i$ :

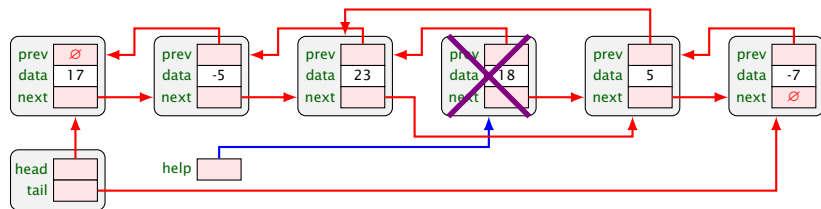
- ▶ Zugriff auf Element  $i$
- ▶ umhängen von Referenzen
- ▶ Speicherplatz freigeben



# Operationen auf doppelt verketteter Liste

## Löschen von Element $i$ :

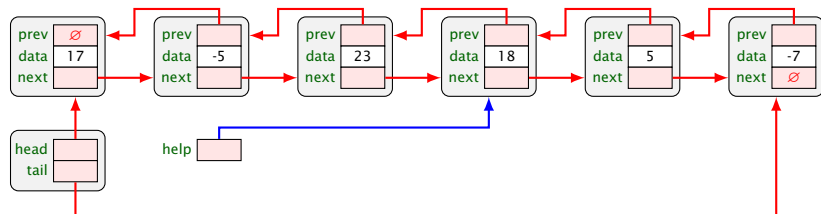
- ▶ Zugriff auf Element  $i$
- ▶ umhängen von Referenzen
- ▶ Speicherplatz freigeben



# Operationen auf doppelt verketteter Liste

## Einfügen von Element an Stelle $i$ :

- ▶ Zugriff auf Element  $i-1$
- ▶ umhängen von Referenzen

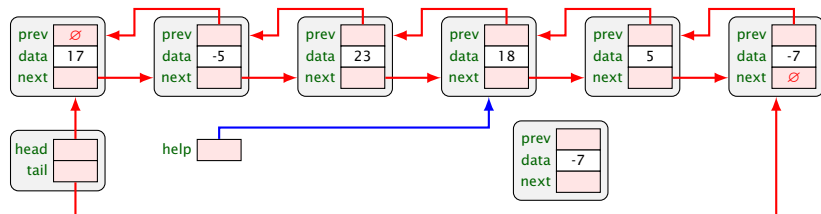




# Operationen auf doppelt verketteter Liste

## Einfügen von Element an Stelle $i$ :

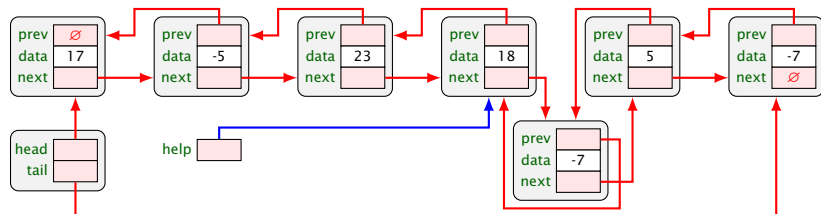
- ▶ Zugriff auf Element  $i-1$
- ▶ umhängen von Referenzen



# Operationen auf doppelt verketteter Liste

## Einfügen von Element an Stelle $i$ :

- ▶ Zugriff auf Element  $i-1$
- ▶ umhängen von Referenzen



# Eigenschaften doppelt verkettete Liste

## Doppelt verkettete Liste

### Vorteile:

- ▶ Durchlauf in beiden Richtungen möglich
- ▶ Einfügen/Löschen potentiell einfacher, da man sich Vorgänger nicht extra merken muss

### Nachteile:

- ▶ zusätzlicher Speicher erforderlich für zwei Referenzen
- ▶ Referenzverwaltung komplizierter und fehleranfällig

# Zusammenfassung Felder

Ein **Feld A** kann repräsentiert werden als:

- ▶ **Array**
- ▶ **verkettete Liste** (linked list)
- ▶ **doppelt verkettete Liste** (doubly linked list)

Eigenschaften:

- ▶ einfach und flexibel
- ▶ aber manche Operationen aufwendig

# Zusammenfassung Felder

Ein **Feld** A kann repräsentiert werden als:

- ▶ **Array**
- ▶ **verkettete Liste** (linked list)
- ▶ **doppelt verkettete Liste** (doubly linked list)

**Eigenschaften:**

- ▶ einfach und flexibel
- ▶ aber manche Operationen aufwendig

# Definition Abstrakter Datentyp

**Abstrakter Datentyp (englisch: abstract data type, ADT)** Ein **abstrakter Datentyp** ist ein mathematisches **Modell** für bestimmte Datenstrukturen mit vergleichbarem Verhalten.

Ein abstrakter Datentyp wird **indirekt** definiert über

- ▶ mögliche **Operationen** auf ihm sowie
- ▶ mathematische Bedingungen (oder: constraints) über die **Auswirkungen der Operationen** (u.U. auch die Kosten der Operationen).

# Beispiel abstrakter Datentyp: abstrakte Variable

Abstrakte Variable  $V$  ist eine veränderliche Dateneinheit mit zwei Operationen

- ▶  $\text{load}(V)$  liefert einen Wert
- ▶  $\text{store}(V, x)$  wobei  $x$  ein Wert ist

und der Bedingung

- ▶  $\text{load}(V)$  liefert immer den Wert  $x$  der letzten Operation  $\text{store}(V, x)$

# Beispiel abstrakter Datentyp: abstrakte Liste (Teil 1)

Abstrakte Liste  $L$  ist ein Datentyp

mit Operationen

- ▶  $\text{pushFront}(L, x)$  liefert eine Liste
- ▶  $\text{front}(L)$  liefert ein Element
- ▶  $\text{rest}(L)$  liefert eine Liste

und den Bedingungen

- ▶ ist  $x$  Element,  $L$  Liste, dann liefert  $\text{front}(\text{pushFront}(L, x))$  das Element  $x$ .
- ▶ ist  $x$  Element,  $L$  Liste, dann liefert  $\text{rest}(\text{pushFront}(L, x))$  die Liste  $L$ .



## Beispiel abstrakter Datentyp: abstrakte Liste (Teil 2)

Abstrakte Liste  $L$ . Weitere Operationen sind

- ▶ `isEmpty(L)` liefert `true` oder `false`
- ▶ `initialize()` liefert eine Listeninstanz

mit den Bedingungen

- ▶ `initialize() ≠ L` für jede Liste  $L$  (d.h. jede neue Liste ist separat von alten Listen)
- ▶ `isEmpty(initialize()) == true` (d.h. eine neue Liste ist leer)
- ▶ `isEmpty(pushFront(L, x)) == false` (d.h. eine Liste ist nach einem `pushFront` nicht leer)

# Definition Stack

**Stack (oder deutsch: Stapel, Keller)** Ein **Stack** ist ein abstrakter Datentyp. Er beschreibt eine spezielle Listenstruktur nach dem **Last In – First Out (LIFO)** Prinzip mit den Eigenschaften

- ▶ löschen, einfügen ist nur **am Ende** der Liste erlaubt,
- ▶ nur das **letzte Element** darf manipuliert werden.

# Definition Stack

**Stack (oder deutsch: Stapel, Keller)** Ein **Stack** ist ein abstrakter Datentyp. Er beschreibt eine spezielle Listenstruktur nach dem **Last In – First Out (LIFO)** Prinzip mit den Eigenschaften

- ▶ löschen, einfügen ist nur **am Ende** der Liste erlaubt,
- ▶ nur das **letzte Element** darf manipuliert werden.

## Operationen auf Stacks:

- ▶ **push**: legt ein Element auf den Stack (einfügen)
- ▶ **pop**: entfernt das letzte Element vom Stack (löschen)
- ▶ **top**: liefert das letzte Stack-Element
- ▶ **isEmpty**: liefert **true** falls Stack leer
- ▶ **initialize**: Stack erzeugen und in Anfangszustand (leer) setzen

# Definition Stack

**Stack (oder deutsch: Stapel, Keller)** Ein **Stack** ist ein abstrakter Datentyp. Er beschreibt eine spezielle Listenstruktur nach dem **Last In – First Out (LIFO)** Prinzip mit den Eigenschaften

- ▶ löschen, einfügen ist nur **am Ende** der Liste erlaubt,
- ▶ nur das **letzte Element** darf manipuliert werden.

## Definition Stack (exakter)

Stack  $S$  ist ein abstrakter Datentyp mit Operationen

- ▶  $\text{pop}(S)$  liefert einen Wert
- ▶  $\text{push}(S, x)$  wobei  $x$  ein Wert

mit der Bedingung

- ▶ ist  $x$  Wert und  $V$  Variable, dann ist die Sequenz  $\text{push}(S, x); V = \text{pop}(S);$  äquivalent zu  $V = x;$

## Definition Stack (exakter)

Stack  $S$  ist ein abstrakter Datentyp mit Operationen

- ▶  $\text{pop}(S)$  liefert einen Wert
- ▶  $\text{push}(S, x)$  wobei  $x$  ein Wert

mit der Bedingung

- ▶ ist  $x$  Wert und  $V$  Variable, dann ist die Sequenz  $\text{push}(S, x); V = \text{pop}(S);$  äquivalent zu  $V = x;$

sowie der Operation

- ▶  $\text{top}(S)$  liefert einen Wert

mit der Bedingung

- ▶ ist  $x$  Wert und  $V$  Variable, dann ist die Sequenz  $\text{push}(S, x); V = \text{top}(S);$  äquivalent zu  $\text{push}(S, x); V = x;$

## Definition Stack (exakter, Teil 2)

Stack  $S$ . Weitere Operationen sind

- ▶ `isEmpty(S)` liefert `true` oder `false`
- ▶ `initialize()` liefert eine Stackinstanz

mit den Bedingungen

- ▶ `initialize() ≠ S` für jeden Stack  $S$  (d.h. jeder neue Stack ist separat von alten Stacks)
- ▶ `isEmpty(initialize()) == true` (d.h. ein neuer Stack ist leer)
- ▶ `isEmpty(push(S, x)) == false` (d.h. ein Stack nach push ist nicht leer)

# Anwendungsbeispiele Stack

Call-Stack bei Funktionsaufrufen

Einfache Vorwärts- / Rückwärts Funktion in Software

- ▶ z.B. im Internet-Browser

Syntaxanalyse eines Programms

- ▶ z.B. zur Erkennung von Syntax-Fehlern durch Compiler

Auswertung arithmetischer Ausdrücke



# Auswertung arithmetischer Ausdrücke

Gegeben sei ein vollständig geklammerter, einfacher arithmetischer Ausdruck mit Bestandteilen Zahl, +, \*, =

**Beispiel:**  $(3 * (4 + 5)) =$

# Auswertung arithmetischer Ausdrücke

Gegeben sei ein vollständig geklammerter, einfacher arithmetischer Ausdruck mit Bestandteilen Zahl, +, \*, =

**Beispiel:**  $(3 * (4 + 5)) =$

## Schema:

- ▶ arbeite Ausdruck von links nach rechts ab, speichere jedes Zeichen ausser ) und = in Stack S
- ▶ bei ) werte die 3 obersten Elemente von S aus, dann entferne die passende Klammer ( vom Stack S und speichere Ergebnis in Stack S
- ▶ bei = steht das Ergebnis im obersten Stack-Element von S

# Beispiel: Auswertung arithmetischer Ausdrücke

$$(((3 + 1) \cdot 7) + (14 - 3))$$

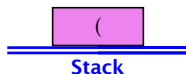
# Beispiel: Auswertung arithmetischer Ausdrücke

Stack



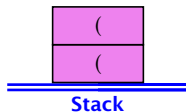
$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



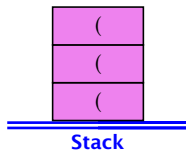
$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



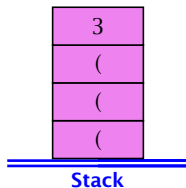
$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



$$(((3 + 1) \cdot 7) + (14 - 3))$$

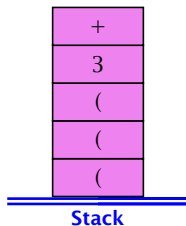
# Beispiel: Auswertung arithmetischer Ausdrücke



$$(((3 + 1) \cdot 7) + (14 - 3))$$

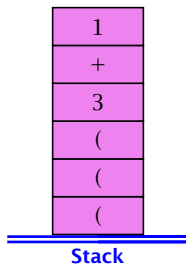


# Beispiel: Auswertung arithmetischer Ausdrücke



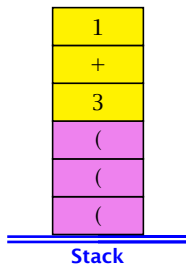
$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



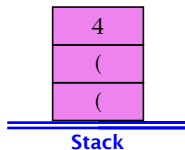
$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



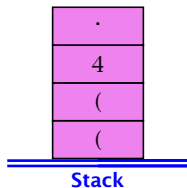
$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



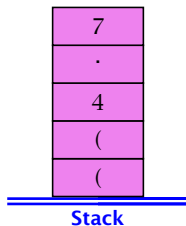
$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



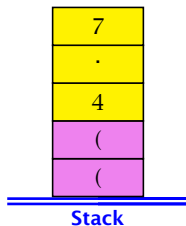
$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



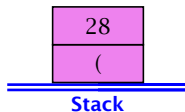
$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



$$(((3 + 1) \cdot 7) + (14 - 3))$$

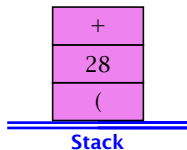
# Beispiel: Auswertung arithmetischer Ausdrücke



$$(((3 + 1) \cdot 7) + (14 - 3))$$

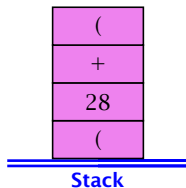


# Beispiel: Auswertung arithmetischer Ausdrücke



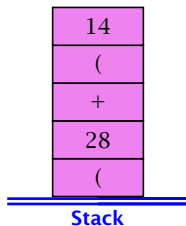
$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



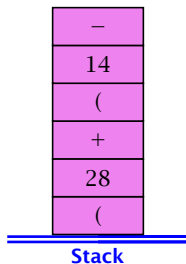
$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



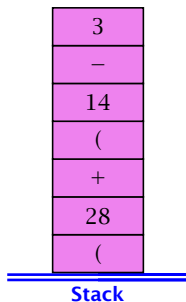
$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



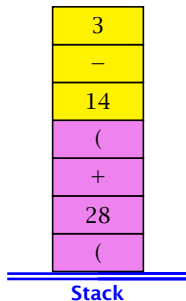
$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



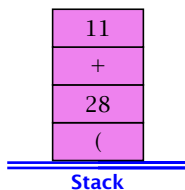
$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



$$(((3 + 1) \cdot 7) + (14 - 3))$$

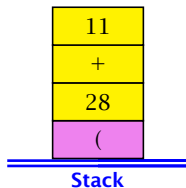
# Beispiel: Auswertung arithmetischer Ausdrücke



( ( ( 3 + 1 ) · 7 ) + ( 14 - 3 ) )

$$(((3 + 1) \cdot 7) + (14 - 3))$$

# Beispiel: Auswertung arithmetischer Ausdrücke



$$(((3 + 1) \cdot 7) + (14 - 3))$$



# Beispiel: Auswertung arithmetischer Ausdrücke

39

Stack

( ( ( 3 + 1 ) · 7 ) + ( 14 - 3 ) )

$((3 + 1) \cdot 7) + (14 - 3)$

# Implementation Stack

## Stack ist abstrakter Datentyp.

- ▶ Implementation ist nicht festgelegt
- ▶ nur Operationen und Bedingungen sind festgelegt

# Implementation Stack

## Stack ist abstrakter Datentyp.

- ▶ Implementation ist nicht festgelegt
- ▶ nur Operationen und Bedingungen sind festgelegt

Stack kann auf viele Arten implementiert werden, zum Beispiel als:

- ▶ Array
- ▶ verkettete Liste

# Implementation Stack via Array

Stack-Elemente im Array (Länge  $n$ ) speichern

oberstes Stack-Element merken mittels Variable  $top$

falls Stack **leer** ist  $top == -1$

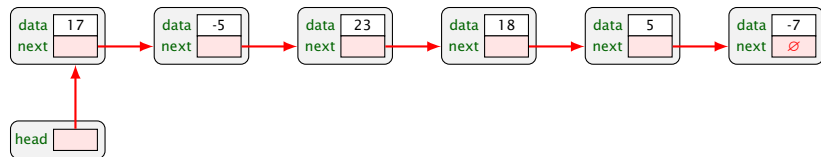


- ▶  $push(x)$  inkrementiert  $top$  und speichert  $x$  in  $A[top]$
- ▶  $pop()$  liefert  $A[top]$  zurück und dekrementiert  $top$
- ▶  $top()$  liefert  $A[top]$  zurück

# Implementation Stack als verkettete Liste

Stack-Elemente speichern in verketteter Liste

oberstes Stack-Element wird durch **head**-Referenz markiert



- ▶ **push(x)** fügt Element an erster Position ein
- ▶ **pop()** liefert Element an erster Position zurück und entfernt es
- ▶ **top()** liefert Element an erster Position zurück

# Zusammenfassung Stack

Stack ist **abstrakter Datentyp** als Metapher für einen Stapel

- ▶ wesentliche Operationen: **push, pop**

Implementation als **Array**

- ▶ fixe Größe (entweder Speicher verschwendet oder zu klein)
- ▶ push, pop sehr effizient

Implementation als **verkettete Liste**

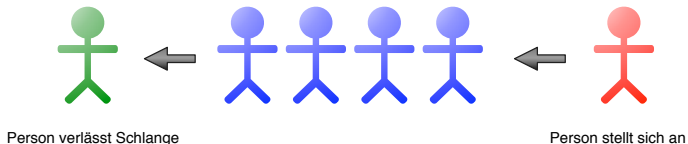
- ▶ dynamische Größe, aber Platz für Zeiger “verschwendet”
- ▶ push, pop effizient
- ▶ eventuell nicht cache-effizient

# Definition Queue

## Queue (oder deutsch: Warteschlange)

Eine **Queue** ist ein abstrakter Datentyp. Sie beschreibt eine spezielle Listenstruktur nach dem **First In – First Out (FIFO)** Prinzip mit den Eigenschaften

- ▶ einfügen ist nur **am Ende** der Liste erlaubt,
- ▶ entfernen ist nur **am Anfang** der Liste erlaubt.



# Definition Queue

## Queue (oder deutsch: Warteschlange)

Eine **Queue** ist ein abstrakter Datentyp. Sie beschreibt eine spezielle Listenstruktur nach dem **First In – First Out (FIFO)** Prinzip mit den Eigenschaften

- ▶ einfügen ist nur **am Ende** der Liste erlaubt,
- ▶ entfernen ist nur **am Anfang** der Liste erlaubt.

## Operationen auf Queues:

- ▶ **enqueue**: fügt ein Element am Ende der Schlange hinzu
- ▶ **dequeue**: entfernt das erste Element der Schlange
- ▶ **isEmpty**: liefert **true** falls Queue leer
- ▶ **initialize**: Queue erzeugen und in Anfangszustand (leer) setzen



## Definition Queue (exakter)

Queue  $Q$  ist ein abstrakter Datentyp mit Operationen

- ▶ `dequeue(Q)` liefert einen Wert
- ▶ `enqueue(Q, x)` wobei  $x$  ein Wert
- ▶ `isEmpty(Q)` liefert `true` oder `false`
- ▶ `initialize` liefert eine Queue Instanz

und mit Bedingungen

- ▶ ist  $x$  Wert,  $V$  Variable,  $Q$  leere Queue, dann ist Sequenz `enqueue(Q, x); V=dequeue(Q);` äquivalent zu `V=x.`
- ▶ sind  $x, y$  Werte,  $V$  Variable und  $Q$  Queue, dann ist Sequenz `enqueue(Q, x); enqueue(Q, y); V=dequeue(Q)` äquivalent zu `enqueue(Q, x); V=dequeue(Q); enqueue(Q, y)`
- ▶ `initialize() ≠ Q` für jede Queue  $Q$
- ▶ `isEmpty(initialize()) == true`
- ▶ `isEmpty(enqueue(Q, x)) == false`

## Definition Queue (exakter)

Queue  $Q$  ist ein abstrakter Datentyp mit Operationen

- ▶ `dequeue(Q)` liefert einen Wert
- ▶ `enqueue(Q, x)` wobei  $x$  ein Wert
- ▶ `isEmpty(Q)` liefert `true` oder `false`
- ▶ `initialize` liefert eine Queue Instanz

und mit Bedingungen

- ▶ ist  $x$  Wert,  $V$  Variable,  $Q$  leere Queue, dann ist Sequenz `enqueue(Q, x); V=dequeue(Q);` äquivalent zu `V=x.`
- ▶ sind  $x, y$  Werte,  $V$  Variable und  $Q$  Queue, dann ist Sequenz `enqueue(Q, x); enqueue(Q, y); V=dequeue(Q)` äquivalent zu `enqueue(Q, x); V=dequeue(Q); enqueue(Q, y)`
- ▶ `initialize() ≠ Q` für jede Queue  $Q$
- ▶ `isEmpty(initialize()) == true`
- ▶ `isEmpty(enqueue(Q, x)) == false`

## Definition Queue (exakter)

Queue  $Q$  ist ein abstrakter Datentyp mit Operationen

- ▶ `dequeue(Q)` liefert einen Wert
- ▶ `enqueue(Q, x)` wobei  $x$  ein Wert
- ▶ `isEmpty(Q)` liefert `true` oder `false`
- ▶ `initialize` liefert eine Queue Instanz

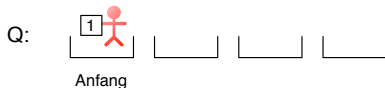
und mit Bedingungen

- ▶ ist  $x$  Wert,  $V$  Variable,  $Q$  leere Queue, dann ist Sequenz `enqueue(Q, x); V=dequeue(Q);` äquivalent zu `V=x.`
- ▶ sind  $x, y$  Werte,  $V$  Variable und  $Q$  Queue, dann ist Sequenz `enqueue(Q, x); enqueue(Q, y); V=dequeue(Q)` äquivalent zu `enqueue(Q, x); V=dequeue(Q); enqueue(Q, y)`
- ▶ `initialize() ≠ Q` für jede Queue  $Q$
- ▶ `isEmpty(initialize()) == true`
- ▶ `isEmpty(enqueue(Q, x)) == false`

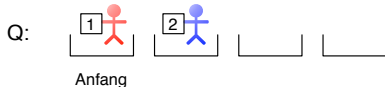
# Beispiel: Queue



`Q = initialize();`



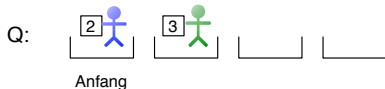
`enqueue(1);`



`enqueue(2);`



`enqueue(3);`



`dequeue();`



`dequeue();`

# Anwendungsbeispiele Queue

- ▶ Druckerwarteschlange
- ▶ Playlist von iTunes (oder ähnlichem Musikprogramm)
- ▶ Kundenaufträge bei Webshops
- ▶ Warteschlange für Prozesse im Betriebssystem (Multitasking)

# Anwendungsbeispiel Stack und Queue

## Palindrom

Ein Palindrom ist eine Zeichenkette, die von vorn und von hinten gelesen gleich bleibt.

**Beispiel:** Reittier

Erkennung ob Zeichenkette ein Palindrom ist

- ▶ ein **Stack** kann die Reihenfolge der Zeichen umkehren
- ▶ eine **Queue** behält die Reihenfolge der Zeichen

# Palindromerkennung

```
1 Input: Zeichenkette str mit Laenge n
2 Output: true falls str Palindrom; sonst false
3
4 Stack S;
5 Queue Q;
6
7 i = 0;
8 while (i<n)
9     S.push(str[i]);
10    Q.enqueue(str[i]);
11    i++;
12 i = 0;
13 while (i<n)
14     s = S.pop();
15     q = Q.dequeue();
16     if (s != q) return false;
17     i++;
18 return true;
```

# Implementation Queue

Auch Queue ist abstrakter Datentyp.

- ▶ Implementation ist nicht festgelegt
- ▶ nur Operationen und Bedingungen sind festgelegt

Queue kann auf viele Arten implementiert werden, zum Beispiel als:

- ▶ verkettete Liste
- ▶ Array

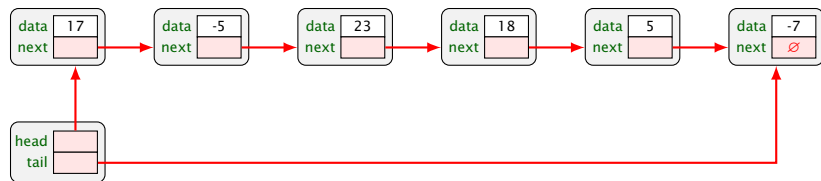


# Implementation Queue als verkettete Liste

Queue-Elemente speichern in verketteter Liste

Anfang der Queue wird durch **head**-Referenz markiert

Ende der Queue wird durch extra **tail**-Referenz markiert



- ▶ **enqueue(x)** fügt Element bei **head**-Referenz ein
- ▶ **dequeue()** liefert Element bei **tail**-Referenz zurück und entfernt es



## Implementation Queue als zwei Stacks

Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- ▶ `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- ▶ falls `outbox` leer, kopiere alle Elemente von `inbox` zu `outbox`: `outbox.push( inbox.pop() )`
- ▶ `Q.dequeue()` liefert `outbox.pop()` zurück

inbox

`push(1)`

outbox

## Implementation Queue als zwei Stacks

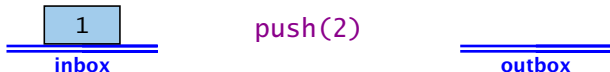
Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- ▶ `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- ▶ falls `outbox` leer, kopiere alle Elemente von `inbox` zu `outbox`: `outbox.push( inbox.pop() )`
- ▶ `Q.dequeue()` liefert `outbox.pop()` zurück



## Implementation Queue als zwei Stacks

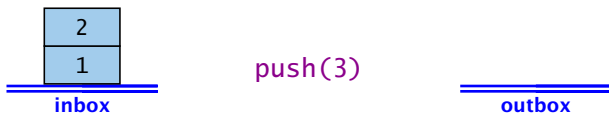
Queue  $Q$  kann mittels zwei Stacks implementiert werden

erster Stack  $inbox$  wird für  $enqueue$  benutzt:

- ▶  $Q.enqueue(x)$  resultiert in  $inbox.push(x)$

zweiter Stack  $outbox$  wird für  $dequeue$  benutzt:

- ▶ falls  $outbox$  leer, kopiere alle Elemente von  $inbox$  zu  $outbox$ :  $outbox.push( inbox.pop() )$
- ▶  $Q.dequeue()$  liefert  $outbox.pop()$  zurück



# Implementation Queue als zwei Stacks

Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- ▶ `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- ▶ falls `outbox` leer, kopiere alle Elemente von `inbox` zu `outbox`: `outbox.push( inbox.pop() )`
- ▶ `Q.dequeue()` liefert `outbox.pop()` zurück



## Implementation Queue als zwei Stacks

Queue  $Q$  kann mittels zwei Stacks implementiert werden

erster Stack  $inbox$  wird für  $enqueue$  benutzt:

- ▶  $Q.enqueue(x)$  resultiert in  $inbox.push(x)$

zweiter Stack  $outbox$  wird für  $dequeue$  benutzt:

- ▶ falls  $outbox$  leer, kopiere alle Elemente von  $inbox$  zu  $outbox$ :  $outbox.push( inbox.pop() )$
- ▶  $Q.dequeue()$  liefert  $outbox.pop()$  zurück



## Implementation Queue als zwei Stacks

Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- ▶ `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- ▶ falls `outbox` leer, kopiere alle Elemente von `inbox` zu `outbox`: `outbox.push( inbox.pop() )`
- ▶ `Q.dequeue()` liefert `outbox.pop()` zurück





# Implementation Queue als zwei Stacks

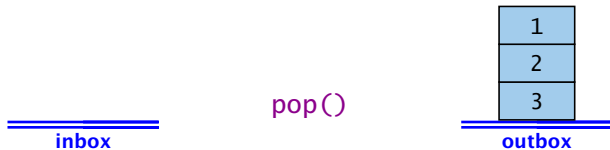
Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- ▶ `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- ▶ falls `outbox` leer, kopiere alle Elemente von `inbox` zu `outbox`: `outbox.push( inbox.pop() )`
- ▶ `Q.dequeue()` liefert `outbox.pop()` zurück



# Implementation Queue als zwei Stacks

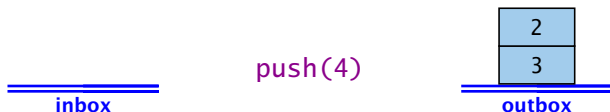
Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- ▶ `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- ▶ falls `outbox` leer, kopiere alle Elemente von `inbox` zu `outbox`: `outbox.push( inbox.pop() )`
- ▶ `Q.dequeue()` liefert `outbox.pop()` zurück



# Implementation Queue als zwei Stacks

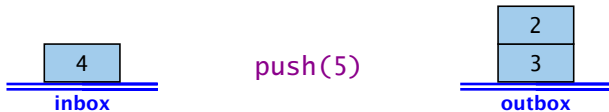
Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- ▶ `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- ▶ falls `outbox` leer, kopiere alle Elemente von `inbox` zu `outbox`: `outbox.push( inbox.pop() )`
- ▶ `Q.dequeue()` liefert `outbox.pop()` zurück



# Implementation Queue als zwei Stacks

Queue  $Q$  kann mittels zwei Stacks implementiert werden

erster Stack  $inbox$  wird für  $enqueue$  benutzt:

- ▶  $Q.enqueue(x)$  resultiert in  $inbox.push(x)$

zweiter Stack  $outbox$  wird für  $dequeue$  benutzt:

- ▶ falls  $outbox$  leer, kopiere alle Elemente von  $inbox$  zu  $outbox$ :  $outbox.push( inbox.pop() )$
- ▶  $Q.dequeue()$  liefert  $outbox.pop()$  zurück



## Implementation Queue als zwei Stacks

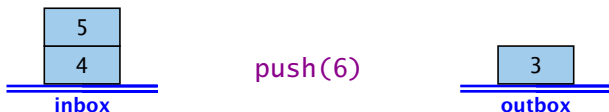
Queue  $Q$  kann mittels zwei Stacks implementiert werden

erster Stack  $inbox$  wird für  $enqueue$  benutzt:

- ▶  $Q.enqueue(x)$  resultiert in  $inbox.push(x)$

zweiter Stack  $outbox$  wird für  $dequeue$  benutzt:

- ▶ falls  $outbox$  leer, kopiere alle Elemente von  $inbox$  zu  $outbox$ :  $outbox.push( inbox.pop() )$
- ▶  $Q.dequeue()$  liefert  $outbox.pop()$  zurück



## Implementation Queue als zwei Stacks

Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- ▶ `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- ▶ falls `outbox` leer, kopiere alle Elemente von `inbox` zu `outbox`: `outbox.push( inbox.pop() )`
- ▶ `Q.dequeue()` liefert `outbox.pop()` zurück



## Implementation Queue als zwei Stacks

Queue `Q` kann mittels zwei Stacks implementiert werden

erster Stack `inbox` wird für `enqueue` benutzt:

- ▶ `Q.enqueue(x)` resultiert in `inbox.push(x)`

zweiter Stack `outbox` wird für `dequeue` benutzt:

- ▶ falls `outbox` leer, kopiere alle Elemente von `inbox` zu `outbox`: `outbox.push( inbox.pop() )`
- ▶ `Q.dequeue()` liefert `outbox.pop()` zurück



# Zusammenfassung Queue

Queue ist abstrakter Datentyp als Metapher für eine Warteschlange

- ▶ wesentliche Operationen: **enqueue**, **dequeue**

Implementation als verkettete Liste

- ▶ dynamische Größe, aber Platz für Referenzen “verschwendet”
- ▶ enqueue, dequeue effizient
- ▶ nicht cache-effizient

Implementation als Array

- ▶ fixe Größe (entweder Speicher verschwendet oder zu klein)
- ▶ enqueue, dequeue sehr effizient



# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- ▶ Speicherverbrauch
- ▶ Laufzeit
- ▶ Anzahl Vergleiche (z.B. Sortieralgorithmen)
- ▶ Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- ▶ Festplattenzugriffe
- ▶ Programmgröße
- ▶ Energieverbrauch
- ▶ ...

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- ▶ Speicherverbrauch
- ▶ Laufzeit
- ▶ Anzahl Vergleiche (z.B. Sortieralgorithmen)
- ▶ Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- ▶ Festplattenzugriffe
- ▶ Programmgröße
- ▶ Energieverbrauch
- ▶ ...

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- ▶ Speicherverbrauch
- ▶ Laufzeit
- ▶ Anzahl Vergleiche (z.B. Sortieralgorithmen)
- ▶ Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- ▶ Festplattenzugriffe
- ▶ Programmgröße
- ▶ Energieverbrauch
- ▶ ...

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- ▶ Speicherverbrauch
- ▶ Laufzeit
- ▶ Anzahl Vergleiche (z.B. Sortieralgorithmen)
- ▶ Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- ▶ Festplattenzugriffe
- ▶ Programmgröße
- ▶ Energieverbrauch
- ▶ ...

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- ▶ Speicherverbrauch
- ▶ Laufzeit
- ▶ Anzahl Vergleiche (z.B. Sortieralgorithmen)
- ▶ Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- ▶ Festplattenzugriffe
- ▶ Programmgröße
- ▶ Energieverbrauch
- ▶ ...

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- ▶ Speicherverbrauch
- ▶ Laufzeit
- ▶ Anzahl Vergleiche (z.B. Sortieralgorithmen)
- ▶ Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- ▶ Festplattenzugriffe
- ▶ Programmgröße
- ▶ Energieverbrauch
- ▶ ...

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- ▶ Speicherverbrauch
- ▶ Laufzeit
- ▶ Anzahl Vergleiche (z.B. Sortieralgorithmen)
- ▶ Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- ▶ Festplattenzugriffe
- ▶ Programmgröße
- ▶ Energieverbrauch
- ▶ ...

# 5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

**Was messen wir?**

- ▶ Speicherverbrauch
- ▶ Laufzeit
- ▶ Anzahl Vergleiche (z.B. Sortieralgorithmen)
- ▶ Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- ▶ Festplattenzugriffe
- ▶ Programmgröße
- ▶ Energieverbrauch
- ▶ ...



# 5 Effizienz von Algorithmen

## Wie messen wir?

- ▶ Implementieren und Testen auf repräsentativen Eingaben.
  - ▶ Welche Eingaben?
  - ▶ Kann sehr aufwendig sein.
  - ▶ Präzise Resultate wenn sorgfältig durchgeführt.
  - ▶ Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- ▶ Theoretische Analyse in einem Rechenmodell.

Die Analyse eines Algorithmus z.B. eines Algorithmus zur Suche in einem Graphen führt immer zu Zeit- und Speicherbedarf. Die genaue Analyse wird durch das Rechenmodell bestimmt. Man kann auch andere Schranken ermitteln, jedoch weniger präzise. Kommentare hierzu sind im nächsten Kapitel zu finden.



# 5 Effizienz von Algorithmen

## Wie messen wir?

- ▶ Implementieren und Testen auf repräsentativen Eingaben.
  - ▶ Welche Eingaben?
  - ▶ Kann sehr aufwendig sein.
  - ▶ Präzise Resultate wenn sorgfältig durchgeführt.
  - ▶ Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- ▶ Theoretische Analyse in einem Rechenmodell.

Wie kann man die Effizienz eines Algorithmus messen?  
• Zeit  
• Speicher  
• Energieverbrauch  
• ...  
• Die Effizienz eines Algorithmus wird durch die Komplexität des Algorithmus bestimmt.  
• Die Komplexität eines Algorithmus ist die Anzahl der Operationen, die er ausführt.  
• Die Komplexität eines Algorithmus ist eine Funktion der Größe der Eingabe.  
• Die Komplexität eines Algorithmus ist eine Funktion der Größe der Eingabe und der Größe der Ausgabe.

# 5 Effizienz von Algorithmen

## Wie messen wir?

- ▶ Implementieren und Testen auf repräsentativen Eingaben.
  - ▶ Welche Eingaben?
  - ▶ Kann sehr aufwendig sein.
  - ▶ Präzise Resultate wenn sorgfältig durchgeführt.
    - ▶ Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- ▶ Theoretische Analyse in einem Rechenmodell.

# 5 Effizienz von Algorithmen

## Wie messen wir?

- ▶ Implementieren und Testen auf repräsentativen Eingaben.
  - ▶ Welche Eingaben?
  - ▶ Kann sehr aufwendig sein.
  - ▶ Präzise Resultate wenn sorgfältig durchgeführt.
  - ▶ Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- ▶ Theoretische Analyse in einem Rechenmodell.

# 5 Effizienz von Algorithmen

## Wie messen wir?

- ▶ Implementieren und Testen auf repräsentativen Eingaben.
  - ▶ Welche Eingaben?
  - ▶ Kann sehr aufwendig sein.
  - ▶ Präzise Resultate wenn sorgfältig durchgeführt.
  - ▶ Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- ▶ Theoretische Analyse in einem **Rechenmodell**.
  - ▶ Gibt **asymptotische Garantien** wie z.B. „dieser Algorithmus läuft immer in Zeit  $\mathcal{O}(n^2)$ “.
  - ▶ Üblicherweise wird der **worst case** betrachtet.
  - ▶ Man kann auch untere Schranken erhalten: „jedes vergleichsbasierte Sortierverfahren benötigt im worst case mindestens  $\Omega(n \log n)$  Vergleiche“.

# 5 Effizienz von Algorithmen

## Wie messen wir?

- ▶ Implementieren und Testen auf repräsentativen Eingaben.
  - ▶ Welche Eingaben?
  - ▶ Kann sehr aufwendig sein.
  - ▶ Präzise Resultate wenn sorgfältig durchgeführt.
  - ▶ Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- ▶ Theoretische Analyse in einem **Rechenmodell**.
  - ▶ Gibt **asymptotische Garantien** wie z.B. „dieser Algorithmus läuft immer in Zeit  $\mathcal{O}(n^2)$ “.
  - ▶ Üblicherweise wird der **worst case** betrachtet.
  - ▶ Man kann auch untere Schranken erhalten: „jedes vergleichsbasierte Sortierverfahren benötigt im worst case mindestens  $\Omega(n \log n)$  Vergleiche“.

# 5 Effizienz von Algorithmen

## Wie messen wir?

- ▶ Implementieren und Testen auf repräsentativen Eingaben.
  - ▶ Welche Eingaben?
  - ▶ Kann sehr aufwendig sein.
  - ▶ Präzise Resultate wenn sorgfältig durchgeführt.
  - ▶ Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- ▶ Theoretische Analyse in einem **Rechenmodell**.
  - ▶ Gibt **asymptotische Garantien** wie z.B. „dieser Algorithmus läuft immer in Zeit  $\mathcal{O}(n^2)$ “.
  - ▶ Üblicherweise wird der **worst case** betrachtet.
    - ▶ Man kann auch untere Schranken erhalten: „jedes vergleichsbasierte Sortierverfahren benötigt im worst case mindestens  $\Omega(n \log n)$  Vergleiche“.



# 5 Effizienz von Algorithmen

## Wie messen wir?

- ▶ Implementieren und Testen auf repräsentativen Eingaben.
  - ▶ Welche Eingaben?
  - ▶ Kann sehr aufwendig sein.
  - ▶ Präzise Resultate wenn sorgfältig durchgeführt.
  - ▶ Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- ▶ Theoretische Analyse in einem **Rechenmodell**.
  - ▶ Gibt **asymptotische Garantien** wie z.B. „dieser Algorithmus läuft immer in Zeit  $\mathcal{O}(n^2)$ “.
  - ▶ Üblicherweise wird der **worst case** betrachtet.
  - ▶ Man kann auch untere Schranken erhalten: „jedes vergleichsbasierte Sortierverfahren benötigt im worst case mindestens  $\Omega(n \log n)$  Vergleiche“.

## 5 Effizienz von Algorithmen

### Eingabelänge

Die theoretischen Schranken werden als Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  von der **Eingabelänge** auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

Die **Eingabelänge** ist z.B.

die Größe der Eingabe (z.B.  $n$ )

die Anzahl der Speicherzellen

die Anzahl der Speicherzellen, die zu einem bestimmten Zeitpunkt belegt sind

die Anzahl der Speicherzellen, die zu einem bestimmten Zeitpunkt belegt sind

die Anzahl der Speicherzellen, die zu einem bestimmten Zeitpunkt belegt sind

## 5 Effizienz von Algorithmen

### Eingabelänge

Die theoretischen Schranken werden als Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  von der **Eingabelänge** auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

Die **Eingabelänge** ist z.B.

## 5 Effizienz von Algorithmen

### Eingabelänge

Die theoretischen Schranken werden als Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  von der **Eingabelänge** auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

Die **Eingabelänge** ist z.B.

- ▶ die Größe der Eingabe (Anzahl an bits)
- ▶ die Anzahl der Argumente

### Example 1

Angenommen  $n$  Zahlen aus dem Bereich  $\{1, \dots, N\}$  sollen sortiert werden. Wir sagen üblicherweise, dass die Eingabelänge  $n$  ist, anstatt z.B.  $n \log N$ , was der Anzahl an Bits entsprechen würde.

## 5 Effizienz von Algorithmen

### Eingabelänge

Die theoretischen Schranken werden als Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  von der **Eingabelänge** auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

Die **Eingabelänge** ist z.B.

- ▶ die Größe der Eingabe (Anzahl an bits)
- ▶ die Anzahl der Argumente

### Example 1

Angenommen  $n$  Zahlen aus dem Bereich  $\{1, \dots, N\}$  sollen sortiert werden. Wir sagen üblicherweise, dass die Eingabelänge  $n$  ist, anstatt z.B.  $n \log N$ , was der Anzahl an Bits entsprechen würde.

## 5 Effizienz von Algorithmen

### Eingabelänge

Die theoretischen Schranken werden als Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  von der **Eingabelänge** auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

Die **Eingabelänge** ist z.B.

- ▶ die Größe der Eingabe (Anzahl an bits)
- ▶ die Anzahl der Argumente

### Example 1

Angenommen  $n$  Zahlen aus dem Bereich  $\{1, \dots, N\}$  sollen sortiert werden. Wir sagen üblicherweise, dass die Eingabelänge  $n$  ist, anstatt z.B.  $n \log N$ , was der Anzahl an Bits entsprechen würde.

## Wie messen wir

Wie lange die Laufzeit in einem bestimmten Rechenmodell  
für einen Algorithmus (z.B. Merge Sort)  
auf einem Rechner mit bestimmten Eigenschaften (z.B. Anzahl an  
Prozessoren, Multithreading, Fortschrittsfortschritt) ...

## Wie messen wir

1. Berechne die Laufzeit in einem idealisierten Rechenmodell (z.B. Random Access Machine (RAM))
2. Berechne Anzahl von Basisoperationen wie z.B. Anzahl an Vergleichen, Multiplikationen, Festplattenzugriffen etc.

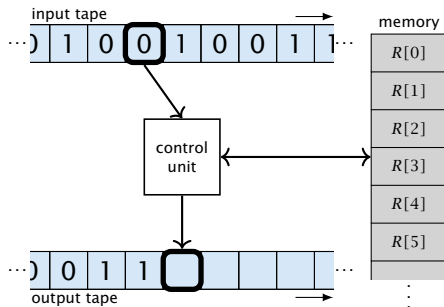


## Wie messen wir

1. Berechne die Laufzeit in einem idealisierten Rechenmodell (z.B. Random Access Machine (RAM))
2. Berechne Anzahl von Basisoperationen wie z.B. Anzahl an Vergleichen, Multiplikationen, Festplattenzugriffen etc.

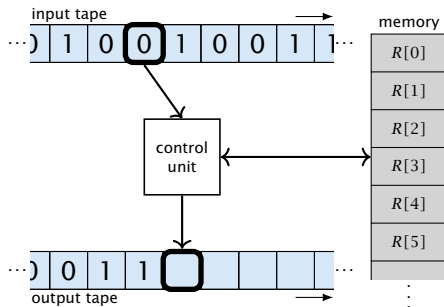
# Random Access Machine (RAM)

- ▶ Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- ▶ Speicher: unendlich viele Register  $R[0], R[1], R[2], \dots$ .
- ▶ Register können beliebige Integer speichern.
- ▶ Indirekte Adressierung.



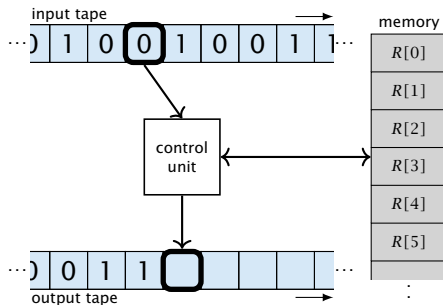
# Random Access Machine (RAM)

- ▶ Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- ▶ Speicher: unendlich viele Register  $R[0], R[1], R[2], \dots$ 
  - ▶ Register können beliebige Integer speichern.
  - ▶ Indirekte Adressierung.



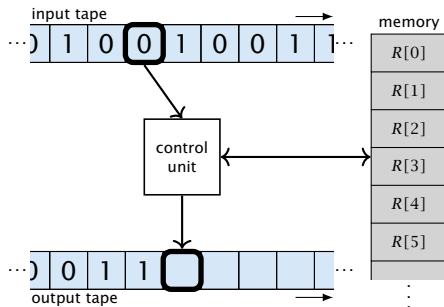
# Random Access Machine (RAM)

- ▶ Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- ▶ Speicher: unendlich viele Register  $R[0], R[1], R[2], \dots$
- ▶ Register können beliebige Integer speichern.
- ▶ Indirekte Adressierung.



# Random Access Machine (RAM)

- ▶ Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- ▶ Speicher: unendlich viele Register  $R[0], R[1], R[2], \dots$
- ▶ Register können beliebige Integer speichern.
- ▶ Indirekte Adressierung.



# Random Access Machine (RAM)

## Operationen

- ▶ Eingabeoperationen (input tape  $\rightarrow R[i]$ )
  - ▶ READ  $i$
- ▶ Ausgabeoperationen ( $R[i] \rightarrow$  output tape)
- ▶ Registertransfers
- ▶ indirekte Adressierung

▶  $R[i]$  = Inhalt des  $i$ -ten Registers in das  $j$ -te Register

▶  $R[j] \leftarrow R[i]$  = Inhalt des  $i$ -ten Registers in das  $j$ -te Register

# Random Access Machine (RAM)

## Operationen

- ▶ Eingabeoperationen (input tape  $\rightarrow R[i]$ )
  - ▶ READ  $i$
- ▶ Ausgabeoperationen ( $R[i] \rightarrow$  output tape)
- ▶ Registertransfers
- ▶ indirekte Adressierung

# Random Access Machine (RAM)

## Operationen

- ▶ Eingabeoperationen (input tape  $\rightarrow R[i]$ )
  - ▶ READ  $i$
- ▶ Ausgabeoperationen ( $R[i] \rightarrow$  output tape)
  - ▶ WRITE  $i$
- ▶ Registertransfers
- ▶ indirekte Adressierung



# Random Access Machine (RAM)

## Operationen

- ▶ Eingabeoperationen (input tape  $\rightarrow R[i]$ )
  - ▶ READ  $i$
- ▶ Ausgabeoperationen ( $R[i] \rightarrow$  output tape)
  - ▶ WRITE  $i$
- ▶ Registertransfers
- ▶ indirekte Adressierung

# Random Access Machine (RAM)

## Operationen

- ▶ Eingabeoperationen (input tape  $\rightarrow R[i]$ )
  - ▶ READ  $i$
- ▶ Ausgabeoperationen ( $R[i] \rightarrow$  output tape)
  - ▶ WRITE  $i$
- ▶ Registertransfers
  - ▶  $R[j] := R[i]$
  - ▶  $R[j] := 4$
- ▶ indirekte Adressierung

# Random Access Machine (RAM)

## Operationen

- ▶ Eingabeoperationen (input tape  $\rightarrow R[i]$ )
  - ▶ READ  $i$
- ▶ Ausgabeoperationen ( $R[i] \rightarrow$  output tape)
  - ▶ WRITE  $i$
- ▶ Registertransfers
  - ▶  $R[j] := R[i]$
  - ▶  $R[j] := 4$
- ▶ indirekte Adressierung

# Random Access Machine (RAM)

## Operationen

- ▶ Eingabeoperationen (input tape  $\rightarrow R[i]$ )
  - ▶ READ  $i$
- ▶ Ausgabeoperationen ( $R[i] \rightarrow$  output tape)
  - ▶ WRITE  $i$
- ▶ Registertransfers
  - ▶  $R[j] := R[i]$
  - ▶  $R[j] := 4$
- ▶ indirekte Adressierung

# Random Access Machine (RAM)

## Operationen

- ▶ Eingabeoperationen (input tape  $\rightarrow R[i]$ )
  - ▶ READ  $i$
- ▶ Ausgabeoperationen ( $R[i] \rightarrow$  output tape)
  - ▶ WRITE  $i$
- ▶ Registertransfers
  - ▶  $R[j] := R[i]$
  - ▶  $R[j] := 4$
- ▶ **indirekte** Adressierung
  - ▶  $R[j] := R[R[i]]$   
lädt den Inhalt des  $R[i]$ -ten Registres in das  $j$ -te Register.
  - ▶  $R[R[i]] := R[j]$   
lädt den Inhalt des  $j$ -ten Registers in das  $R[i]$ -te Register

# Random Access Machine (RAM)

## Operationen

- ▶ Eingabeoperationen (input tape  $\rightarrow R[i]$ )
  - ▶ READ  $i$
- ▶ Ausgabeoperationen ( $R[i] \rightarrow$  output tape)
  - ▶ WRITE  $i$
- ▶ Registertransfers
  - ▶  $R[j] := R[i]$
  - ▶  $R[j] := 4$
- ▶ **indirekte** Adressierung
  - ▶  $R[j] := R[R[i]]$   
lädt den Inhalt des  $R[i]$ -ten Registres in das  $j$ -te Register.
  - ▶  $R[R[i]] := R[j]$   
lädt den Inhalt des  $j$ -ten Registers in das  $R[i]$ -te Register

# Random Access Machine (RAM)

## Operationen

- ▶ Eingabeoperationen (input tape  $\rightarrow R[i]$ )
  - ▶ READ  $i$
- ▶ Ausgabeoperationen ( $R[i] \rightarrow$  output tape)
  - ▶ WRITE  $i$
- ▶ Registertransfers
  - ▶  $R[j] := R[i]$
  - ▶  $R[j] := 4$
- ▶ **indirekte** Adressierung
  - ▶  $R[j] := R[R[i]]$   
lädt den Inhalt des  $R[i]$ -ten Registres in das  $j$ -te Register.
  - ▶  $R[R[i]] := R[j]$   
lädt den Inhalt des  $j$ -ten Registers in das  $R[i]$ -te Register

# Random Access Machine (RAM)

## Operationen

- ▶ Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - ▶ `jump  $x$`   
springe zur Position  $x$  im Programm  
setze Befehlszähler auf  $x$   
der nächste Befehl wird aus Register  $R[x]$  gelesen
  - ▶ `jumpz  $x R[i]$`   
springe zu  $x$  falls  $R[i] = 0$   
falls nicht wird der Befehlszähler um 1 erhöht
  - ▶ `jumpi  $i$`   
springe zu  $R[i]$  (indirekter Sprung);
- ▶ arithmetische Operationen:  $+$ ,  $-$ ,  $\times$ ,  $/$



# Random Access Machine (RAM)

## Operationen

- ▶ Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - ▶ `jump  $x$` 
    - springe zur Position  $x$  im Programm
    - setze Befehlszähler auf  $x$
    - der nächste Befehl wird aus Register  $R[x]$  gelesen
  - ▶ `jumpz  $x$   $R[i]$` 
    - springe zu  $x$  falls  $R[i] = 0$
    - falls nicht wird der Befehlszähler um 1 erhöht
  - ▶ `jumpi  $i$` 
    - springe zu  $R[i]$  (indirekter Sprung);
- ▶ arithmetische Operationen:  $+$ ,  $-$ ,  $\times$ ,  $/$

# Random Access Machine (RAM)

## Operationen

- ▶ Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - ▶ `jump  $x$` 
    - springe zur Position  $x$  im Programm
    - setze Befehlszähler auf  $x$
    - der nächste Befehl wird aus Register  $R[x]$  gelesen
  - ▶ `jumpz  $x R[i]$` 
    - springe zu  $x$  falls  $R[i] = 0$
    - falls nicht wird der Befehlszähler um 1 erhöht
  - ▶ `jumpi  $i$` 
    - springe zu  $R[i]$  (indirekter Sprung);
- ▶ arithmetische Operationen:  $+$ ,  $-$ ,  $\times$ ,  $/$

# Random Access Machine (RAM)

## Operationen

- ▶ Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - ▶ `jump  $x$`   
springe zur Position  $x$  im Programm  
setze Befehlszähler auf  $x$   
der nächste Befehl wird aus Register  $R[x]$  gelesen
  - ▶ `jumpz  $x R[i]$`   
springe zu  $x$  falls  $R[i] = 0$   
falls nicht wird der Befehlszähler um 1 erhöht
  - ▶ `jumpi  $i$`   
springe zu  $R[i]$  (indirekter Sprung);
- ▶ arithmetische Operationen:  $+$ ,  $-$ ,  $\times$ ,  $/$

# Random Access Machine (RAM)

## Operationen

- ▶ Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - ▶ jump  $x$   
springe zur Position  $x$  im Programm  
setze Befehlszähler auf  $x$   
der nächste Befehl wird aus Register  $R[x]$  gelesen
  - ▶ jumpz  $x R[i]$   
springe zu  $x$  falls  $R[i] = 0$   
falls nicht wird der Befehlszähler um 1 erhöht
  - ▶ jumpi  $i$   
springe zu  $R[i]$  (indirekter Sprung);
- ▶ arithmetische Operationen:  $+$ ,  $-$ ,  $\times$ ,  $/$ 
  - ▶  $R[i] := R[j] + R[k];$
  - ▶  $R[i] := -R[k];$

# Random Access Machine (RAM)

## Operationen

- ▶ Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - ▶ jump  $x$   
springe zur Position  $x$  im Programm  
setze Befehlszähler auf  $x$   
der nächste Befehl wird aus Register  $R[x]$  gelesen
  - ▶ jumpz  $x R[i]$   
springe zu  $x$  falls  $R[i] = 0$   
falls nicht wird der Befehlszähler um 1 erhöht
  - ▶ jumpi  $i$   
springe zu  $R[i]$  (indirekter Sprung);
- ▶ arithmetische Operationen:  $+$ ,  $-$ ,  $\times$ ,  $/$ 
  - ▶  $R[i] := R[j] + R[k];$   
 $R[i] := -R[k];$

# Rechenmodell

Man nimmt normalerweise an, dass jeder Befehl eine Zeiteinheit kostet.

# Komplexitätsschranken

Es gibt **unterschiedliche Komplexitätsschranken**:

- ▶ **best-case** Komplexität:

$$C_{bc}(n) := \min\{C(x) \mid |x| = n\}$$

Normalerweise einfach zu analysieren; nicht sehr hilfreich

- ▶ **worst-case** Komplexität:

$$C_{wc}(n) := \max\{C(x) \mid |x| = n\}$$

Standard. Manchmal zu pessimistisch.

- ▶ **average case** Komplexität:

$$C_{avg}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

Manchmal schwierig zu analysieren.

# Komplexitätsschranken

Es gibt **unterschiedliche Komplexitätsschranken**:

- ▶ **best-case** Komplexität:

$$C_{bc}(n) := \min\{C(x) \mid |x| = n\}$$

Normalerweise einfach zu analysieren; nicht sehr hilfreich

- ▶ **worst-case** Komplexität:

$$C_{wc}(n) := \max\{C(x) \mid |x| = n\}$$

Standard. Manchmal zu pessimistisch.

- ▶ **average case** Komplexität:

$$C_{avg}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

Manchmal schwierig zu analysieren.



# Komplexitätsschranken

Es gibt **unterschiedliche Komplexitätsschranken**:

- ▶ **best-case** Komplexität:

$$C_{bc}(n) := \min\{C(x) \mid |x| = n\}$$

Normalerweise einfach zu analysieren; nicht sehr hilfreich

- ▶ **worst-case** Komplexität:

$$C_{wc}(n) := \max\{C(x) \mid |x| = n\}$$

Standard. Manchmal zu pessimistisch.

- ▶ **average case** Komplexität:

$$C_{avg}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

Manchmal schwierig zu analysieren.

# Komplexitätsschranken

Es gibt **unterschiedliche Komplexitätsschranken**:

- ▶ **best-case** Komplexität:

$$C_{bc}(n) := \min\{C(x) \mid |x| = n\}$$

Normalerweise einfach zu analysieren; nicht sehr hilfreich

- ▶ **worst-case** Komplexität:

$$C_{wc}(n) := \max\{C(x) \mid |x| = n\}$$

Standard. Manchmal zu pessimistisch.

- ▶ **average case** Komplexität:

$$C_{avg}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

Manchmal schwierig zu analysieren.

# Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine **asymptotische Klassifikation** der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

# Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine **asymptotische Klassifikation** der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

- ▶ Wir interessieren uns für Laufzeiten bei großen Werten von  $n$ . Konstante additive Terme sind dann unwichtig.
- ▶ Eine superexakte Analyse (e.g. das *exakte* Zählen der Operationen auf einer RAM) ist schwierig, und würde die Resultate nicht verbessern, da das Rechenmodell die Realität nicht so exakt abbildet.
- ▶ Ein linearer speed-up (z.B. um einen konstanten Faktor) läßt sich z.B. erreichen wenn man den Algorithmus auf einem schnelleren Rechner laufen läßt.
- ▶ Laufzeiten sollte man durch einfache Funktionen ausdrücken können.

# Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine **asymptotische Klassifikation** der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

- ▶ Wir interessieren uns für Laufzeiten bei großen Werten von  $n$ . Konstante additive Terme sind dann unwichtig.
- ▶ Eine superexakte Analyse (e.g. das *exakte* Zählen der Operationen auf einer RAM) ist schwierig, und würde die Resultate nicht verbessern, da das Rechenmodell die Realität nicht so exakt abbildet.
- ▶ Ein linearer speed-up (z.B. um einen konstanten Faktor) läßt sich z.B. erreichen wenn man den Algorithmus auf einem schnelleren Rechner laufen läßt.
- ▶ Laufzeiten sollte man durch einfache Funktionen ausdrücken können.

# Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine **asymptotische Klassifikation** der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

- ▶ Wir interessieren uns für Laufzeiten bei großen Werten von  $n$ . Konstante additive Terme sind dann unwichtig.
- ▶ Eine superexakte Analyse (e.g. das *exakte* Zählen der Operationen auf einer RAM) ist schwierig, und würde die Resultate nicht verbessern, da das Rechenmodell die Realität nicht so exakt abbildet.
- ▶ Ein linearer speed-up (z.B. um einen konstanten Faktor) läßt sich z.B. erreichen wenn man den Algorithmus auf einem schnelleren Rechner laufen läßt.
- ▶ Laufzeiten sollte man durch einfache Funktionen ausdrücken können.

# Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine **asymptotische Klassifikation** der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

- ▶ Wir interessieren uns für Laufzeiten bei großen Werten von  $n$ . Konstante additive Terme sind dann unwichtig.
- ▶ Eine superexakte Analyse (e.g. das *exakte* Zählen der Operationen auf einer RAM) ist schwierig, und würde die Resultate nicht verbessern, da das Rechenmodell die Realität nicht so exakt abbildet.
- ▶ Ein linearer speed-up (z.B. um einen konstanten Faktor) läßt sich z.B. erreichen wenn man den Algorithmus auf einem schnelleren Rechner laufen läßt.
- ▶ Laufzeiten sollte man durch einfache Funktionen ausdrücken können.

# Asymptotische Notation

## Formale Definition

Sei  $f$  eine Funktion von  $\mathbb{N}$  nach  $\mathbb{R}^+$ .

- ▶  $\mathcal{O}(f) = \{g \mid \exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$   
(Funktionen die asymptotisch **nicht schneller** als  $f$  wachsen)



# Asymptotische Notation

## Formale Definition

Sei  $f$  eine Funktion von  $\mathbb{N}$  nach  $\mathbb{R}^+$ .

- ▶  $\mathcal{O}(f) = \{g \mid \exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$   
(Funktionen die asymptotisch **nicht schneller** als  $f$  wachsen)
- ▶  $\Omega(f) = \{g \mid \exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$   
(Funktionen die asymptotisch **nicht langsamer** als  $f$  wachsen)

# Asymptotische Notation

## Formale Definition

Sei  $f$  eine Funktion von  $\mathbb{N}$  nach  $\mathbb{R}^+$ .

- ▶  $\mathcal{O}(f) = \{g \mid \exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$   
(Funktionen die asymptotisch **nicht schneller** als  $f$  wachsen)
- ▶  $\Omega(f) = \{g \mid \exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$   
(Funktionen die asymptotisch **nicht langsamer** als  $f$  wachsen)
- ▶  $\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$   
(Funktionen die asymptotisch **gleiches** Wachstum wie  $f$  haben)

# Asymptotische Notation

## Formale Definition

Sei  $f$  eine Funktion von  $\mathbb{N}$  nach  $\mathbb{R}^+$ .

- ▶  $\mathcal{O}(f) = \{g \mid \exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$   
(Funktionen die asymptotisch **nicht schneller** als  $f$  wachsen)
- ▶  $\Omega(f) = \{g \mid \exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$   
(Funktionen die asymptotisch **nicht langsamer** als  $f$  wachsen)
- ▶  $\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$   
(Funktionen die asymptotisch **gleiches** Wachstum wie  $f$  haben)
- ▶  $o(f) = \{g \mid \forall c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$   
(Funktionen die asymptotische **langsamer** als  $f$  wachsen)

# Asymptotische Notation

## Formale Definition

Sei  $f$  eine Funktion von  $\mathbb{N}$  nach  $\mathbb{R}^+$ .

- ▶  $\mathcal{O}(f) = \{g \mid \exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$   
(Funktionen die asymptotisch **nicht schneller** als  $f$  wachsen)
- ▶  $\Omega(f) = \{g \mid \exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$   
(Funktionen die asymptotisch **nicht langsamer** als  $f$  wachsen)
- ▶  $\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$   
(Funktionen die asymptotisch **gleiches** Wachstum wie  $f$  haben)
- ▶  $o(f) = \{g \mid \forall c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$   
(Funktionen die asymptotische **langsamer** als  $f$  wachsen)
- ▶  $\omega(f) = \{g \mid \forall c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$   
(Funktionen die asymptotisch **schneller** als  $f$  wachsen)

# Asymptotische Notation

Äquivalente Definition mit Grenzwerten (gilt nur falls der jeweilige Grenzwert existiert).  $f$  und  $g$  seien Funktionen von  $\mathbb{N}_0$  nach  $\mathbb{R}_0^+$ .

$$\blacktriangleright g \in \mathcal{O}(f): 0 \leq \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} < \infty$$

# Asymptotische Notation

Äquivalente Definition mit Grenzwerten (**gilt nur falls der jeweilige Grenzwert existiert**).  $f$  und  $g$  seien Funktionen von  $\mathbb{N}_0$  nach  $\mathbb{R}_0^+$ .

$$\blacktriangleright g \in \mathcal{O}(f): 0 \leq \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} < \infty$$

$$\blacktriangleright g \in \Omega(f): 0 < \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \leq \infty$$

# Asymptotische Notation

Äquivalente Definition mit Grenzwerten (**gilt nur falls der jeweilige Grenzwert existiert**).  $f$  und  $g$  seien Funktionen von  $\mathbb{N}_0$  nach  $\mathbb{R}_0^+$ .

$$\blacktriangleright g \in \mathcal{O}(f): 0 \leq \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} < \infty$$

$$\blacktriangleright g \in \Omega(f): 0 < \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \leq \infty$$

$$\blacktriangleright g \in \Theta(f): 0 < \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} < \infty$$

# Asymptotische Notation

Äquivalente Definition mit Grenzwerten (gilt nur falls der jeweilige Grenzwert existiert).  $f$  und  $g$  seien Funktionen von  $\mathbb{N}_0$  nach  $\mathbb{R}_0^+$ .

$$\blacktriangleright g \in \mathcal{O}(f): 0 \leq \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} < \infty$$

$$\blacktriangleright g \in \Omega(f): 0 < \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \leq \infty$$

$$\blacktriangleright g \in \Theta(f): 0 < \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} < \infty$$

$$\blacktriangleright g \in o(f): \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$$



# Asymptotische Notation

Äquivalente Definition mit Grenzwerten (gilt nur falls der jeweilige Grenzwert existiert).  $f$  und  $g$  seien Funktionen von  $\mathbb{N}_0$  nach  $\mathbb{R}_0^+$ .

$$\blacktriangleright g \in \mathcal{O}(f): 0 \leq \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} < \infty$$

$$\blacktriangleright g \in \Omega(f): 0 < \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \leq \infty$$

$$\blacktriangleright g \in \Theta(f): 0 < \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} < \infty$$

$$\blacktriangleright g \in o(f): \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$$

$$\blacktriangleright g \in \omega(f): \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \infty$$

## Missbrauch dieser Notation

1. Man schreibt  $f = \mathcal{O}(g)$ , anstatt  $f \in \mathcal{O}(g)$ . Dies ist **keine** Gleichheit (wie kann eine Funktion das gleiche wie eine Funktionsmenge sein?).
2. Man schreibt  $f(n) = \mathcal{O}(g(n))$ , anstatt  $f \in \mathcal{O}(g)$ , with  $f: \mathbb{N} \rightarrow \mathbb{R}^+, n \mapsto f(n)$ , and  $g: \mathbb{N} \rightarrow \mathbb{R}^+, n \mapsto g(n)$ .
3. Man schreibt  $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$ , anstatt  $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$ .

## Missbrauch dieser Notation

1. Man schreibt  $f = \mathcal{O}(g)$ , anstatt  $f \in \mathcal{O}(g)$ . Dies ist **keine** Gleichheit (wie kann eine Funktion das gleiche wie eine Funktionsmenge sein?).
2. Man schreibt  $f(n) = \mathcal{O}(g(n))$ , anstatt  $f \in \mathcal{O}(g)$ , with  $f: \mathbb{N} \rightarrow \mathbb{R}^+, n \mapsto f(n)$ , and  $g: \mathbb{N} \rightarrow \mathbb{R}^+, n \mapsto g(n)$ .
3. Man schreibt  $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$ , anstatt  $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$ .

## Missbrauch dieser Notation

1. Man schreibt  $f = \mathcal{O}(g)$ , anstatt  $f \in \mathcal{O}(g)$ . Dies ist **keine** Gleichheit (wie kann eine Funktion das gleiche wie eine Funktionsmenge sein?).
2. Man schreibt  $f(n) = \mathcal{O}(g(n))$ , anstatt  $f \in \mathcal{O}(g)$ , with  $f: \mathbb{N} \rightarrow \mathbb{R}^+, n \mapsto f(n)$ , and  $g: \mathbb{N} \rightarrow \mathbb{R}^+, n \mapsto g(n)$ .
3. Man schreibt  $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$ , anstatt  $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$ .

# Asymptotische Notation

Man kann einen Ausdruck mit asymptotischer Notation als **Menge** ansehen:

$$n^2 \cdot \mathcal{O}(n) + \mathcal{O}(\log n)$$

repräsentiert

$$\{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid f(n) = n^2 \cdot g(n) + h(n)$$

$$\text{mit } g(n) \in \mathcal{O}(n) \text{ und } h(n) \in \mathcal{O}(\log n)\}$$

## Lemma 2

Seien  $f, g$  Funktionen mit  $\exists n_0 > 0 \forall n \geq n_0 : f(n) > 0$  (das gleiche für  $g$ ). Dann

- ▶  $c \cdot f(n) = \mathcal{O}(f(n))$  für eine beliebige Konstante  $c$
- ▶  $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- ▶  $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- ▶  $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

*Alle obigen Relationen gelten auch für  $\Omega$  und  $\Theta$ .*

# Asymptotische Notation

## Lemma 2

Seien  $f, g$  Funktionen mit  $\exists n_0 > 0 \forall n \geq n_0 : f(n) > 0$  (das gleiche für  $g$ ). Dann

- ▶  $c \cdot f(n) = \mathcal{O}(f(n))$  für eine beliebige Konstante  $c$
- ▶  $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- ▶  $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- ▶  $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

*Alle obigen Relationen gelten auch für  $\Omega$  und  $\Theta$ .*

# Asymptotische Notation

## Lemma 2

Seien  $f, g$  Funktionen mit  $\exists n_0 > 0 \forall n \geq n_0 : f(n) > 0$  (das gleiche für  $g$ ). Dann

- ▶  $c \cdot f(n) = \mathcal{O}(f(n))$  für eine beliebige Konstante  $c$
- ▶  $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- ▶  $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- ▶  $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

*Alle obigen Relationen gelten auch für  $\Omega$  und  $\Theta$ .*



# Asymptotische Notation

## Lemma 2

Seien  $f, g$  Funktionen mit  $\exists n_0 > 0 \forall n \geq n_0 : f(n) > 0$  (das gleiche für  $g$ ). Dann

- ▶  $c \cdot f(n) = \mathcal{O}(f(n))$  für eine beliebige Konstante  $c$
- ▶  $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- ▶  $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- ▶  $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

*Alle obigen Relationen gelten auch für  $\Omega$  und  $\Theta$ .*

# Asymptotische Notation

## Lemma 2

Seien  $f, g$  Funktionen mit  $\exists n_0 > 0 \forall n \geq n_0 : f(n) > 0$  (das gleiche für  $g$ ). Dann

- ▶  $c \cdot f(n) = \mathcal{O}(f(n))$  für eine beliebige Konstante  $c$
- ▶  $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- ▶  $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- ▶  $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

Alle obigen Relationen gelten auch für  $\Omega$  und  $\Theta$ .

# Rechenregel für $\mathcal{O}$ -Notation

Zu zeigen:

$$T_1(n) + T_2(n) = \mathcal{O}(\max(f(n), g(n)))$$

für  $T_1(n) \in \mathcal{O}(f(n))$  und  $T_2(n) \in \mathcal{O}(g(n))$ .

- ▶ da  $T_1(n) = \mathcal{O}(f(n))$ , gibt es  $c_1 > 0$  und  $n_1 \in \mathbb{N}$  mit  $T_1(n) \leq c_1 f(n)$  für  $n \geq n_1$
- ▶ da  $T_2(n) = \mathcal{O}(g(n))$ , gibt es  $c_2 > 0$  und  $n_2 \in \mathbb{N}$  mit  $T_2(n) \leq c_2 g(n)$  für  $n \geq n_2$
- ▶ Setze  $n_0 := \max(n_1, n_2)$ , dann ist für  $n \geq n_0$

$$\begin{aligned} T_1(n) + T_2(n) &\leq c_1 f(n) + c_2 g(n) \\ &\leq (c_1 + c_2) \max(f(n), g(n)) \quad \checkmark \end{aligned}$$

# Rechenregel für $\mathcal{O}$ -Notation

Zu zeigen:

$$T_1(n) + T_2(n) = \mathcal{O}(\max(f(n), g(n)))$$

für  $T_1(n) \in \mathcal{O}(f(n))$  und  $T_2(n) \in \mathcal{O}(g(n))$ .

- ▶ da  $T_1(n) = \mathcal{O}(f(n))$ , gibt es  $c_1 > 0$  und  $n_1 \in \mathbb{N}$  mit  $T_1(n) \leq c_1 f(n)$  für  $n \geq n_1$
- ▶ da  $T_2(n) = \mathcal{O}(g(n))$ , gibt es  $c_2 > 0$  und  $n_2 \in \mathbb{N}$  mit  $T_2(n) \leq c_2 g(n)$  für  $n \geq n_2$
- ▶ Setze  $n_0 := \max(n_1, n_2)$ , dann ist für  $n \geq n_0$

$$\begin{aligned} T_1(n) + T_2(n) &\leq c_1 f(n) + c_2 g(n) \\ &\leq (c_1 + c_2) \max(f(n), g(n)) \quad \checkmark \end{aligned}$$

# Laufzeiten

Funktion $f(n)$	Eingabelänge $n$							
	10	$10^2$	$10^3$	$10^4$	$10^5$	$10^6$	$10^7$	$10^8$
$\log n$	33ns	66ns	0.1 $\mu$ s	0.1 $\mu$ s	0.2 $\mu$ s	0.2 $\mu$ s	0.2 $\mu$ s	0.3 $\mu$ s
$\sqrt{n}$	32ns	0.1 $\mu$ s	0.3 $\mu$ s	1 $\mu$ s	3.1 $\mu$ s	10 $\mu$ s	31 $\mu$ s	0.1ms
$n$	100ns	1 $\mu$ s	10 $\mu$ s	0.1ms	1ms	10ms	0.1s	1s
$n \log n$	0.3 $\mu$ s	6.6 $\mu$ s	0.1ms	1.3ms	16ms	0.2s	2.3s	27s
$n^{3/2}$	0.3 $\mu$ s	10 $\mu$ s	0.3ms	10ms	0.3s	10s	5.2min	2.7h
$n^2$	1 $\mu$ s	0.1ms	10ms	1s	1.7min	2.8h	11d	3.2y
$n^3$	10 $\mu$ s	10ms	10s	2.8h	115d	317y	$3.2 \cdot 10^5$ y	
$1.1^n$	26ns	0.1ms	$7.8 \cdot 10^{25}$ y					
$2^n$	10 $\mu$ s	$4 \cdot 10^{14}$ y						
$n!$	36ms	$3 \cdot 10^{142}$ y						

1 Operation = 10ns; 100MHz

Alter des Universums: ca.  $13.8 \cdot 10^9$ y

# Typische Laufzeitklassen

$\mathcal{O}$ -Notation erlaubt **Klassifizierung** der Effizienz von Algorithmen

## $\Theta(1)$ : konstante Laufzeit

- ▶ unabhängig von Problemgröße
- ▶ *Beispiel*: Löschen von erstem Element in verketteter Liste

## $\Theta(\log n)$ : logarithmische Laufzeit

- ▶ Laufzeit wächst langsamer als Problemgröße
- ▶ typisch für Divide & Conquer Algorithmen
- ▶ *Beispiel*: Suchen in sortierter Liste

## $\Theta(n)$ : lineare Laufzeit

- ▶ Laufzeit wächst vergleichbar zur Problemgröße
- ▶ jedes Eingabe-Element erfordert  $\mathcal{O}(1)$  Arbeit
- ▶ *Beispiele*: Suchen in unsortierter Liste, Löschen von Element im Array

# Typische Laufzeitklassen

## $\Theta(n \log n)$ : “loglinear” Laufzeit

- ▶ Laufzeit wächst schneller als Problemgröße
- ▶ typisch für Divide & Conquer
- ▶ *Beispiele*: Quicksort, FFT

## $\Theta(n^2)$ : quadratische Laufzeit

- ▶ typisch für Algorithmen, die Element paarweise kombinieren
- ▶ *Beispiele*: Insertion Sort, Matrix-Vektor Multiplikation

## $\Theta(n^3)$ : kubische Laufzeit

- ▶ *Beispiel*: Matrix-Matrix Multiplikation

## $\Theta(2^n)$ : exponentielle Laufzeit

- ▶ auch als “unlösbar” bezeichnet (intractable)
- ▶ *Beispiel*: Traveling Salesman (kürzeste Route, so dass alle Städte exakt einmal besucht)

## Hinweise

- ▶ Man sollte asymptotische Notation nicht in Induktionsbeweisen verwenden.
- ▶ Für beliebige Konstanten  $a, b$  gilt  $\log_a n = \Theta(\log_b n)$ . Deshalb ignorieren wir die Basis des Algorithmus in asymptotischer Notation.
- ▶ Für diese Vorlesung:  $\log n = \log_2 n$ , d.h., wir nehmen 2 als Standardbasis für den Logarithmus.



## Hinweise

- ▶ Man sollte asymptotische Notation nicht in Induktionsbeweisen verwenden.
- ▶ Für beliebige Konstanten  $a, b$  gilt  $\log_a n = \Theta(\log_b n)$ . Deshalb ignorieren wir die Basis des Algorithmus in asymptotischer Notation.
- ▶ Für diese Vorlesung:  $\log n = \log_2 n$ , d.h., wir nehmen 2 als Standardbasis für den Logarithmus.

## Hinweise

- ▶ Man sollte asymptotische Notation nicht in Induktionsbeweisen verwenden.
- ▶ Für beliebige Konstanten  $a, b$  gilt  $\log_a n = \Theta(\log_b n)$ . Deshalb ignorieren wir die Basis des Algorithmus in asymptotischer Notation.
- ▶ Für diese Vorlesung:  $\log n = \log_2 n$ , d.h., wir nehmen 2 als Standardbasis für den Logarithmus.

# Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- ▶ Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes  $n$  überlegen.
- ▶ Aber:

# Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- ▶ Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes  $n$  überlegen.
- ▶ **Aber:**
  - ▶ Algorithmus A: Laufzeit  $f(n) = 1000 \log n = \mathcal{O}(\log n)$ .
  - ▶ Algorithmus B: Laufzeit  $g(n) = \log^2 n$ .

Es gilt  $f = o(g)$ . Aber solange  $\log n \leq 1000$  ist Algorithmus B effizienter.

# Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- ▶ Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes  $n$  überlegen.
- ▶ **Aber:**
  - ▶ Algorithmus A: Laufzeit  $f(n) = 1000 \log n = \mathcal{O}(\log n)$ .
  - ▶ Algorithmus B: Laufzeit  $g(n) = \log^2 n$ .

Es gilt  $f = o(g)$ . Aber solange  $\log n \leq 1000$  ist Algorithmus B effizienter.

# Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- ▶ Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes  $n$  überlegen.
- ▶ **Aber:**
  - ▶ Algorithmus A: Laufzeit  $f(n) = 1000 \log n = \mathcal{O}(\log n)$ .
  - ▶ Algorithmus B: Laufzeit  $g(n) = \log^2 n$ .

Es gilt  $f = o(g)$ . Aber solange  $\log n \leq 1000$  ist Algorithmus B effizienter.

# Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- ▶ Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes  $n$  überlegen.
- ▶ **Aber:**
  - ▶ Algorithmus A: Laufzeit  $f(n) = 1000 \log n = \mathcal{O}(\log n)$ .
  - ▶ Algorithmus B: Laufzeit  $g(n) = \log^2 n$ .

Es gilt  $f = o(g)$ . Aber solange  $\log n \leq 1000$  ist Algorithmus B effizienter.

# Komplexität der elementaren Bausteine

## Elementarer Verarbeitungsschritt:

- ▶  $\mathcal{O}(1)$

## Sequenz:

- ▶ Addition in  $\mathcal{O}$ -Notation

## Bedingter Verarbeitungsschritt:

- ▶ Maximum von Komplexität von if und else Block, sowie
- ▶  $\mathcal{O}(1)$  für Auswertung der Bedingung

## Wiederholung:

- ▶ Anzahl Wiederholungen multipliziert mit Komplexität Schleifenkörper, sowie
- ▶ Anzahl Wiederholungen + 1 multipliziert mit  $\mathcal{O}(1)$  für Auswertung der Schleifenbedingung



# Komplexität Datenstrukturenoperationen

## Feld via Array:

- ▶ elementAt  $\mathcal{O}(1)$ , insert  $\mathcal{O}(n)$ , erase  $\mathcal{O}(n)$

## Feld via LinkedList:

- ▶ elementAt  $\mathcal{O}(n)$ , insert  $\mathcal{O}(n)$ , erase  $\mathcal{O}(n)$

## Stack via Array:

- ▶ push, pop, top alle  $\mathcal{O}(1)$

## Stack via LinkedList:

- ▶ push, pop, top alle  $\mathcal{O}(1)$

## Queue via LinkedList:

- ▶ enqueue, dequeue beide  $\mathcal{O}(1)$

# Sortieren durch Einfügen

**Gegeben:** eine Folge von ganzen Zahlen.

**Gesucht:** die zugehörige aufsteigend sortierte Folge.

**Idee:**

- ▶ speichere die Folge in einem Feld ab;
- ▶ lege ein weiteres Feld an;
- ▶ füge der Reihe nach jedes Element des ersten Felds an der richtigen Stelle in das zweite Feld ein!

⇒ Sortieren durch Einfügen (**InsertionSort**)

# Beispiel

17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

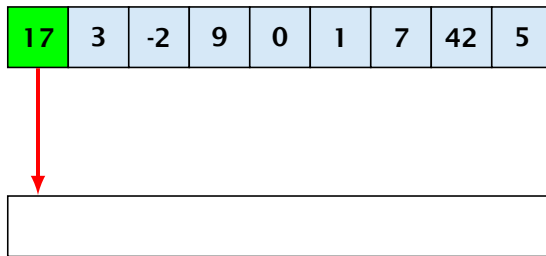
--

# Beispiel

17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

--

# Beispiel

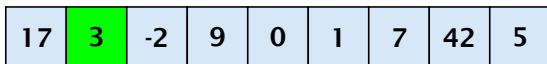


# Beispiel

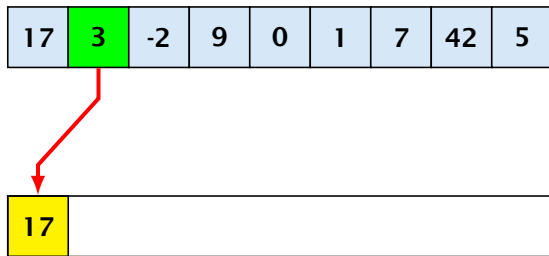
17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

17								
----	--	--	--	--	--	--	--	--

# Beispiel

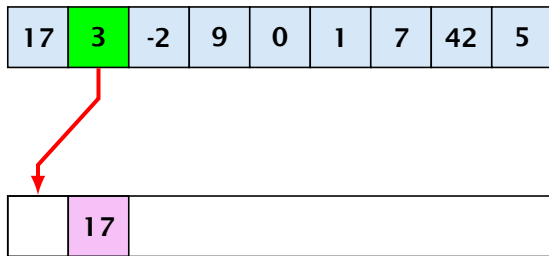


# Beispiel





# Beispiel

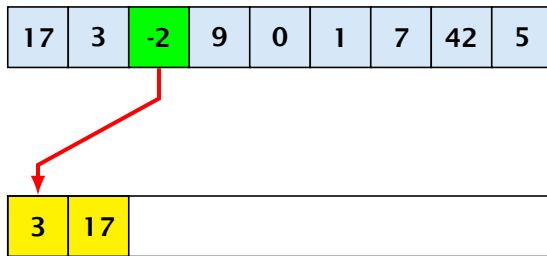


# Beispiel

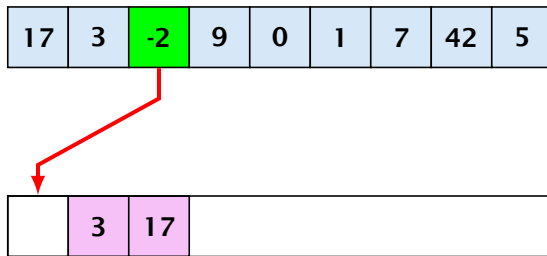
17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

3	17							
---	----	--	--	--	--	--	--	--

# Beispiel



# Beispiel

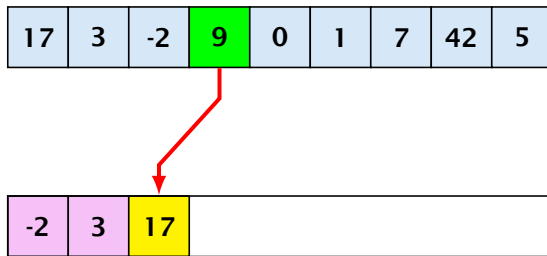


# Beispiel

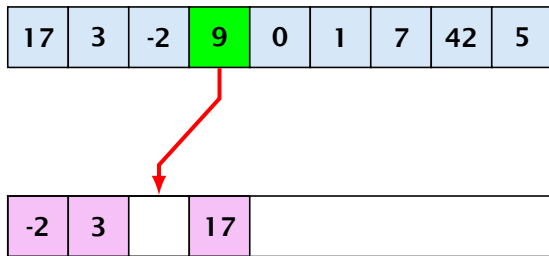
17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

-2	3	17						
----	---	----	--	--	--	--	--	--

# Beispiel



# Beispiel



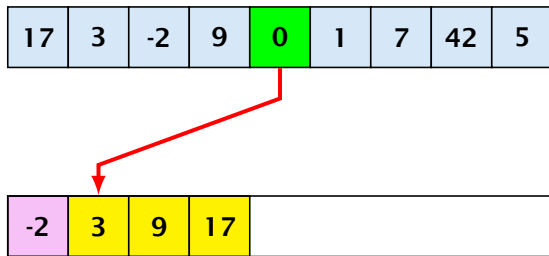
# Beispiel

17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

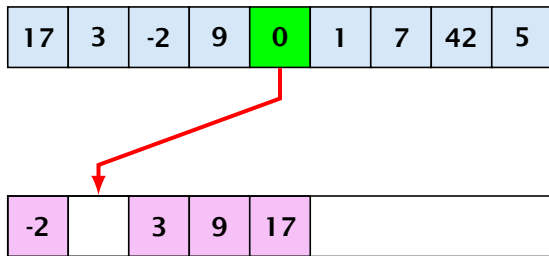
-2	3	9	17					
----	---	---	----	--	--	--	--	--



# Beispiel



# Beispiel

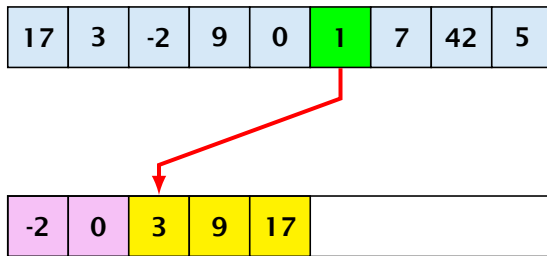


# Beispiel

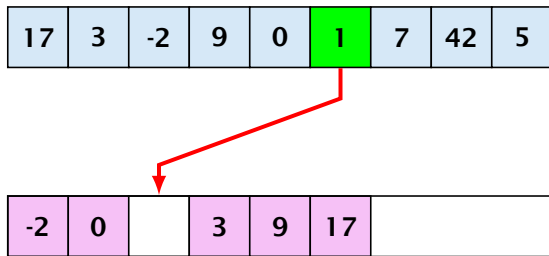
17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

-2	0	3	9	17				
----	---	---	---	----	--	--	--	--

# Beispiel



# Beispiel

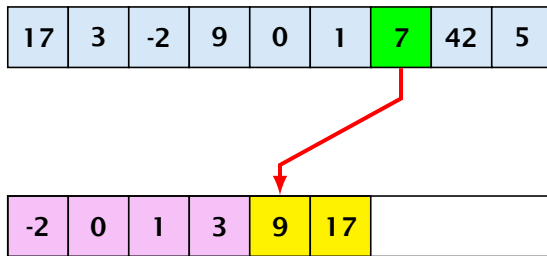


# Beispiel

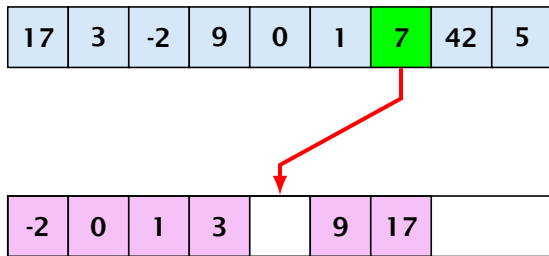
17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

-2	0	1	3	9	17			
----	---	---	---	---	----	--	--	--

# Beispiel



# Beispiel



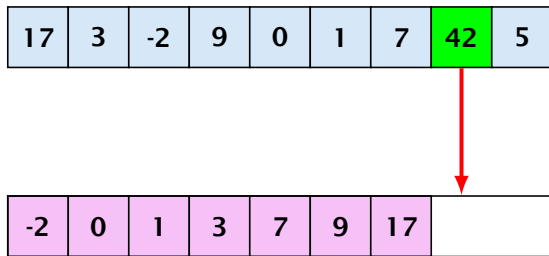


# Beispiel

17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

-2	0	1	3	7	9	17	
----	---	---	---	---	---	----	--

# Beispiel

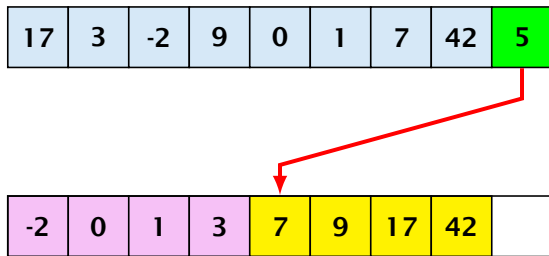


# Beispiel

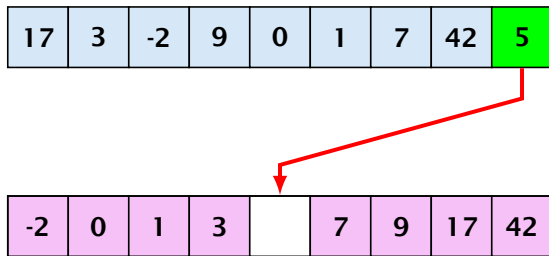
17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

-2	0	1	3	7	9	17	42	
----	---	---	---	---	---	----	----	--

# Beispiel



# Beispiel



# Beispiel

17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

-2	0	1	3	5	7	9	17	42
----	---	---	---	---	---	---	----	----

# Beispiel

Wir brauchen kein zweites Array:

17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

# Beispiel

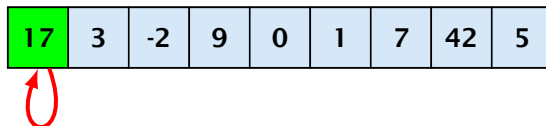
Wir brauchen kein zweites Array:

17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---



# Beispiel

Wir brauchen kein zweites Array:



# Beispiel

Wir brauchen kein zweites Array:

17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

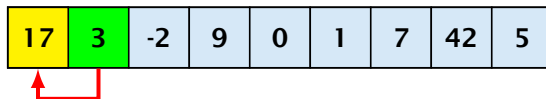
# Beispiel

Wir brauchen kein zweites Array:

17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

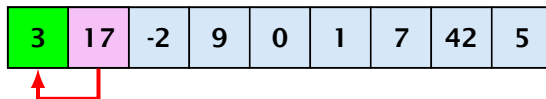
# Beispiel

Wir brauchen kein zweites Array:



# Beispiel

Wir brauchen kein zweites Array:



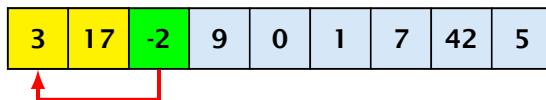
# Beispiel

Wir brauchen kein zweites Array:

3	17	-2	9	0	1	7	42	5
---	----	----	---	---	---	---	----	---

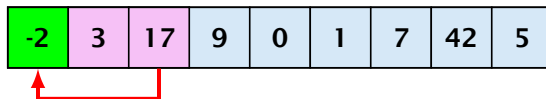
# Beispiel

Wir brauchen kein zweites Array:



# Beispiel

Wir brauchen kein zweites Array:





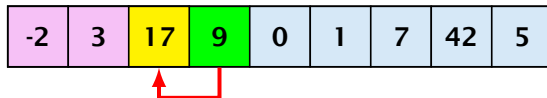
# Beispiel

Wir brauchen kein zweites Array:

-2	3	17	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

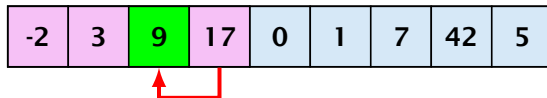
# Beispiel

Wir brauchen kein zweites Array:



# Beispiel

Wir brauchen kein zweites Array:



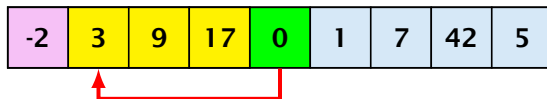
# Beispiel

Wir brauchen kein zweites Array:

-2	3	9	17	0	1	7	42	5
----	---	---	----	---	---	---	----	---

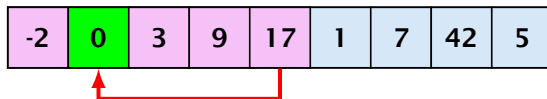
# Beispiel

Wir brauchen kein zweites Array:



# Beispiel

Wir brauchen kein zweites Array:



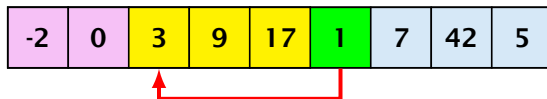
# Beispiel

Wir brauchen kein zweites Array:

-2	0	3	9	17	1	7	42	5
----	---	---	---	----	---	---	----	---

# Beispiel

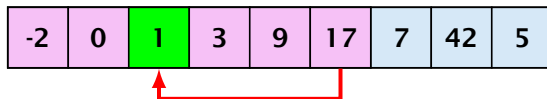
Wir brauchen kein zweites Array:





# Beispiel

Wir brauchen kein zweites Array:



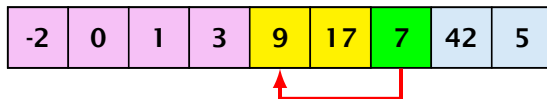
# Beispiel

Wir brauchen kein zweites Array:

-2	0	1	3	9	17	7	42	5
----	---	---	---	---	----	---	----	---

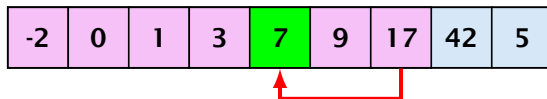
# Beispiel

Wir brauchen kein zweites Array:



# Beispiel

Wir brauchen kein zweites Array:



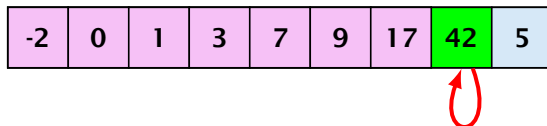
# Beispiel

Wir brauchen kein zweites Array:

-2	0	1	3	7	9	17	42	5
----	---	---	---	---	---	----	----	---

# Beispiel

Wir brauchen kein zweites Array:



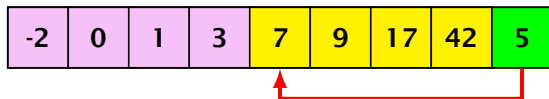
# Beispiel

Wir brauchen kein zweites Array:

-2	0	1	3	7	9	17	42	5
----	---	---	---	---	---	----	----	---

# Beispiel

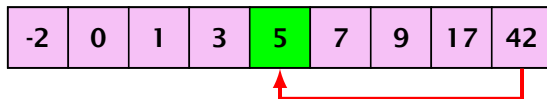
Wir brauchen kein zweites Array:





# Beispiel

Wir brauchen kein zweites Array:



# Beispiel

Wir brauchen kein zweites Array:

-2	0	1	3	5	7	9	17	42
----	---	---	---	---	---	---	----	----

# Algorithmus: Insertion Sort

```
1 Input: Array A mit n Elementen
2 Output: Array A aufsteigend sortiert
3
4 InsertionSort(A)
5     j = 0;
6     while (j < n)
7         key = A[j];
8         i = j-1;
9         while (i >= 0 && A[i] > key)
10             A[i+1] = A[i];
11             i--;
12         A[i+1] = key;
13         j++;
```

InsertionSort

# Laufzeit InsertionSort

Kostenübersicht		
Zeile	Kosten	Anzahl
5	$\mathcal{O}(1)$	1
6	$\mathcal{O}(1)$	$n + 1$
7	$\mathcal{O}(1)$	$n - 1$
8	$\mathcal{O}(1)$	$t_j$
9	$\mathcal{O}(1)$	$t_j - 1$
10	$\mathcal{O}(1)$	$t_j - 1$
11	$\mathcal{O}(1)$	$n$
12	$\mathcal{O}(1)$	$n$

$t_j$  bezeichnet Anzahl der Abfragen der while-Bedingung in Zeile 8 für Durchlauf  $j$

```
1 Input: Array A mit Laenge n
2 Output: sortiertes Array
3
4 InsertionSort(A)
5   j = 0;
6   while (j < n)
7       key = A[j];
8       i = j-1;
9       while (i >= 0 && A[i] > key)
10          A[i+1] = A[i];
11          i--;
12          A[i+1] = key;
13          j++;
```

# Laufzeit InsertionSort

$$\begin{aligned} & \mathcal{O}(1) + (n+1)\mathcal{O}(1) + (n-1)\mathcal{O}(1) + \mathcal{O}(1) \sum_{j=0}^{n-1} t_j + \\ & \mathcal{O}(1) \sum_{j=0}^{n-1} (t_j - 1) + \mathcal{O}(1) \sum_{j=0}^{n-1} (t_j - 1) + n\mathcal{O}(1) + n\mathcal{O}(1) \\ & = \mathcal{O}(1)n + \mathcal{O}(1) \sum_{j=0}^{n-1} t_j \\ & = \mathcal{O}\left(n + \sum_{j=0}^{n-1} t_j\right) \end{aligned}$$

Die Laufzeit hängt stark vom Input ab.

# Laufzeit InsertionSort

## Best-case:

Wenn das Array sortiert wird ist, ist  $t_j = 1$ .

⇒ Laufzeit:  $\mathcal{O}(n)$ .

## Worst-case:

Wenn das Array absteigend sortiert ist, ist  $t_j = j + 1$ .

⇒ Laufzeit:  $\mathcal{O}(n^2)$ .

## Beobachtung:

Wenn ein Element höchstens  $h$  Positionen von seiner Zielposition entfernt ist, dann ist  $t_j \leq h + 1$ .

⇒ Laufzeit:  $\mathcal{O}(hn)$ .

## 5 Effizienz von Algorithmen

**Gegeben:** Array  $A$  ganzer Zahlen; Element  $x$

**Gesucht:** Wo kommt  $x$  in  $A$  vor?

**Naives Vorgehen:**

- ▶ Vergleiche  $x$  der Reihe nach mit  $A[0]$ ,  $A[1]$ , usw.
- ▶ Finden wir  $i$  mit  $A[i] == x$ , geben wir  $i$  aus.
- ▶ Andernfalls geben wir  $-1$  aus: „Element nicht gefunden“!

# Naives Suchen

```
1 Input: Array A mit Laenge n; Element x
2 Output: i mit A[i] == x falls existent
3         sonst -1
4
5 find(A,x)
6     i = 0;
7     while (i < n && A[i] != x)
8         i++;
9     if (i == n)
10        return -1;
11    else
12        return i;
```

## Naives Suchen

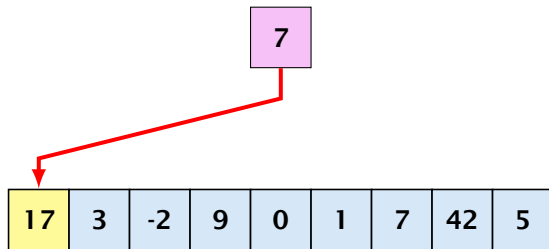


# Beispiel

7

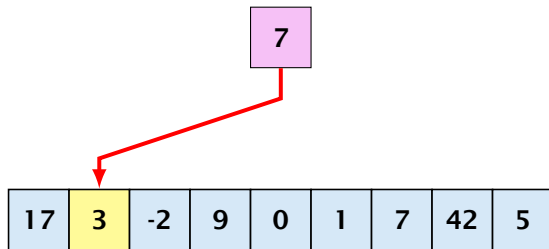
17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

# Beispiel



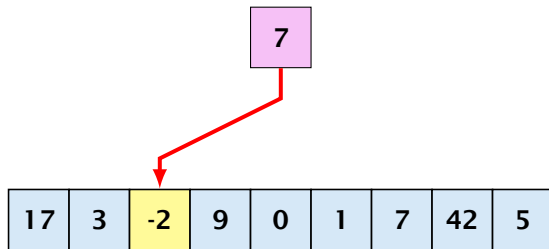
no

# Beispiel



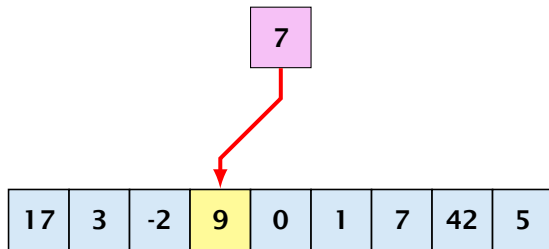
no

# Beispiel



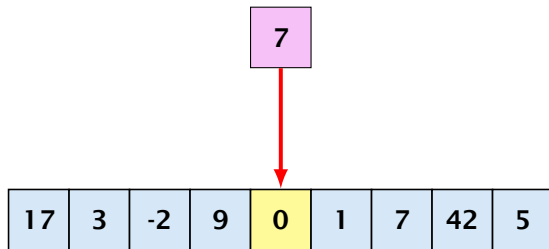
no

# Beispiel



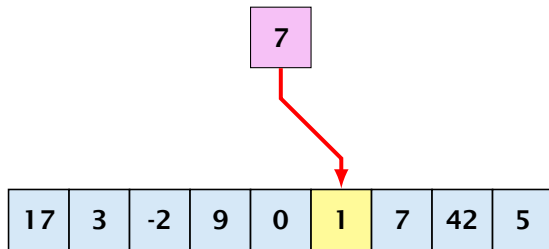
no

# Beispiel



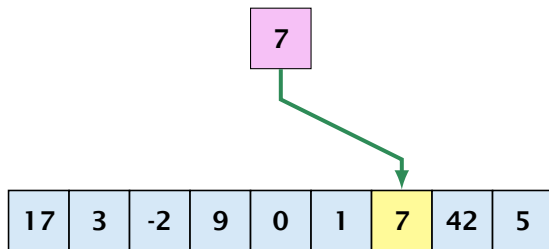
no

# Beispiel



no

# Beispiel



yes



# Laufzeit Naives Suchen

## Best-case:

Wenn  $x$  an Position 0.

⇒ Laufzeit:  $\mathcal{O}(1)$ .

## Worst-case:

Wenn  $x$  nicht vorkommt.

⇒ Laufzeit:  $\mathcal{O}(n)$ .

**...geht das besser?**

# Binäre Suche

**Annahme:** Input ist sortiert.

**Idee:**

- ▶ Vergleiche  $x$  mit dem Wert, der in der Mitte steht.
- ▶ Liegt Gleichheit vor, sind wir fertig.
- ▶ Ist  $x$  kleiner, brauchen wir nur noch links weitersuchen.
- ▶ Ist  $x$  größer, brauchen wir nur noch rechts weiter suchen.

⇒ binäre Suche

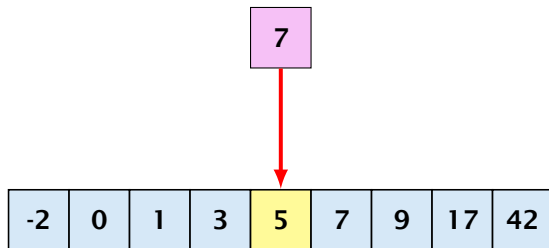
# Beispiel

7

-2	0	1	3	5	7	9	17	42
----	---	---	---	---	---	---	----	----

► wir benötigen nur **drei** Vergleiche

# Beispiel



no

► wir benötigen nur drei Vergleiche

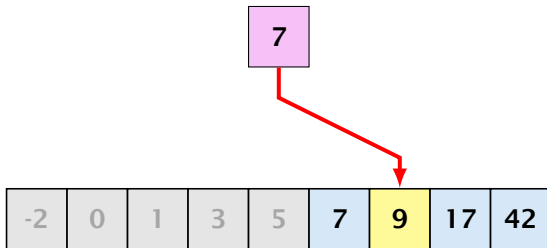
# Beispiel

7

-2	0	1	3	5	7	9	17	42
----	---	---	---	---	---	---	----	----

► wir benötigen nur **drei** Vergleiche

# Beispiel



no

► wir benötigen nur drei Vergleiche

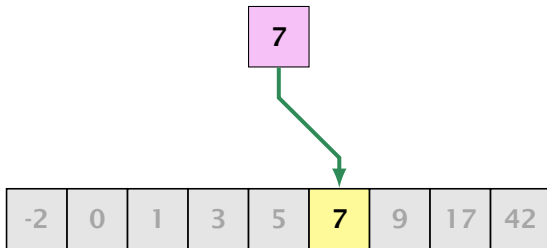
# Beispiel

7

-2	0	1	3	5	7	9	17	42
----	---	---	---	---	---	---	----	----

► wir benötigen nur **drei** Vergleiche

# Beispiel



yes

- ▶ wir benötigen nur **drei** Vergleiche



# Implementierung

```
1 Input: sortiertes Array A; Element x;
2 linker Index n1; rechter Index nr; n1<=nr
3 Output: Index i; n1 ≤ i ≤ nr mit A[i]==x falls existent
4 sonst -1
5 find(A, x, n1, nr) // Inputlänge ist n=nr-n1+1
6 t = (n1 + nr) / 2;
7 if (A[t] == x)
8     return t;
9 if (n1 == nr)
10    return -1;
11 if (x > A[t])
12    return find(A, x, t+1, nr); ← T(nr - t)
13 if (n1 < t)
14    return find(A, x, n1, t-1); ← T(n1 - t)
15 return -1;
```

# Laufzeit Binäre Suche



**Laufzeit:**


$$T(n) = \begin{cases} \mathcal{O}(1) & A[t] == x \\ \mathcal{O}(1) & n_l == n_r \\ \mathcal{O}(1) + T(nr - t) & x > A[t] \\ \mathcal{O}(1) + T(nl - t) & n_l < t \end{cases}$$

**oder**

$$T(n) \leq \begin{cases} \mathcal{O}(1) & n = 1 \\ \mathcal{O}(1) + T(\lfloor n/2 \rfloor) & \text{sonst} \end{cases}$$

# Laufzeit Binäre Suche

Lösen der Rekursionsgleichung:

$$T(n) \leq \begin{cases} \mathcal{O}(1) & n = 1 \\ \mathcal{O}(1) + T(\lfloor n/2 \rfloor) & \text{sonst} \end{cases}$$


Üblicherweise nur für  $n = 2^k$ ; z.B. durch vollständige Induktion.

Wie finden wir einen geschlossenen Ausdruck für die Laufzeit?

Dafür müssen wir die Rekursionsgleichung lösen.

Wie finden wir einen geschlossenen Ausdruck für die Laufzeit?

Dafür müssen wir die Rekursionsgleichung **lösen**.

## 1. Raten+Induktion

Man rät die richtige Lösung und beweist die Korrektheit mittels vollständiger Induktion. Man benötigt Erfahrung um richtig zu raten...

## 2. Mastertheorem

Für die meisten Rekurrenzen gibt es ein allgemeines Theorem, dass die asymptotisch korrekte Laufzeit für die jeweilige Rekurrenz angibt.

~~$O(n)$~~   
 $O(1)$

Zuerst müssen wir die  $O$ -Notation entfernen:

$$T(n) \leq \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

$\swarrow$   
 $O(1)$

# Raten+Induktion

Zuerst müssen wir die  $O$ -Notation entfernen:

$$T(n) \leq \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + c_1 & n > 1 \\ O(1) & \text{sonst} \end{cases}$$

$$\boxed{n \cdot c_1 + c_2}$$

||

$c_2$   
||

$$T(1) + k \cdot c_1$$

||

⋮

$$T(2^{k-i}) + i \cdot c_1$$

für  $n = 2^k$ :

Für diesen Fall können wir stattdessen die folgende Rekursionsgleichung betrachten:

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

$$T(2^n) = [T(2^{n-1})] + c_1$$

$$= [T(2^{n-2}) + c_1] + c_1 = \dots =$$



# Raten+Induktion

Zuerst müssen wir die  $\mathcal{O}$ -Notation entfernen:

$$T(n) \leq \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + c_1 n & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**für  $n = 2^k$ :**

Für diesen Fall können wir stattdessen die folgende Rekursionsgleichung betrachten:

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 n & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Man rät die richtige Lösung und beweist, dass durch Einsetzen, dass diese Lösung korrekt ist.

## Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

## Raten+Induktion

Ansatz:  $T(n) \leq a \log_2 n + b$ .

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

## Raten+Induktion

**Ansatz:**  $T(n) \leq a \log n + b$ .

**Beweis.** (durch Induktion)

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

## Raten+Induktion

**Ansatz:**  $T(n) \leq a \log n + b$ .

**Beweis.** (durch Induktion)

▶ **Anfang** ( $n = 1$ ):

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:**  $T(n) \leq a \log n + b$ .

**Beweis.** (durch Induktion)

► **Anfang** ( $n = 1$ ): **wahr** falls  $b \geq c_2$ .

$$c_2 = T(1) \leq \cancel{0} + b$$

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:**  $T(n) \leq a \log n + b$ .

**Beweis.** (durch Induktion)

- ▶ **Anfang** ( $n = 1$ ): **wahr** falls  $b \geq c_2$ .
- ▶ **Induktionsschritt**  $1, \dots, n-1 \rightarrow n$ :

$$\frac{n}{2} \rightarrow k$$

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:**  $T(n) \leq a \log n + b$ .

**Beweis.** (durch Induktion)

- ▶ **Anfang** ( $n = 1$ ): **wahr** falls  $b \geq c_2$ .
- ▶ **Induktionsschritt**  $1, \dots, n - 1 \rightarrow n$ :

Angenommen Aussage wahr für  $n' \in \{1, \dots, n - 1\}$ , und  $n > 1$ . Wir beweisen sie für  $n$ :



# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:**  $T(n) \leq a \log n + b$ .

**Beweis.** (durch Induktion)

- ▶ **Anfang** ( $n = 1$ ): **wahr** falls  $b \geq c_2$ .
- ▶ **Induktionsschritt**  $1, \dots, n - 1 \rightarrow n$ :

Angenommen Aussage wahr für  $n' \in \{1, \dots, n - 1\}$ , und  $n > 1$ . Wir beweisen sie für  $n$ :

$$T(n) \leq T\left(\frac{n}{2}\right) + c_1$$

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:**  $T(n) \leq a \log n + b$ .

**Beweis.** (durch Induktion)

- ▶ **Anfang** ( $n = 1$ ): **wahr** falls  $b \geq c_2$ .
- ▶ **Induktionsschritt**  $1, \dots, n-1 \rightarrow n$ :

Angenommen Aussage wahr für  $n' \in \{1, \dots, n-1\}$ , und  $n > 1$ . Wir beweisen sie für  $n$ :

$$\begin{aligned} T(n) &\leq \underbrace{T\left(\frac{n}{2}\right)}_{\downarrow} + c_1 \\ &\leq \left(a \log \frac{n}{2} + b\right) + c_1 \end{aligned}$$

$$n' = \frac{n}{2}$$

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:**  $T(n) \leq a \log n + b$ .

**Beweis.** (durch Induktion)

- ▶ **Anfang** ( $n = 1$ ): **wahr** falls  $b \geq c_2$ .
- ▶ **Induktionsschritt**  $1, \dots, n-1 \rightarrow n$ :

Angenommen Aussage wahr für  $n' \in \{1, \dots, n-1\}$ , und  $n > 1$ . Wir beweisen sie für  $n$ :

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + c_1 \\ &\leq \left(a \log \frac{n}{2} + b\right) + c_1 \\ &= a(\log n - 1) + b + c_1 \end{aligned}$$

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:**  $T(n) \leq a \log n + b$ .

**Beweis.** (durch Induktion)

- ▶ **Anfang** ( $n = 1$ ): **wahr** falls  $b \geq c_2$ .
- ▶ **Induktionsschritt**  $1, \dots, n-1 \rightarrow n$ :

Angenommen Aussage wahr für  $n' \in \{1, \dots, n-1\}$ , und  $n > 1$ . Wir beweisen sie für  $n$ :

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + c_1 \\ &\leq \left(a \log \frac{n}{2} + b\right) + c_1 \\ &= a(\log n - 1) + b + c_1 \\ &= a \log n + \underbrace{(c_1 - a)}_{\in 0} + b \end{aligned}$$

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:**  $T(n) \leq a \log n + b$ .

**Beweis.** (durch Induktion)

- ▶ **Anfang** ( $n = 1$ ): **wahr** falls  $b \geq c_2$ .
- ▶ **Induktionsschritt**  $1, \dots, n-1 \rightarrow n$ :

Angenommen Aussage wahr für  $n' \in \{1, \dots, n-1\}$ , und  $n > 1$ . Wir beweisen sie für  $n$ :

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + c_1 \\ &\leq \left(a \log \frac{n}{2} + b\right) + c_1 \\ &= a(\log n - 1) + b + c_1 \\ &= a \log n + (c_1 - a) + b \\ &\leq a \log n + b \end{aligned}$$

# Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

**Ansatz:**  $T(n) \leq a \log n + b$ .

**Beweis.** (durch Induktion)

► **Anfang** ( $n = 1$ ): **wahr** falls  $b \geq c_2$ .

► **Induktionsschritt**  $1, \dots, n-1 \rightarrow n$ :

$$f \in \mathcal{O}(1)$$

$$f(n) \leq c_1$$

Angenommen Aussage wahr für  $n' \in \{1, \dots, n-1\}$ , und  $n > 1$ . Wir beweisen sie für  $n$ :

$$T(n) \leq T\left(\frac{n}{2}\right) + c_1$$

$$n = 2^h$$

$$\leq \left(a \log \frac{n}{2} + b\right) + c_1 \quad \Leftrightarrow \quad \kappa = \log n$$

$$= a(\log n - 1) + b + c_1$$

$$= a \log n + (c_1 - a) + b$$

$$\leq a \log n + b$$

$\begin{matrix} \parallel & \parallel & \parallel \\ c_1 & \kappa & c_2 \end{matrix}$

Gilt falls  $a \geq c_1$ .

# Mastertheorem

Binäre Suche  $a=1$   $b=2$

$$\Theta(g(n)) = \mathcal{O}(g(n)) \cap \Omega(g(n))$$

$$n \begin{matrix} \nearrow \frac{n}{\log n} \\ \Rightarrow \vdots \\ \Rightarrow \vdots \end{matrix}$$

## Lemma 3

Seien  $a \geq 1, b \geq 1$  und  $\epsilon > 0$  **Konstanten**. Betrachte die Rekurrenz

$$T(n) = a T\left(\frac{n}{b}\right) + f(n).$$

$$\gamma = f(n) = \mathcal{O}\left(\frac{1}{n^\epsilon}\right)$$

1. Fall:

Falls  $f(n) = \mathcal{O}\left(\frac{1}{n^\epsilon}\right)$  gilt  $T(n) = \Theta(n^{\log_b a})$ .

2. Fall:

Falls  $f(n) = \Theta(n^{\log_b a} \log^k n)$  gilt  $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$ ,  
 $k \geq 0$ .

$$\rightarrow u=0$$

$$\gamma = \mathcal{O}(\log^n n)$$

# Mastertheorem

Wir beweisen das Mastertheorem für den Fall  $n = b^l$ , und nehmen an, dass der nichtrekursive Fall für Problemgröße  $1$  Kosten  $1$  verursacht.



# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:

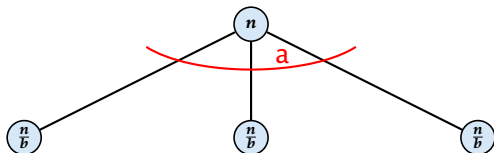
# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



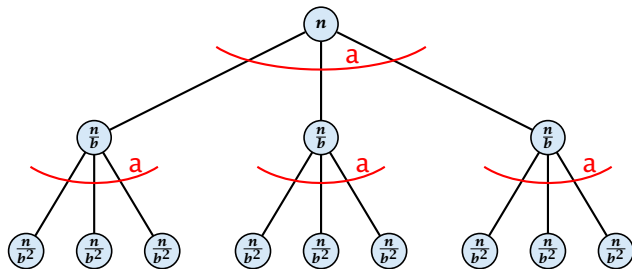
# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



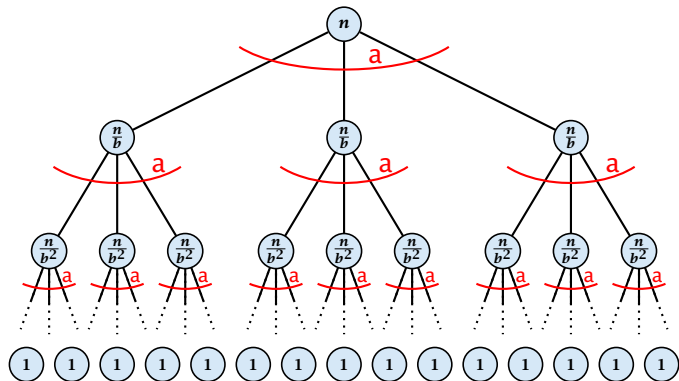
# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



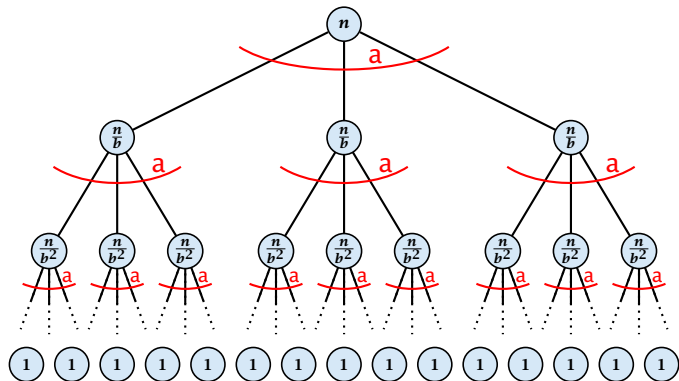
# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



# Der Rekursionsbaum

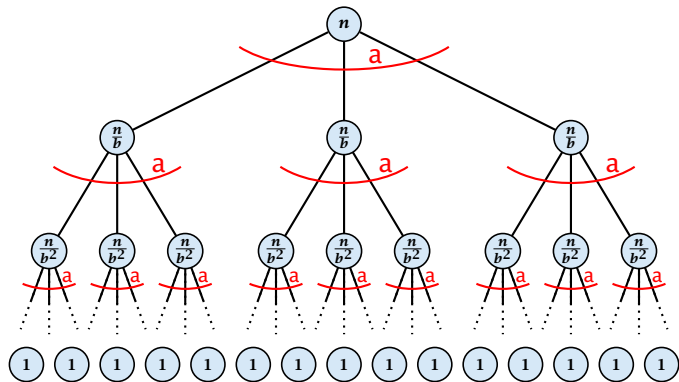
Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



$$f(n)$$

# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:

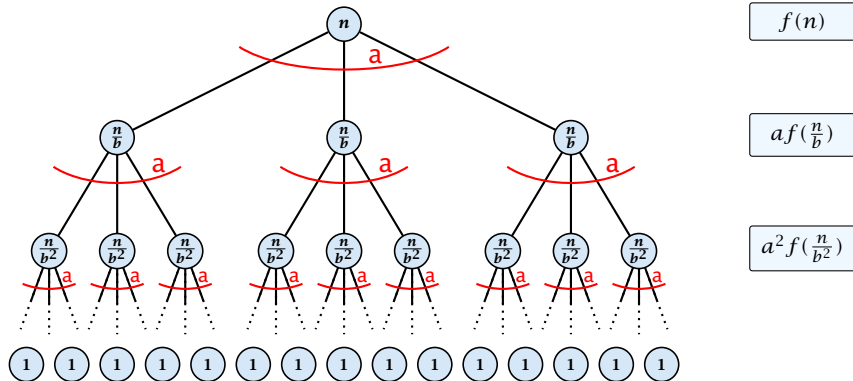


$$f(n)$$

$$af\left(\frac{n}{b}\right)$$

# Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:

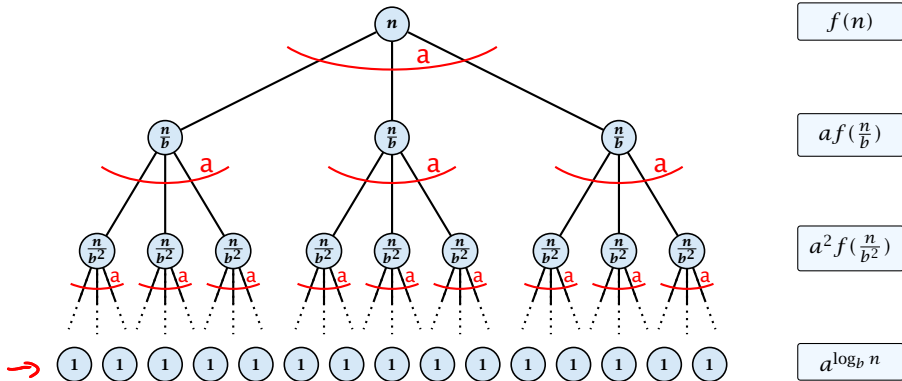




# Der Rekursionsbaum

$$f(1) = 1$$

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:

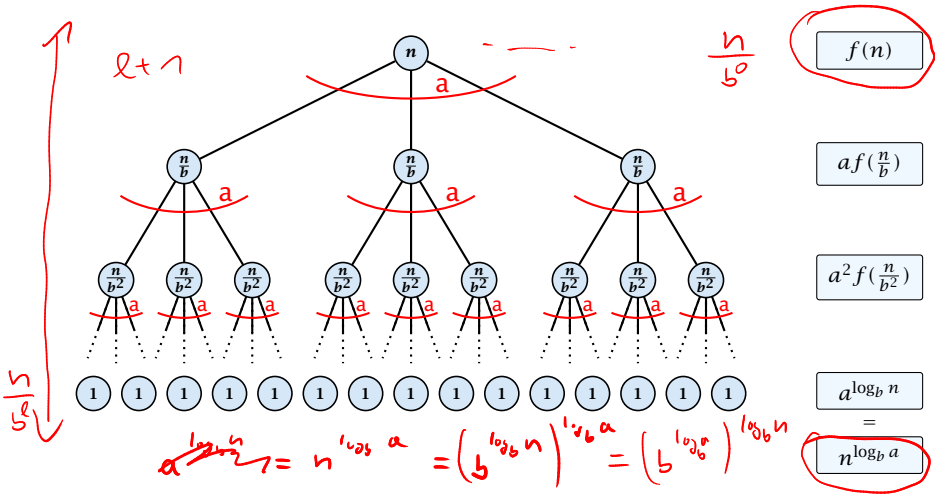


$$a^{\log_b n} = a^2 = a^2 \cdot f(1)$$

# Der Rekursionsbaum

$$\boxed{\Omega}(n^{\log_b a})$$

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



# Mastertheorem

Das heißt unsere Kosten sind

$$T(n) = n^{\log_b a} + \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) .$$

Fall 1. Sei  $f(n) \leq cn^{\log_b a - \epsilon}$ .

$$f(n) = \Theta(n^{\log_b a - \epsilon})$$

Fall 1. Sei  $f(n) \leq cn^{\log_b a - \epsilon}$ .

$$T(n) = n^{\log_b a}$$

Fall 1. Sei  $f(n) \leq cn^{\log_b a - \epsilon}$ .

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

Fall 1. Sei  $f(n) \leq cn^{\log_b a - \epsilon}$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \end{aligned}$$

Fall 1. Sei  $f(n) \leq cn^{\log_b a - \epsilon}$ .

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$-\epsilon \log_b a + \epsilon i$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon}$$

||

$$b^{-i(\log_b a - \epsilon)} = b^{\epsilon i} (b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}$$



Fall 1. Sei  $f(n) \leq cn^{\log_b a - \epsilon}$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \end{aligned}$$

$$\boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i} (b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}} = cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} (b^{\epsilon})^i$$

Fall 1. Sei  $f(n) \leq cn^{\log_b a - \epsilon}$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \end{aligned}$$

$$\boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i} (b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}} = cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} (b^{\epsilon})^i$$

$$\boxed{\sum_{i=0}^k q^i = \frac{q^{k+1} - 1}{q - 1}}$$

Fall 1. Sei  $f(n) \leq cn^{\log_b a - \epsilon}$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \end{aligned}$$

$$\boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i} (b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}} = cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} (b^{\epsilon})^i$$

$$\boxed{\sum_{i=0}^k q^i = \frac{q^{k+1} - 1}{q - 1}} = cn^{\log_b a - \epsilon} (b^{\epsilon \log_b n} - 1) / (b^{\epsilon} - 1)$$

Fall 1. Sei  $f(n) \leq cn^{\log_b a - \epsilon}$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \end{aligned}$$

$$\begin{aligned} b^{-i(\log_b a - \epsilon)} &= b^{\epsilon i} (b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i} \\ &= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} (b^\epsilon)^i \\ \sum_{i=0}^k q^i &= \frac{q^{k+1} - 1}{q - 1} \\ &= cn^{\log_b a - \epsilon} (b^{\epsilon \log_b n} - 1) / (b^\epsilon - 1) \\ &= cn^{\log_b a - \epsilon} (n^\epsilon - 1) / (b^\epsilon - 1) \end{aligned}$$

Fall 1. Sei  $f(n) \leq cn^{\log_b a - \epsilon}$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \end{aligned}$$

$$\begin{aligned} \boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i} (b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}} &= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} (b^\epsilon)^i \\ \boxed{\sum_{i=0}^k q^i = \frac{q^{k+1} - 1}{q - 1}} &= cn^{\log_b a - \epsilon} (b^{\epsilon \log_b n} - 1) / (b^\epsilon - 1) \\ &= cn^{\log_b a - \epsilon} (n^\epsilon - 1) / (b^\epsilon - 1) \\ &= \frac{c}{b^\epsilon - 1} n^{\log_b a} (n^\epsilon - 1) / (n^\epsilon) \end{aligned}$$

Fall 1. Sei  $f(n) \leq cn^{\log_b a - \epsilon}$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \end{aligned}$$

$$\begin{aligned} \boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i} (b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}} &= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} (b^\epsilon)^i \\ \boxed{\sum_{i=0}^k q^i = \frac{q^{k+1} - 1}{q - 1}} &= cn^{\log_b a - \epsilon} (b^{\epsilon \log_b n} - 1) / (b^\epsilon - 1) \\ &= cn^{\log_b a - \epsilon} (n^\epsilon - 1) / (b^\epsilon - 1) \\ &= \frac{c}{b^\epsilon - 1} n^{\log_b a} (n^\epsilon - 1) / (n^\epsilon) \end{aligned}$$

Also,

$$T(n) \leq \left( \frac{c}{b^\epsilon - 1} + 1 \right) n^{\log_b(a)}$$

Fall 1. Sei  $f(n) \leq cn^{\log_b a - \epsilon}$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \end{aligned}$$

$$\begin{aligned} \boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i} (b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}} &= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} (b^\epsilon)^i \\ \boxed{\sum_{i=0}^k q^i = \frac{q^{k+1} - 1}{q - 1}} &= cn^{\log_b a - \epsilon} (b^{\epsilon \log_b n} - 1) / (b^\epsilon - 1) \\ &= cn^{\log_b a - \epsilon} (n^\epsilon - 1) / (b^\epsilon - 1) \\ &= \frac{c}{b^\epsilon - 1} n^{\log_b a} (n^\epsilon - 1) / (n^\epsilon) \end{aligned}$$

Also,

$$T(n) \leq \left( \frac{c}{b^\epsilon - 1} + 1 \right) n^{\log_b(a)} \quad \Rightarrow T(n) = \mathcal{O}(n^{\log_b a}).$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a}$ .



Fall 2. Sei  $f(n) \leq cn^{\log_b a}$ .

$$T(n) = n^{\log_b a}$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a}$ .

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a}$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \end{aligned}$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a}$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\ &= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \end{aligned}$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a}$ .

$$\begin{aligned}T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\&= cn^{\log_b a} \log_b n\end{aligned}$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a}$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\ &= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\ &= cn^{\log_b a} \log_b n \end{aligned}$$

*Handwritten notes:*  $\left(\frac{1}{b^i}\right)^{\log_b a}$  and  $\frac{1}{a^i}$

Also,

$$T(n) = \mathcal{O}(n^{\log_b a} \log_b n)$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a}$ .

$$\begin{aligned}T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\ &= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\ &= cn^{\log_b a} \log_b n\end{aligned}$$

Also,

$$T(n) = \mathcal{O}(n^{\log_b a} \log_b n)$$

$$\Rightarrow T(n) = \mathcal{O}(n^{\log_b a} \log n).$$

Fall 2. Sei  $f(n) \geq cn^{\log_b a}$ .



Fall 2. Sei  $f(n) \geq cn^{\log_b a}$ .

$$T(n) - n^{\log_b a}$$

Fall 2. Sei  $f(n) \geq cn^{\log_b a}$ .

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

Fall 2. Sei  $f(n) \geq cn^{\log_b a}$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \end{aligned}$$

Fall 2. Sei  $f(n) \geq cn^{\log_b a}$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\ &= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \end{aligned}$$

Fall 2. Sei  $f(n) \geq cn^{\log_b a}$ .

$$\begin{aligned}T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\ &= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\ &= cn^{\log_b a} \log_b n\end{aligned}$$

Fall 2. Sei  $f(n) \geq cn^{\log_b a}$ .

$$\begin{aligned}T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\&\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\&= cn^{\log_b a} \log_b n\end{aligned}$$

Also,

$$T(n) = \underline{\underline{\Omega}}(n^{\log_b a} \log_b n)$$

Fall 2. Sei  $f(n) \geq cn^{\log_b a}$ .

$$\begin{aligned}T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\ &= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\ &= cn^{\log_b a} \log_b n\end{aligned}$$

Also,

$$T(n) = \Omega(n^{\log_b a} \log_b n)$$

$$\Rightarrow T(n) = \Omega(n^{\log_b a} \log n).$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a} (\log_b(n))^k$ .



Fall 2. Sei  $f(n) \leq cn^{\log_b a} (\log_b(n))^k$ .

$$T(n) = n^{\log_b a}$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a} (\log_b(n))^k$ .

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a} (\log_b(n))^k$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \end{aligned}$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a} (\log_b(n))^k$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \end{aligned}$$

$$n = b^\ell \Rightarrow \ell = \log_b n$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a} (\log_b(n))^k$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b \left(\frac{n}{b^i}\right)\right)^k \end{aligned}$$

$n = b^\ell \Rightarrow \ell = \log_b n$
--

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b \left(\frac{b^\ell}{b^i}\right)\right)^k$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a} (\log_b(n))^k$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \end{aligned}$$

$$n = b^\ell \Rightarrow \ell = \log_b n$$

$$\begin{aligned} &= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\ &= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k \end{aligned}$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a} (\log_b(n))^k$ .

$$1 + 2 + 3 + 4 + \dots = \frac{1}{2} n(n+1)$$

$$1^2 + 2^2 + 3^2 + 4^2 + \dots =$$

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b \left(\frac{n}{b^i}\right)\right)^k$$

$$n = b^\ell \Rightarrow \ell = \log_b n$$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b \left(\frac{b^\ell}{b^i}\right)\right)^k$$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k$$

$$= cn^{\log_b a} \sum_{i=1}^{\ell} i^k$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a} (\log_b(n))^k$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \end{aligned}$$

$$n = b^\ell \Rightarrow \ell = \log_b n$$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k$$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k$$

$$= cn^{\log_b a} \sum_{i=1}^{\ell} i^k \approx \frac{1}{k} \ell^{k+1}$$



Fall 2. Sei  $f(n) \leq cn^{\log_b a} (\log_b(n))^k$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \end{aligned}$$

$$n = b^\ell \Rightarrow \ell = \log_b n$$

$$\begin{aligned} &= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\ &= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k \\ &= cn^{\log_b a} \sum_{i=1}^{\ell} i^k \\ &\approx \frac{c}{k} n^{\log_b a} \ell^{k+1} \end{aligned}$$

Fall 2. Sei  $f(n) \leq cn^{\log_b a} (\log_b(n))^k$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \end{aligned}$$

$$n = b^\ell \Rightarrow \ell = \log_b n$$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k$$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k$$

$$= cn^{\log_b a} \sum_{i=1}^{\ell} i^k$$

$$\approx \frac{c}{k} n^{\log_b a} \ell^{k+1}$$

$$\Rightarrow T(n) = \mathcal{O}(n^{\log_b a} \log^{k+1} n).$$

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{r} 1\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ A \\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ B \\ \hline \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

1	1	0	1	1	0	1	0	1	$A$
1	0	0	0	1	0	0	1	1	$B$
<hr/>									

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{r} 1\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1 \\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1 \\ \hline \phantom{1\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1} 0 \end{array}$$

The diagram shows the addition of two 9-bit integers, A and B, to produce a 10-bit result. The numbers are aligned to the right. A horizontal line is drawn under the 9th bit of the second number. A carry bit '1' is shown below the 9th bit of the second number. The result '0' is shown in a box below the 9th bit of the second number. The labels A and B are to the right of the numbers.

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{r} 1\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ A \\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ B \\ \hline \phantom{1\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ } 0 \end{array}$$

The diagram illustrates the addition of two 10-bit integers, A and B. The bits of A are 1, 1, 0, 1, 1, 0, 1, 0, 1, 1. The bits of B are 1, 0, 0, 0, 1, 0, 0, 1, 1. A horizontal line is drawn under the 8th bit of B. A vertical box highlights the 8th bit of A (0) and the 8th bit of B (1), with a '1' written below the box. Below the horizontal line, a '0' is written under the 9th bit position.



## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{r} 1\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ A \\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ B \\ \hline 0\ 0 \end{array}$$

The diagram illustrates the addition of two 10-bit integers, A and B. The bits of A are 1, 1, 0, 1, 1, 0, 1, 0, 1, 1. The bits of B are 1, 0, 0, 0, 1, 0, 0, 1, 1. A horizontal line is drawn under the bits of B. The result of the addition is shown below the line as 0, 0. A vertical box highlights the 8th and 9th bits of the result, which are 0 and 0. Small '1' characters are placed below the 8th and 9th bits of the input numbers, indicating carry bits.

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{rcccccccc} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\ \hline & & & & & & 1 & 1 & & \\ & & & & & & & 0 & 0 & \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{rcccccccc} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\ \hline & & & & & & 0 & 0 & 0 & \end{array}$$

The diagram illustrates the addition of two 9-bit integers, A and B, to produce a 9-bit result. The numbers are aligned to the right. A horizontal line is drawn under the numbers. The result is shown below the line, with a vertical line separating the 9th bit from the carry bits. The 9th bit of the result is 0, and the carry bits are 0, 0, and 0. The 9th bit of the result is highlighted with a light blue box.

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{rcccccccc} & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\ & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\ \hline & & & & & & 1 & 1 & 1 & & \\ & & & & & & 0 & 0 & 0 & & \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{rcccccccc} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\ \hline & & & & & 1 & 0 & 0 & 0 & \end{array}$$

Carry bits: 0, 1, 1, 1

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{rcccccccc} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\ \hline & & & & 0 & 1 & 1 & 1 & & \\ & & & & 1 & 0 & 0 & 0 & & \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{rcccccccc} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\ \hline & & & & 1 & 0 & 1 & 1 & 1 & \\ & & & & 0 & 1 & 0 & 0 & 0 & \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{rcccccccc} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\ \hline & & & 1 & 0 & 1 & 1 & 1 & & \\ & & & 0 & 1 & 0 & 0 & 0 & & \end{array}$$



## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{rcccccccc} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\ \hline & & & 0 & 0 & 1 & 0 & 0 & 0 & \end{array}$$

The diagram illustrates the addition of two 9-bit integers, A and B, to produce a 9-bit result. The numbers are aligned by their least significant bits. A horizontal line is drawn under the second row. The result is shown below the line. A vertical box highlights the fourth bit position (index 3 from the right) in both the second row and the result row, indicating a carry-in of 1 from the third bit position. Small subscripts '1' are placed below the third bit of the second row and the third bit of the result row to indicate the carry-in. The result row shows a carry-out of 0 from the fourth bit position.

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{rcccccccc} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\ \hline & & & 1 & 1 & 0 & 1 & 1 & 1 & \\ & & & 0 & 0 & 1 & 0 & 0 & 0 & \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

1	1	0	1	1	0	1	0	1	$A$
1	0	0	0	1	0	0	1	1	$B$
	<sub>0</sub>	<sub>1</sub>	<sub>1</sub>	<sub>0</sub>	<sub>1</sub>	<sub>1</sub>	<sub>1</sub>		
<hr/>									
		1	0	0	1	0	0	0	

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{rcccccccc} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\ \hline & 0 & 1 & 1 & 0 & 1 & 1 & 1 & & \\ & 1 & 0 & 0 & 1 & 0 & 0 & 0 & & \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{rcccccccc} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\ \hline & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \end{array}$$

0 0 1 1 0 1 1 1

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

The diagram illustrates the addition of two 8-bit integers, A and B, to produce an 8-bit result. The numbers are aligned to the right, and a horizontal line is drawn below the second row. The first bit of the first row is enclosed in a light blue box. Below the horizontal line, the bits of the result are shown. Small subscripts are placed below the bits of the second row to indicate carry bits.

1	1	0	1	1	0	1	0	1	$A$
1	0	0	0	1	0	0	1	1	$B$
<hr/>									
	1	1	0	0	1	0	0	0	

Carry bits (subscripts): 0, 0, 1, 1, 0, 1, 1, 1

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{r} \boxed{1} \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \quad A \\ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \quad B \\ \hline 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

	1	1	0	1	1	0	1	0	1	$A$
	1	0	0	0	1	0	0	1	1	$B$
	<hr/>									
	0	1	1	0	0	1	0	0	0	



## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

		1	1	0	1	1	0	1	0	1	$A$
		1	0	0	0	1	0	0	1	1	$B$
		<hr/>									
	1	0	1	1	0	0	1	0	0	0	

## Beispiel: Integermultiplikation

Angenommen wir möchten zwei  $n$ -bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integer konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen  $A$  and  $B$  addieren können:

$$\begin{array}{r} 1\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ A \\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ B \\ \hline 1\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 0\ 0 \end{array}$$

Das heißt wir können zwei  $n$ -bit Integer in Zeit  $\mathcal{O}(n)$  addieren.

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 10001 \times 1011 \\ \hline \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \\ \times 1\ 0\ 1\ 1 \\ \hline \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1 \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline 1\ 0\ 0\ 0\ 1 \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline 1\ 0\ 0\ 0\ 1 \\ \phantom{1\ 0\ 0\ 0\ 1} 0 \\ \phantom{1\ 0\ 0\ 0\ 1} 0 \\ \phantom{1\ 0\ 0\ 0\ 1} 0 \end{array}$$



## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0 \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0 \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0 \\ \phantom{1\ 0\ 0\ 0\ 1} \phantom{1\ 0\ 0\ 0\ 1} 0\ 0 \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0 \\ \phantom{1\ 0\ 0\ 0\ 1} 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0 \\ \phantom{1\ 0\ 0\ 0\ 1} 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0 \\ \phantom{1\ 0\ 0\ 0\ 1} 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ \phantom{1\ 0\ 0\ 0\ 1} \phantom{0\ 0\ 0\ 0\ 0} 0\ 0\ 0 \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0 \\ \phantom{1\ 0\ 0\ 0\ 1} 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0 \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0 \\ \phantom{1\ 0\ 0\ 0\ 1} 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0 \\ \hline \end{array}$$



## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0 \\ \phantom{1\ 0\ 0\ 0\ 1} 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0 \\ \hline 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1 \end{array}$$

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0 \\ \phantom{1\ 0\ 0\ 0\ 1} 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0 \\ \hline 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1 \end{array}$$

**Laufzeit:**

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

*m Zahlen* *m+n bits*

$$\begin{array}{r} 10001 \times 1011 \\ \hline 10001 \\ 100010 \\ 0000000 \\ 10001000 \\ \hline 10111011 \end{array}$$

*m*

*n*

*(m-1)*

**Laufzeit:**

- ▶ Zwischenergebnisse berechnen:  $\mathcal{O}(nm)$ .

## Beispiel: Integermultiplikation

Angenommen wir möchten ein  $n$ -bit Integer  $A$  und ein  $m$ -bit Integer  $B$  ( $m \leq n$ ) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0 \\ \phantom{1\ 0\ 0\ 0\ 1} 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ \phantom{1\ 0\ 0\ 0\ 1} 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0 \\ \hline 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1 \end{array}$$

### Laufzeit:

- ▶ Zwischenergebnisse berechnen:  $\mathcal{O}(nm)$ .
- ▶ Addieren von  $m$  Zahlen der Länge  $\leq 2n$ :  
 $\mathcal{O}((m+n)m) = \mathcal{O}(nm)$ .

# Beispiel: Integermultiplikation

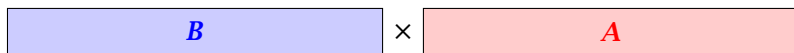
**Ein rekursiver Ansatz:**

Angenommen die Integer  $A$  und  $B$  haben Länge  $n = 2^k$ .

# Beispiel: Integermultiplikation

Ein rekursiver Ansatz:

Angenommen die Integer  $A$  und  $B$  haben Länge  $n = 2^k$ .



# Beispiel: Integermultiplikation

**Ein rekursiver Ansatz:**

Angenommen die Integer  $A$  und  $B$  haben Länge  $n = 2^k$ .

$$\boxed{b_{n-1} \quad \dots \quad b_0} \times \boxed{a_{n-1} \quad \dots \quad a_0}$$

# Beispiel: Integermultiplikation

Ein rekursiver Ansatz:

Angenommen die Integer  $A$  und  $B$  haben Länge  $n = 2^k$ .

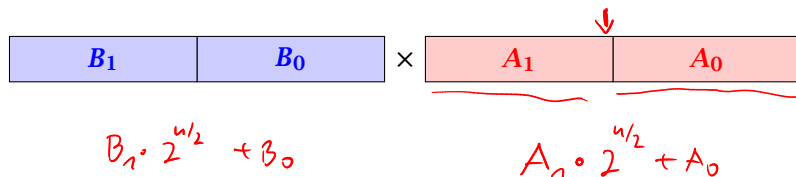
$$\boxed{b_{n-1} \quad \dots \quad b_{\frac{n}{2}} \quad b_{\frac{n}{2}-1} \quad \dots \quad b_0} \times \boxed{a_{n-1} \quad \dots \quad a_{\frac{n}{2}} \quad a_{\frac{n}{2}-1} \quad \dots \quad a_0}$$



# Beispiel: Integermultiplikation

Ein rekursiver Ansatz:

Angenommen die Integer  $A$  und  $B$  haben Länge  $n = 2^k$ .



# Beispiel: Integermultiplikation

Ein rekursiver Ansatz:

Angenommen die Integer  $A$  und  $B$  haben Länge  $n = 2^k$ .



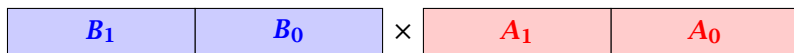
Dann gilt

$$A = A_1 \cdot 2^{\frac{n}{2}} + A_0 \text{ und } B = B_1 \cdot 2^{\frac{n}{2}} + B_0$$

# Beispiel: Integermultiplikation

Ein rekursiver Ansatz:

Angenommen die Integer  $A$  und  $B$  haben Länge  $n = 2^k$ .



Dann gilt

$$A = A_1 \cdot 2^{\frac{n}{2}} + A_0 \text{ und } B = B_1 \cdot 2^{\frac{n}{2}} + B_0$$

Also,

$$A \cdot B = A_1 B_1 \cdot 2^n + (A_1 B_0 + A_0 B_1) \cdot 2^{\frac{n}{2}} + A_0 B_0$$

## Beispiel: Integermultiplikation

```
1 Input: Zahlen A, B, repräsentiert durch Bitarrays
2       der Länge n
3 Output: Bitarray, das A*B enthält
4
5 mult(A, B, n)
6     if (n == 1)
7         return new int(A[0]*B[0]);
8     O(n) split(A, A0, A1);
9     O(n) split(B, B0, B1);
10    T(n/2) Z2 = mult(A1, B1, n/2);
11    O(n) + T(n/2) Z1 = mult(A0, B1, n/2) + mult(A1, B0, n/2);
12    T(n/2) Z0 = mult(A0, B0, n/2);
13    O(n) return Z2*2^n + Z1*2^{n/2} + Z0;
```

Wir erhalten folgende Rekurrenzgleichung:

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n)$$

# Beispiel: Integermultiplikation

$$a=4 \quad f(n)=n$$
$$b=2$$

**Mastertheorem:** Rekurrenz:  $T[n] = aT(\frac{n}{b}) + f(n)$ .

► Fall 1:  $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$

$$T(n) = \Theta(n^{\log_b a})$$

► Fall 2:  $f(n) = \Theta(n^{\log_b a} \log^k n)$   $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

# Beispiel: Integermultiplikation

**Mastertheorem:** Rekurrenz:  $T[n] = aT(\frac{n}{b}) + f(n)$ .

- ▶ Fall 1:  $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$        $T(n) = \Theta(n^{\log_b a})$
- ▶ Fall 2:  $f(n) = \Theta(n^{\log_b a} \log^k n)$        $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

In unserem Fall  $a = 4$ ,  $b = 2$ , und  $f(n) = \Theta(n)$ . Also, Fall 1, da  $n = \mathcal{O}(n^{2-\epsilon}) = \mathcal{O}(n^{\log_b a - \epsilon})$ .

# Beispiel: Integermultiplikation

**Mastertheorem:** Rekurrenz:  $T[n] = aT(\frac{n}{b}) + f(n)$ .

- ▶ Fall 1:  $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$        $T(n) = \Theta(n^{\log_b a})$
- ▶ Fall 2:  $f(n) = \Theta(n^{\log_b a} \log^k n)$        $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

In unserem Fall  $a = 4$ ,  $b = 2$ , und  $f(n) = \Theta(n)$ . Also, Fall 1, da  $n = \mathcal{O}(n^{2-\epsilon}) = \mathcal{O}(n^{\log_b a - \epsilon})$ .

Wir erhalten Laufzeit  $\mathcal{O}(n^2)$ .

# Beispiel: Integermultiplikation

**Mastertheorem:** Rekurrenz:  $T[n] = aT(\frac{n}{b}) + f(n)$ .

► Fall 1:  $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$

$$T(n) = \Theta(n^{\log_b a})$$

► Fall 2:  $f(n) = \Theta(n^{\log_b a} \log^k n)$      $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

In unserem Fall  $a = 4$ ,  $b = 2$ , und  $f(n) = \Theta(n)$ . Also, Fall 1, da  $n = \mathcal{O}(n^{2-\epsilon}) = \mathcal{O}(n^{\log_b a - \epsilon})$ .

Wir erhalten Laufzeit  $\mathcal{O}(n^2)$ .

⇒ Nicht besser als „Schulmethode“.



## Beispiel: Integermultiplikation

We can use the following identity to compute  $Z_1$ :

## Beispiel: Integermultiplikation

We can use the following identity to compute  $Z_1$ :

$$Z_1 = A_1B_0 + A_0B_1$$

## Beispiel: Integermultiplikation

We can use the following identity to compute  $Z_1$ :

$$\begin{aligned}Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - A_1B_1 - A_0B_0\end{aligned}$$

## Beispiel: Integermultiplikation

We can use the following identity to compute  $Z_1$ :

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 && = Z_2 && = Z_0 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - \underbrace{A_1B_1}_{Z_2} - \underbrace{A_0B_0}_{Z_0} \end{aligned}$$

## Beispiel: Integermultiplikation

We can use the following identity to compute  $Z_1$ :

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 && = Z_2 && = Z_0 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - \underbrace{A_1B_1} && - \underbrace{A_0B_0} \end{aligned}$$

```
1 Input: Zahlen A, B, repraesentiert durch Bitarrays
2       der Laenge n
3 Output: Bitarray, das A*B enthaelt
4
5 mult(A, B, n)
6     if (n == 1)
7         return new int(A[0]*B[0]);
8     split(A,A0,A1);
9     split(B,B0,B1);
10    Z2 = mult(A1,B1,n/2);
11    Z1 = mult(A0+A1,B0+B1,n/2)-Z0-Z2;
12    Z0 = mult(A0,B0,n/2);
13    return Z2*2^n + Z1*2^{n/2} + Z0;
```

## Beispiel: Integermultiplikation

Zeile 11 ist leider nicht korrekt, da  $A0 + A1$  bzw.  $B0 + B1$  eventuell  $n/2 + 1$  bits haben können. Wenn man eine  $n/2 + 1$ -bit Zahl  $X$  in das höchstwertige Bit  $X_{\frac{n}{2}}$  und die restlichen Bits  $\tilde{X}$  zerlegt kann man folgendes nutzen:

$$\begin{aligned} & \dots \\ X &= A0 + A1; \\ Y &= B0 + B1; \\ Z1 &= X_{\frac{n}{2}} * Y_{\frac{n}{2}} * 2^n + (X_{\frac{n}{2}} * \tilde{Y} + Y_{\frac{n}{2}} * \tilde{X}) * 2^{n/2} + \text{mult}(\tilde{X}, \tilde{Y}) - Z0 - Z2; \\ & \dots \end{aligned}$$

Laufzeit hierfür ist  $T(n/2) + \mathcal{O}(n)$ .

# Beispiel: Integermultiplikation

Wir erhalten folgende Rekurrenz:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) .$$

Master Theorem: Rekurrenz:  $T[n] = aT\left(\frac{n}{b}\right) + f(n)$ .

- ▶ Case 1:  $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$        $T(n) = \Theta(n^{\log_b a})$
- ▶ Case 2:  $f(n) = \Theta(n^{\log_b a} \log^k n)$        $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Wir sind im Fall 1. Laufzeit  $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$ .

Eine deutliche Verbesserung der „Schulmethode“.

# Beispiel: Integermultiplikation

Wir erhalten folgende Rekurrenz:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) .$$

**Master Theorem:** Rekurrenz:  $T[n] = aT\left(\frac{n}{b}\right) + f(n)$ .

- ▶ Case 1:  $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$        $T(n) = \Theta(n^{\log_b a})$
- ▶ Case 2:  $f(n) = \Theta(n^{\log_b a} \log^k n)$        $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Wir sind im Fall 1. Laufzeit  $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$ .

Eine deutliche Verbesserung der „Schulmethode“.



# Beispiel: Integermultiplikation

Wir erhalten folgende Rekurrenz:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) .$$

**Master Theorem:** Rekurrenz:  $T[n] = aT\left(\frac{n}{b}\right) + f(n)$ .

- ▶ Case 1:  $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$        $T(n) = \Theta(n^{\log_b a})$
- ▶ Case 2:  $f(n) = \Theta(n^{\log_b a} \log^k n)$        $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Wir sind im Fall 1. Laufzeit  $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$ .

Eine deutliche Verbesserung der „Schulmethode“.

## Beispiel: Integermultiplikation

Wir erhalten folgende Rekurrenz:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) .$$

**Master Theorem:** Rekurrenz:  $T[n] = aT\left(\frac{n}{b}\right) + f(n)$ .

- ▶ Case 1:  $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$        $T(n) = \Theta(n^{\log_b a})$
- ▶ Case 2:  $f(n) = \Theta(n^{\log_b a} \log^k n)$        $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Wir sind im Fall 1. Laufzeit  $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$ .

Eine deutliche Verbesserung der „Schulmethode“.

# Algorithmenentwurf

Kein Patentrezept zum Entwurf von Algorithmen!

- ▶ insbesondere Ableitung von Algorithmus aus Spezifikation nicht automatisierbar

Programmieren ist **creative** Tätigkeit

- ▶ “The Art of Computer Programming” (D. Knuth)

Unterstützung durch **Algorithmen-Muster**

- ▶ auch **Design Patterns** genannt
- ▶ “best practice”

# Divide and Conquer

## Definition: Divide and Conquer

Divide and Conquer ist die **rekursive** Rückführung eines zu lösenden Problems auf mehrere **identische** Problem mit **kleinerer** Eingabemenge.

**Divide and Conquer:** zu deutsch “Teile und herrsche”

## Prinzip:

- ▶ teile große Aufgabe in mehrere kleine Teilaufgaben
- ▶ rufe denselben Algorithmus rekursiv auf den Teilaufgaben auf

# Divide and Conquer

1. **Teile** gegebene Aufgabe in mehrere getrennte Teilaufgaben
  - ▶ **löse** Teilaufgaben einzeln
  - ▶ **setze** Lösung der Gesamtaufgabe aus Teillösungen zusammen
2. Wende dieselbe Technik auf jede Teilaufgabe an, dann auf deren Teilaufgaben etc., bis die Teilaufgabe so klein ist, dass Lösung explizit berechnet werden kann
3. Jede Teilaufgabe sollte **von derselben Art** sein wie die Gesamtaufgabe, so dass der gleiche Algorithmus rekursiv aufgerufen werden kann

# Divide and Conquer

```
1 Input: Aufgabe A
2
3 DivideAndConquer(A)
4   if (A klein)
5     loese A explizit;
6   else
7     teile A in Teilaufgaben  $A_1, \dots, A_n$ ;
8     DivideAndConquer( $A_1$ )
9     ...
10    DivideAndConquer( $A_n$ )
11    berechne Loesung fuer A aus Lsgn fuer  $A_1, \dots, A_n$ 
```

# Divide and Conquer

- ▶ Berechnung der **Fibonacci**-Zahlen (untypisch, da Teilprobleme Größe  $n-1$  und  $n-2$  haben).
- ▶ Binäre Suche (nur ein Teilproblem der Größe  $\leq n/2$ ).
- ▶ Karatsuba für die Multiplikation großer Zahlen.
- ▶ **MergeSort**
- ▶ **QuickSort**
- ▶ **Fast Fourier Transformation (FFT)**
- ▶ **Medianberechnung**
- ▶ ...

# Divide and Conquer: MergeSort

Sei  $L$  verkettete Liste mit  $n$  natürlichen Zahlen  $a_i \in \mathbb{N}$ .

**Aufgabe:** sortiere  $L$  in aufsteigender Reihenfolge.

Lösung mit Divide and Conquer-Muster: **MergeSort**

Idee:

- ▶ **Divide:** teile  $L$  auf in zwei gleich große Teillisten
- ▶ **Rekursion:** rufe MergeSort rekursiv für die zwei Teillisten auf
- ▶ **Conquer:** setze die Teillisten zusammen (**merge** bzw. mischen)

Wann ist Teilliste **“klein”**, d.h. wann löst man explizit?

→ Teilliste mit nur **einem** Element → sortiert!



# Divide and Conquer: MergeSort

```
1 Input: Liste (ueber this pointer)
2 Output: Liste ist sortiert
3
4 MergeSort()
5     if (head == null || head->next == null)
6         return;
7     List L = l.half();
8     MergeSort();
9     L.MergeSort();
10    merge(L);
```

## Divide and Conquer: MergeSort

```
1 merge(L)
2     Node* a = head;
3     Node* b = L.head;
4     Node* dummy = new Node(-\infty, NULL);
5     Node* r = dummy;
6
7     while (1)
8         if (a == null)
9             r->next = b; break;
10        if (b == null)
11            r->next = a; break;
12        if (a.data < b.data)
13            r->next = a;
14            r = r->next;
15        else
16            r->next = b;
17            r = r->next;
18    head = dummy->next;
19    delete dummy;
```

# MergeSort: Algorithmus

```
1 Input: Liste mit mindestens 2 Elemente (ueber  
2   this-pointer)  
3 Output: Die Liste, fuer die die Funktion aufgerufen wird  
4   enthaelt die erste Haelfte der Elemente; eine  
5   Liste mit den restlichen Elementen wird zurueck-  
6   gegeben  
7  
8 List half()  
9   int n = length();  
10  Node* t = head;  
11  for (int i=0; i<n/2-1; i++)  
12    t = t.next;  
13  List* result = new List(t.next);  
14  t.next = null;  
15  return result;
```