SS 2019

Algorithmen und Datenstrukturen

Harald Räcke

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 Hünermann

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)

Informationen online

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 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?

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 $f : E \to A$, mit $E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, 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 $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, 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 $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, 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 $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, 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 $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, 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

Algorithmus

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

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

Resultat	***Algorithmus***	***Einzelschritte***
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 $\mathrm{ggT}(a, b)$.

Algorithmus:

1. Falls $a = b$, brich Berechnung ab. Es gilt $\mathrm{ggT}(a, b) = a$. 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 $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

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

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:

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

Beispiel: Euklidischer Algorithmus

Warum geht das?

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

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

Dann gilt:

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

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

Beispiel: Euklidischer Algorithmus

Hier sind $q_a, q_b, q'_{a-b}, q'_b \in \mathbb{Z}$.

Warum geht das?

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

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

Dann gilt:

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

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

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

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

Eigenschaften

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

(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

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

(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

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

(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

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

(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

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

(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)

Analyse von 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:

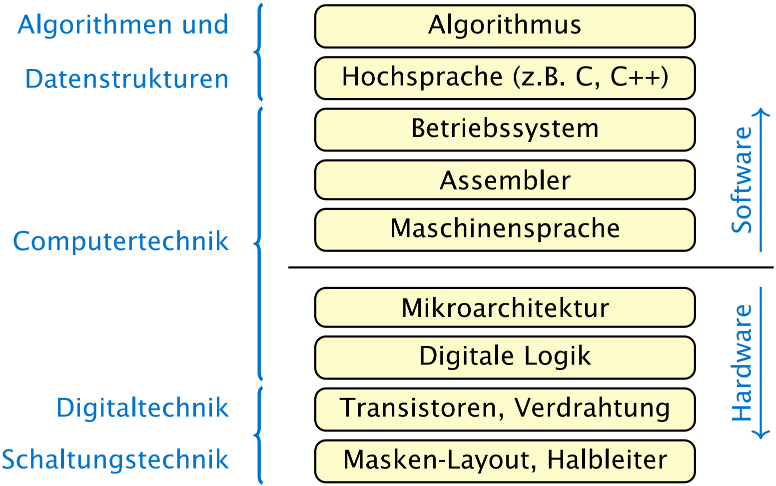
- Element auf Stapel legen – **push**
- Element von Stapel nehmen – **pop**

Operationen jeweils nur auf oberstem Element!

Weitere Beispiele von Datenstrukturen

- **Felder**, **Listen**, **Stack**, **Queue** $\longrightarrow$ Kapitel 3
- **Bäume**, **Graphen** $\longrightarrow$ 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.

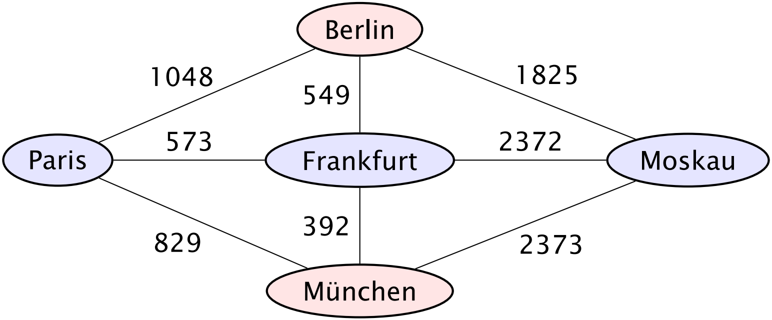
Berlin $\longrightarrow$  $\longrightarrow$ München

Einordnung Algorithmen und Datenstrukturen

Beispiel-Problem Navigationssystem Auto

Finde kürzesten Weg von Berlin nach München.

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

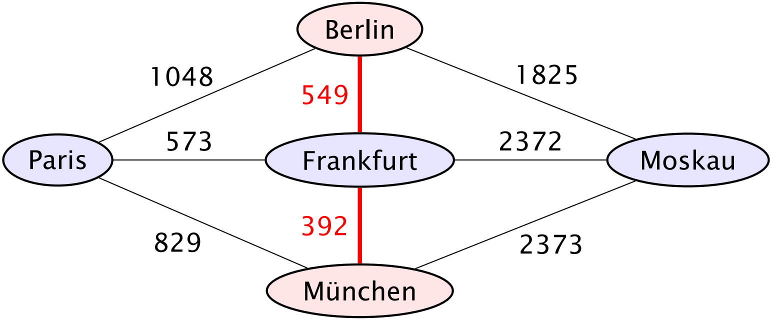


Einordnung Algorithmen und Datenstrukturen

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)

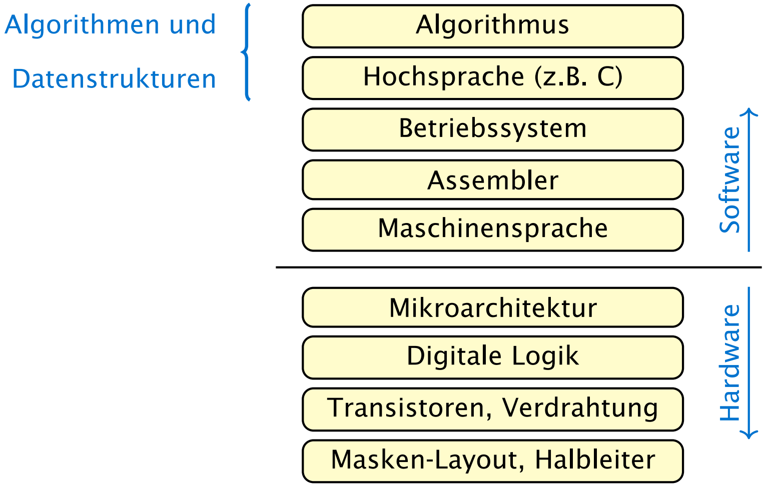
Einordnung Algorithmen und Datenstrukturen

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



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

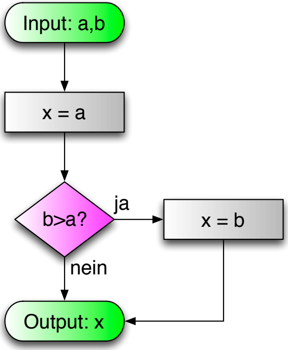
- informelle Veranschaulichung von Algorithmus
- nicht von Rechner ausführbar
- nicht standardisiert

```
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

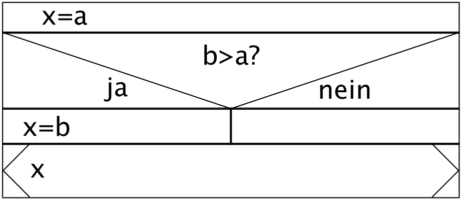


Darstellung von Algorithmen III

Struktogramm

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

max(a,b)



Darstellung von Algorithmen IV

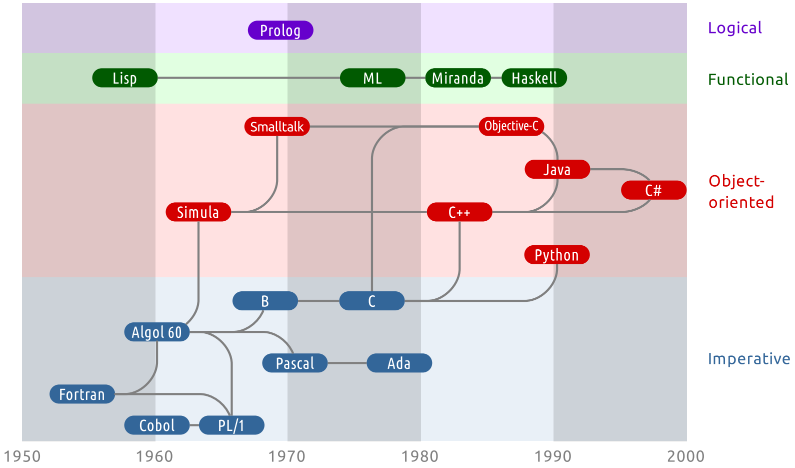
Programmiersprache

- 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:

```
int max(int a, int b) {
    int x = a;
    if (b > a)
        x = b;
    return x;
}
```

Programmiersprachen Übersicht



Grafik von Alexandru Duliu.

Äquivalenz von Algorithmen-Beschreibungen

Churchsche These

Alle „vernünftigen“ Definitionen von Algorithmen sind äquivalent.

Die fehlende Zutat für eine präzise Algorithmendefinition ist die Definition eines *elementaren Einzelschrittes*. Dies wird üblicherweise durch ein Maschinenmodells gemacht (z.B. Turingmaschine). Es gibt auch sehr esoterische Maschinenmodelle (z.B. *billiard ball computer*).

- alle gängigen Programmiersprachen leisten dasselbe
- jeder Computer ist äquivalent

Die These postuliert, dass alle diese Maschinenmodelle equivalent ist.

- formal: berechenbare Funktionen, formale Sprachen, Automaten, Turing-Maschinen

↑theoretische Informatik

Äquivalenz von Algorithmen-Beschreibungen

Churchsche These

Alle „vernünftigen“ Definitionen von Algorithmen sind äquivalent.

Die fehlende Zutat für eine präzise Algorithmendefinition ist die Definition eines *elementaren Einzelschrittes*. Dies wird üblicherweise durch ein Maschinenmodells gemacht (z.B. Turingmaschine). Es gibt auch sehr esoterische Maschinenmodelle (z.B. *billiard ball computer*).

- alle gängigen Programmiersprachen leisten dasselbe
- jeder Computer ist äquivalent

Die These postuliert, dass alle diese Maschinenmodelle equivalent ist.

- 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 erfuellt
      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: Euklidscher Algorithmus

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

```
euklid(a,b)
    if (a == 0)
        return b;
    while (b > 0) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
```

Euklidscher Algorithmus

Beispiel: Euklidscher Algorithmus

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

```
euklid(a,b)
    if (a == 0)
        return b;
    while (b > 0)
        if (a > b)
            a = a - b;
        else
            b = b - a;
    return a;
```

Euklidscher Algorithmus

Euklidscher Algorithmus

Input: Natürliche Zahlen a, b
Output: $\mathrm{ggT}(a, b)$

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

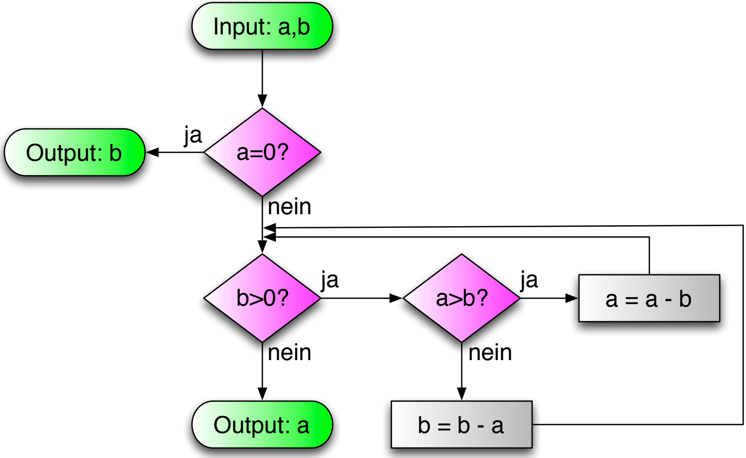
Euklidscher Algorithmus

Input: Natürliche Zahlen a, b
Output: $\mathrm{ggT}(a, b)$

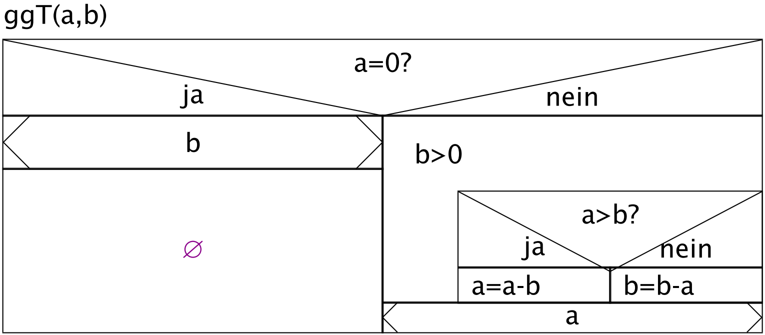
1. Falls $a = 0$ liefere b zurück
2. Solange $b > 0$ wiederhole
   Falls $a > b$ setze $a = a - b$
   sonst setze $b = b - a$
3. Liefere a zurück

$\longrightarrow$ **das ist Pseudocode!**

Euklidscher Algorithmus als Flussdiagramm



Euklidscher Algorithmus als Struktogramm



Euklidscher Algorithmus als Pseudocode

```
Input: natuerliche Zahlen a, b
Output: ggT(a,b)
euklid(a,b)
    if (a == 0)
        return b;
    while (b > 0) { // Hauptschleife
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
```

Euklidscher Algorithmus

Euklidscher Algorithmus als C

```c
int ggT(int a, int b)
{
    if (a==0)
        return b;
    while (b>0) {
        if (a>b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

Euklidscher Algorithmus als Python

```python
def ggT(a, b):
    if a == 0:
        return b
    while b > 0:
        if a > b:
            a = a - b
        else:
            b = b - a
    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 $f_1, f_2, f_3, \ldots$, für die gilt

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

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

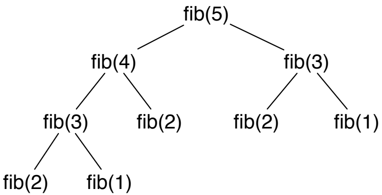
Beispiel: Fibonacci Funktion

```
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);
   }
```

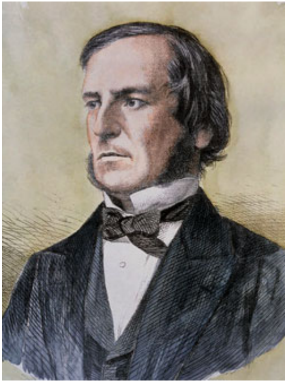
Beispiel: Fibonacci Funktion

```
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)

Logische Werte

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: $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

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

Disjunktion: $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

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

Negation: $\neg : \mathbb{B} \to \mathbb{B}$

- auch bezeichnet als NICHT bzw. NOT

Logische Werte und Verknüpfungen

„Grundrechenarten" mit logischen Werten:

Konjunktion: $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

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

Disjunktion: $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

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

Negation: $\neg : \mathbb{B} \to \mathbb{B}$

- 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:

Wahrheitstabelle:

Konjunktion: $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

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

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

Disjunktion: $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

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

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

Negation: $\neg : \mathbb{B} \to \mathbb{B}$

- auch bezeichnet als NICHT bzw. NOT

Logische Werte und Verknüpfungen

„Grundrechenarten“ mit logischen Werten:

Konjunktion: $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

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

Disjunktion: $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

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

Negation: $\neg : \mathbb{B} \to \mathbb{B}$

- 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: $\uparrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

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

NOR: $\downarrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

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

XOR: $\oplus\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- 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: $\uparrow: \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

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

NOR: $\downarrow: \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

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

XOR: $\oplus: \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- 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: $\uparrow: \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

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

NOR: $\downarrow: \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

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

XOR: $\oplus: \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- 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: $\Rightarrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- 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: $\Leftrightarrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- 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: $\Rightarrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- 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: $\Leftrightarrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- 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?

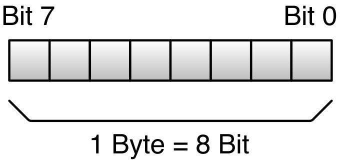
Primitive Datentypen
Wir bezeichnen grundlegende, in Programmiersprachen eingebaute Datentypen als primitive Datentypen.

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

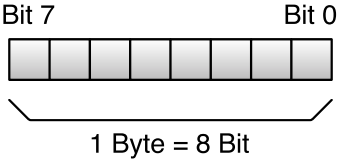
Bits und Bytes



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)

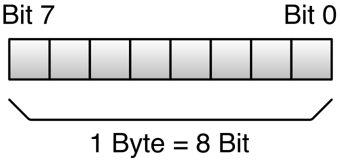
Bits und Bytes



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

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

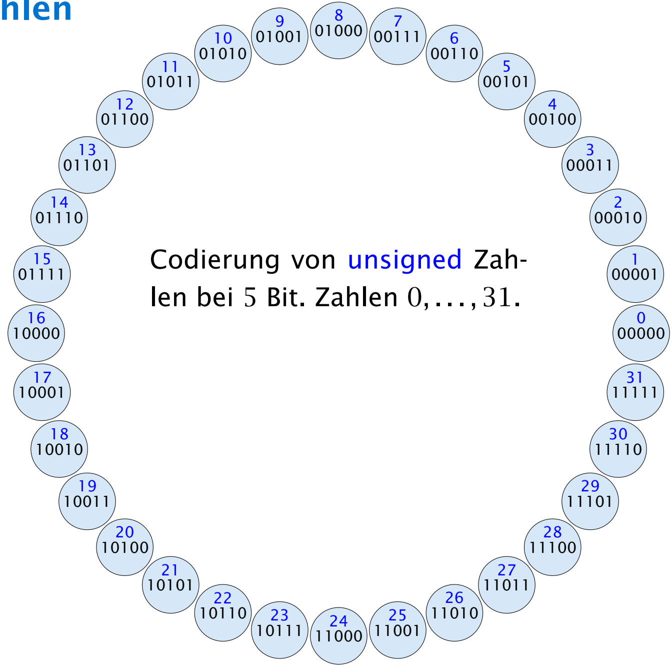
Natürliche Zahlen

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

Die primitiven Datentypen für natürliche Zahlen sind:

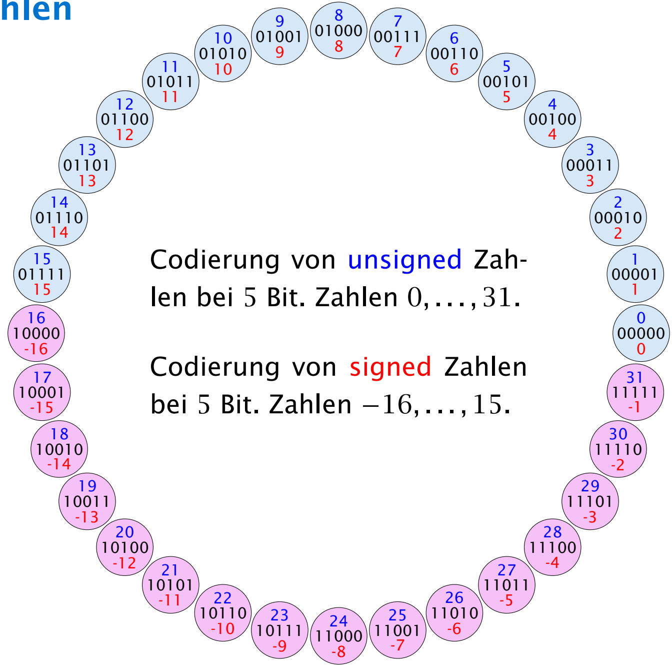
- **8 Bits** (ein Byte), darstellbare Zahlen: $\{0, \dots, 255\}$
  in C: `unsigned char`
- **16 Bits**, darstellbare Zahlen: $\{0, \dots, 65\,535\}$
  in C: `unsigned short`
- **32 Bits**, darstellbare Zahlen: $\{0, \dots, 4\,294\,967\,295\}$
  in C: `unsigned long`
- **64 Bits**, darstellbare Zahlen: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

Negative Zahlen



Codierung von unsigned Zahlen bei 5 Bit. Zahlen $0, \ldots, 31$.

Negative Zahlen



Codierung von unsigned Zahlen bei 5 Bit. Zahlen $0, \ldots, 31$.

Codierung von signed Zahlen bei 5 Bit. Zahlen $-16, \ldots, 15$.

Negative Zahlen

Bitfolge		**Zahl**
$x = \langle x_{n-1}, \dots, x_0 \rangle$	$\xrightarrow{f}$ / $\xleftarrow[f^{-1}]{}$	$\sum_{i=0}^{n-1} x_i 2^i$
$x = \langle x_{n-1}, \dots, x_0 \rangle$	$\xrightarrow{f_{\text{ZK}}}$ / $\xleftarrow[f_{\text{ZK}}^{-1}]{}$	$-x_{n-1} 2^n + \sum_{i=0}^{n-1} x_i 2^i$

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

Beobachtungen

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

Negative Zahlen

Beachte, dass die Regel für die Zahl -2^{n-1} nicht gelten kann, da $+2^{n-1}$ im 2er-Komplement mit n bits nicht darstellbar ist. Für die Null gilt die Regel nur wenn man die Addition $1 + f(\bar{x})$ modulo 2^n ausführt.

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. $f_{\text{ZK}}(x)$ negativ

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

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. $f_{\text{ZK}}(x)$ strikt positiv

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

Für eine Zahl z die das höchstwertige Bit $n-1$ gesetzt hat gilt $f_{\text{ZK}}^{-1}(z - 2^n) = f^{-1}(z)$. 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:

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

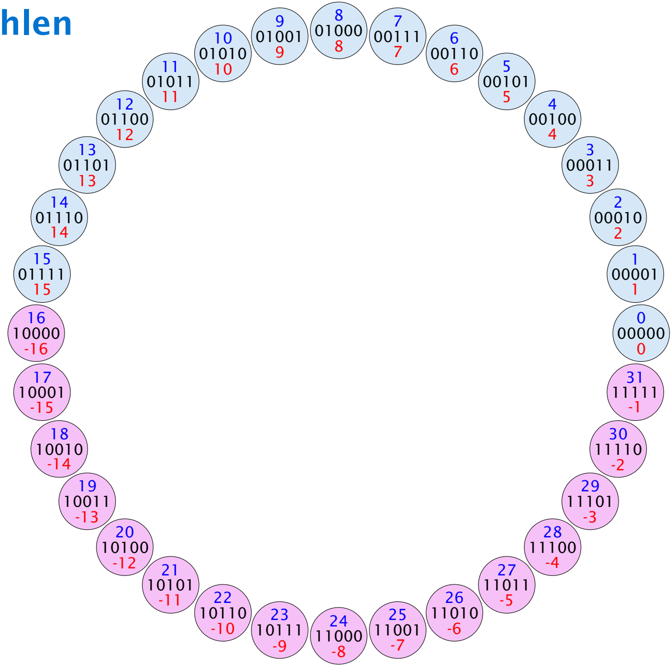
Negative Zahlen

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

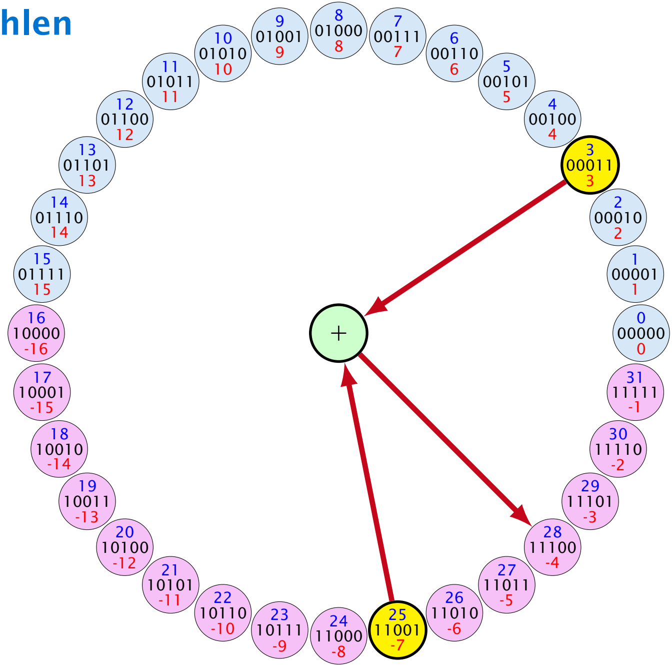
Im Prinzip wird eine Operation (Addition / Subtraktion / Multiplikation / Ganzahldivision) ausgeführt, und dann werden überzählige Bits verworfen (d.h., dass Ergebnis wird modulo 2^n genommen).

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

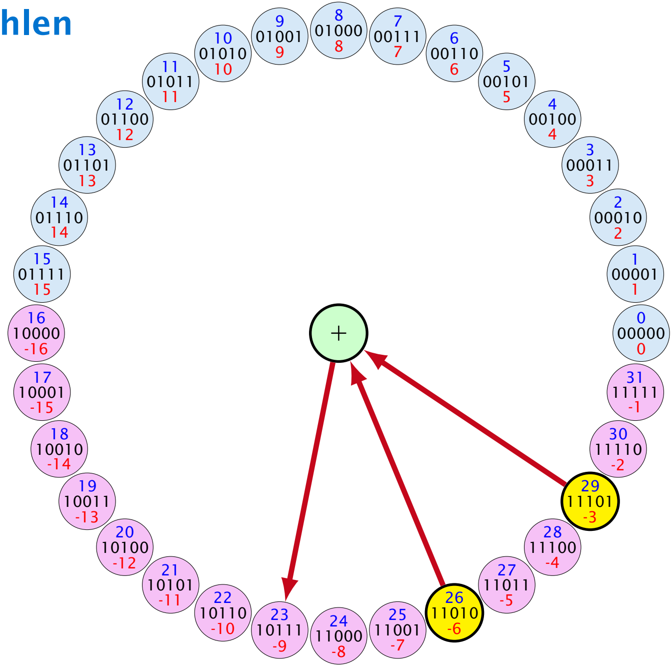
Negative Zahlen



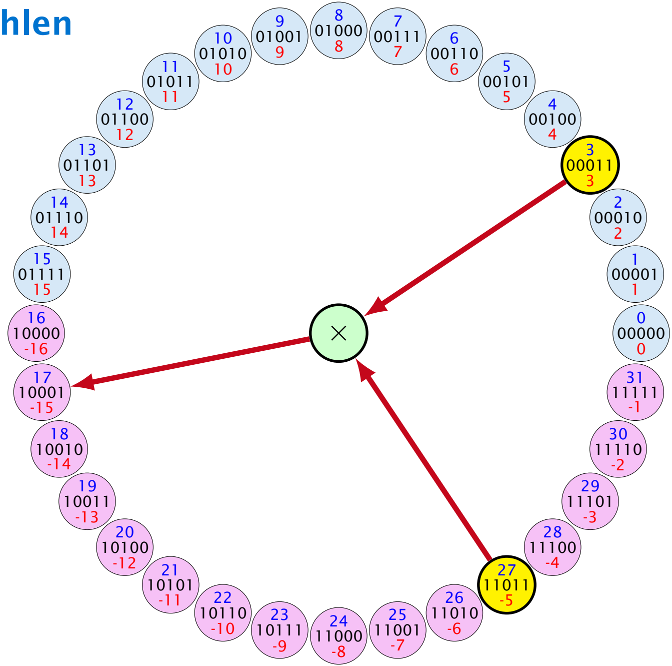
Negative Zahlen



Negative Zahlen



Negative Zahlen



Ganze Zahlen in C-ähnlichen Sprachen

Ganze Zahlen:

Die primitiven Datentypen für ganze Zahlen sind:

- **8 Bits:** `unsigned char` $\{0, \ldots, 255\}$
  `signed char` $\{-128, \ldots, 127\}$
- **16 Bits:** `unsigned short` $\{0, \ldots, 65535\}$
  `signed short` $\{-32768, \ldots, 32767\}$
- **32 Bits:** `unsigned long` $\{0, \ldots, 2^{32} - 1\}$
  `signed long` $\{-2^{31}, \ldots, 2^{31} - 1\}$
- **64 Bits:** `unsigned long long` $\{0, \ldots, 2^{64} - 1\}$
  `signed long long` $\{-2^{63}, \ldots, 2^{63} - 1\}$

- `signed` kann weggelassen werden (ausser bei `char`!)
- `unsigned int` und `signed int` sind je nach System 16, 32 oder 64 Bit

Dies sind nur Mindestvorgaben. Auf der Cray ist ein `short` z.B. 64 Bit lang.

Rationale Zahlen I

Festkommadarstellung: Komma an fester Stelle in Zahl

Beispiel mit $n = 16$:

$$\underbrace{c_{15}\ c_{14}\ c_{13}\ c_{12}\ c_{11}\ c_{10}\ c_{9}\ c_{8}}_{\text{ganzzahliger Anteil}}\ \Big|\ \underbrace{c_{7}\ c_{6}\ c_{5}\ c_{4}\ c_{3}\ c_{2}\ c_{1}\ c_{0}}_{\text{gebrochener Anteil}}$$

Komma

Nachteile:

- weniger große Zahlen darstellbar
- feste Genauigkeit der Nachkommastellen

Rationale Zahlen II

$$\underbrace{c_n \quad c_{n-1} \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad c_1 \quad c_0}_{\text{ganzzahliger Anteil}} \;\Big|\; \underbrace{c_{-1} \quad c_{-2} \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad c_{-m}}_{\text{gebrochener Anteil}}$$

Komma

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

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

mit $n + 1$ Vorkomma- und m Nachkommaziffern

Beispiel:

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

Floating Point Zahlen I

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

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

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

1 Bit	8 Bit	23 Bit	32 Bit float
1 Bit	11 Bit	52 Bit	64 Bit double
V	Exponent E	Mantisse M	

- wissenschaftliche Darstellung mit Basis 2

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

- Vorzeichen Bit V
- Mantisse M hat immer die Form $\texttt{1.abc}$, 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:

Bit	*Vorz.*	*Exp.*	*Mant.*	*Dezimal stellen*	*Bereich*
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: $0.1_{10} = 0.0001100110011\ldots_2$ (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: $0.1_{10} = 0.0001100110011\ldots_2$ (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: $0.1_{10} = 0.0001100110011\ldots_2$ (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: $0.1_{10} = 0.0001100110011\ldots_2$ (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 $(d_i)_{i=1,\ldots,n}$,

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

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

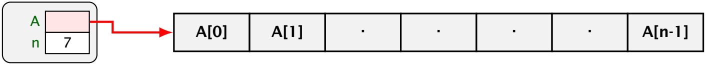
- `F.initialize()`: initialisiere leeres Feld `A`
- `F.elementAt(i)`: Zugriff auf `i`-tes Element von `A`.
  `i` muss zwischen `1` und `F.size()` liegen.
- `F.insert(d,i)`: füge Element `d` an Position `i` in Feld `A` ein.
  `i` muss zwischen `1` und `F.size()+1` liegen
- `F.erase(i)`: entferne `i`-tes 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

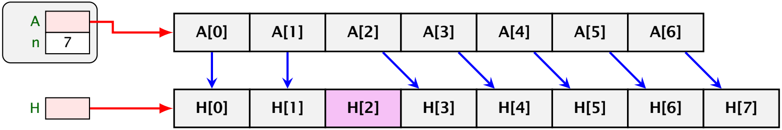
Hinzufügen eines Elementes

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

```
class Feld {
    int  n;
    int* A;

  public:
    Feld() {
        n = 0;
        A = new int[n];
    }
```

Implementierung: Feld via Array

```
    void insert(int d, int i) {
        int* H = new int[n+1];
        for (int j=0; j<i-1; j++) {
            H[j] = A[j];
        }
        H[i-1] = d;
        for (int j=i-1; j<n; j++) {
            H[j+1] = A[j];
        }
        delete[] A;
        A = H;
        n++;
    }
```

Implementierung: Feld via Array

```
void erase(int i) {
    int* H = new int[n-1];
    for (int j=0; j<i-1; j++) {
        H[j] = A[j];
    }
    for (int j=i; j<n; j++) {
        H[j-1] = A[j];
    }
    delete[] A;
    A = H;
    n--;
}
```

Implementierung: Feld via Array

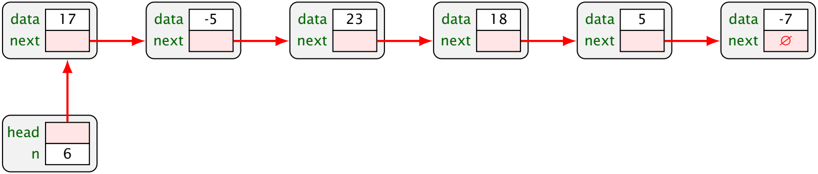
```
    int elementAt(int i) {
        return A[i-1];
    }

    int size() {
        return n;
    }
};
```

Feld als einfach verkettete Liste

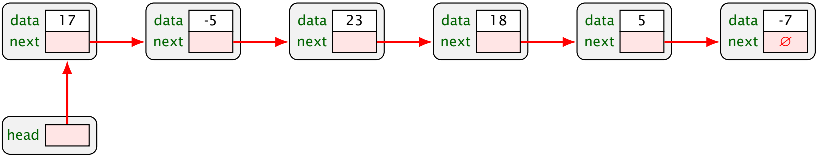
Repräsentation von Feld als verkettete Liste

- 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

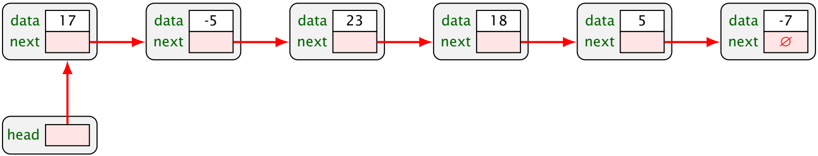
- **data**: Wert des Feldes an Position i
- **next**: Referenz auf das nächste Element

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

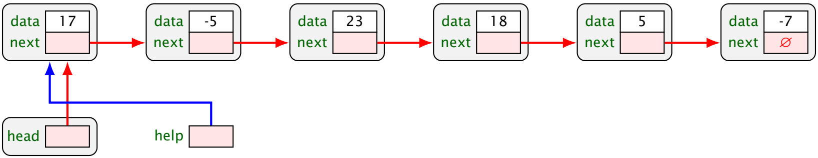
letztes Element hat keinen Nachfolger

- symbolisiert durch null-Referenz

Verketteter Liste – Zugriff auf Element

Zugriff auf Element i:

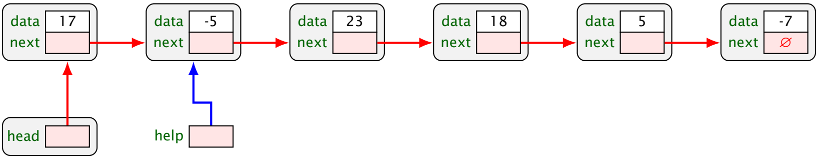
- beginne bei head-Referenz
- "vorhangeln" entlang next-Referenzen bis zum i-ten Element



Verketteter Liste – Zugriff auf Element

Zugriff auf Element i:

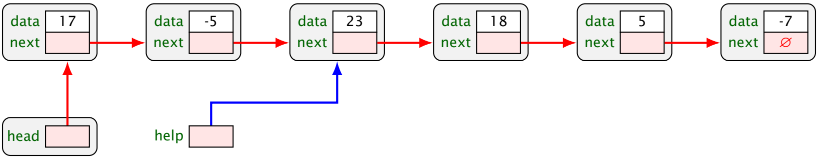
- beginne bei head-Referenz
- “vorhangeln” entlang next-Referenzen bis zum i-ten Element



Verketteter Liste – Zugriff auf Element

Zugriff auf Element i:

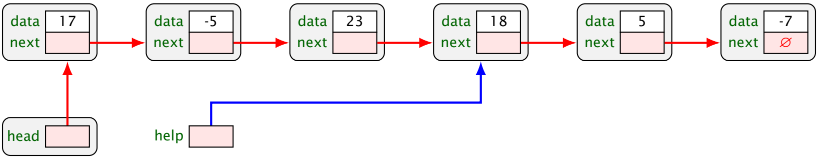
- beginne bei head-Referenz
- “vorhangeln” entlang next-Referenzen bis zum i-ten Element



Verketteter Liste – Zugriff auf Element

Zugriff auf Element i:

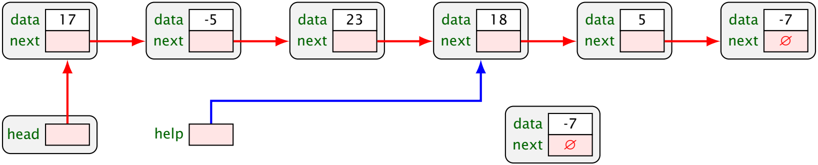
- beginne bei head-Referenz
- “vorhangeln” entlang next-Referenzen bis zum i-ten Element



Verketteter Liste – Einfügen nach Referenz

Einfügen nach Element i:

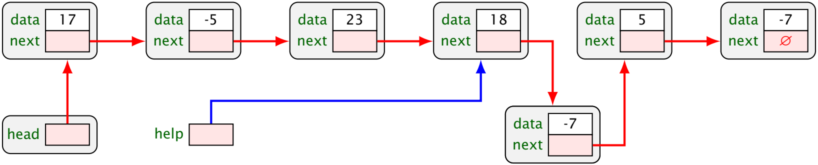
- beginne bei head-Referenz
- “vorhangeln” entlang next-Referenzen bis zum i-1-ten Element
- Referenzen umsetzen



Verketteter Liste – Einfügen nach Referenz

Einfügen nach Element i:

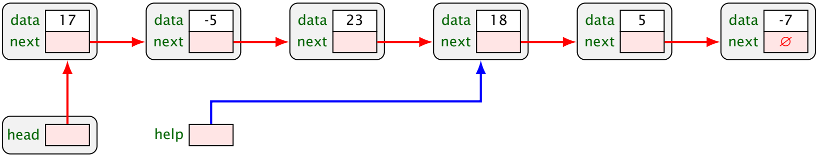
- beginne bei head-Referenz
- "vorhangeln" entlang next-Referenzen bis zum i-1-ten Element
- Referenzen umsetzen



Verketteter Liste – Löschen nach Referenz

Einfügen nach Element i:

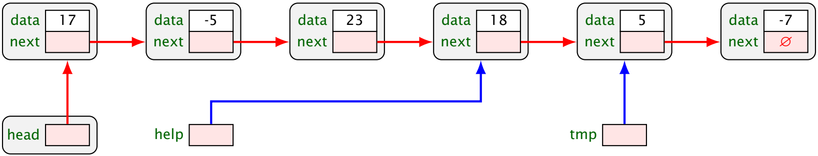
- beginne bei head-Referenz
- "vorhangeln" entlang next-Referenzen bis zum i-1-ten Element
- Referenzen umsetzen + Speicher freigeben



Verketteter Liste – Löschen nach Referenz

Einfügen nach Element i:

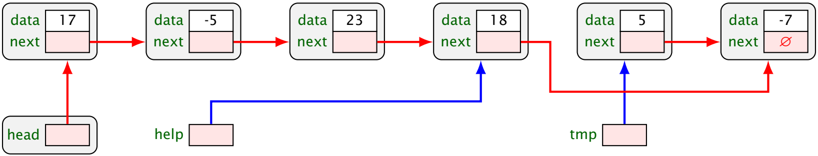
- beginne bei head-Referenz
- "vorhangeln" entlang next-Referenzen bis zum i-1-ten Element
- Referenzen umsetzen + Speicher freigeben



Verketteter Liste – Löschen nach Referenz

Einfügen nach Element i:

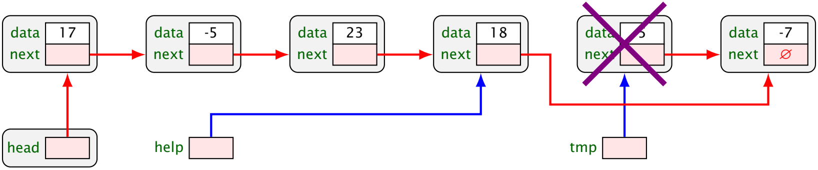
- beginne bei head-Referenz
- "vorhangeln" entlang next-Referenzen bis zum i-1-ten Element
- Referenzen umsetzen + Speicher freigeben



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

```
struct Node {
    int data;
    Node *next;

    Node(int d, Node* n) {
        data = d;
        next = n;
    }
};
```

Implementierung: Feld via Liste

```
class Feld {
    int n;
    Node *head;
  public:

    Feld() { // erzeuge leeres Feld
        n    = 0;
        head = NULL;
    }

    int size() { return n; }

    int elementAt(int i) {
        Node* h = head;
        while (i-- > 1)
            h = h->next;
        return h->data;
    }
```

Implementierung: Feld via Liste

```
    void insert(int d, int i) {
        Node* tmp = new Node(d, NULL);
        n++;
        if (i == 1) {
            tmp->next = head;
            head      = tmp;
            return;
        }
        Node* h = head;
        i--;
        while (i-- > 1)
            h = h->next;
        tmp->next = h->next;
        h->next   = tmp;
    }
```

Implementierung: Feld via Liste

```
    void erase(int i) {
        n--;
        if (i == 1) {
            Node* tmp = head;
            head = head->next;
            delete tmp;
            return;
        }
        Node* h = head;
        i--;
        while (i-- > 1)
            h = h->next;
        Node* tmp = h->next;
        h->next = h->next->next;
        delete tmp;
    }
}; // end class
```

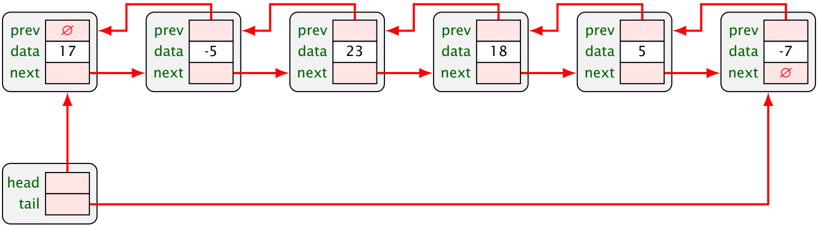
Gegenüberstellung Array und verkettete Liste

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

Feld als doppelt verkettete Liste

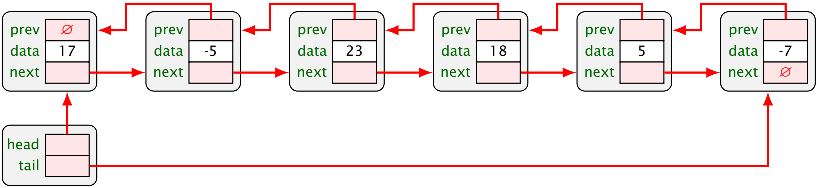
Repräsentation von Feld A als doppelt verkettete Liste

- 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

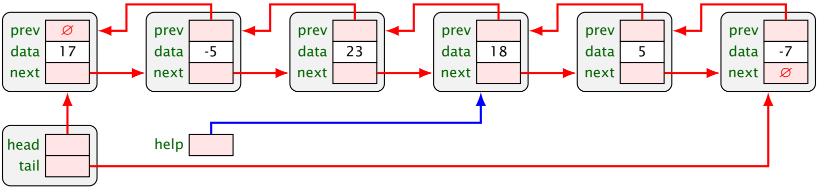
- `data`: Wert des Feldes
- `next`: Referenz auf das nächste Element
- `prev`: Referenz auf das vorherige Element

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

Operationen auf doppelt verketteter Liste

Löschen von Element i:

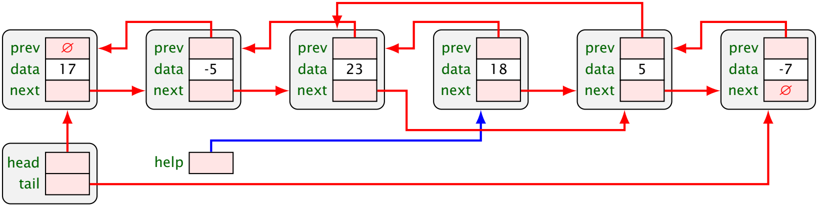
- Zugriff auf Element i
- umhängen von Referenzen
- Speicherplatz freigeben



Operationen auf doppelt verketteter Liste

Löschen von Element i:

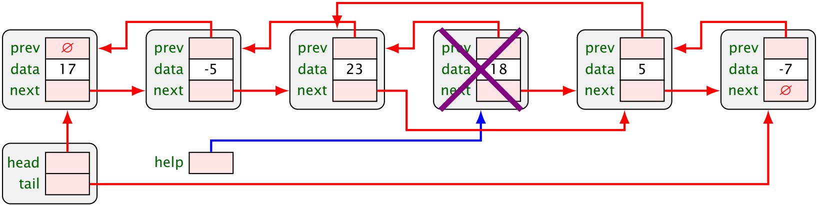
- Zugriff auf Element `i`
- umhängen von Referenzen
- Speicherplatz freigeben



Operationen auf doppelt verketteter Liste

Löschen von Element i:

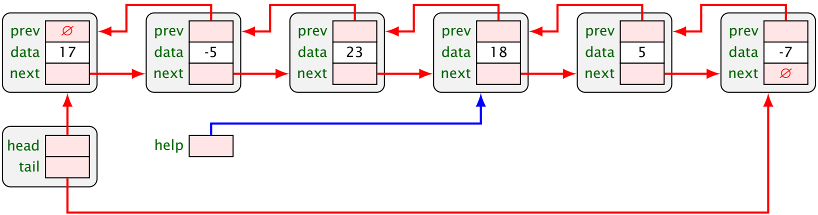
- Zugriff auf Element i
- umhängen von Referenzen
- Speicherplatz freigeben



Operationen auf doppelt verketteter Liste

Einfügen von Element an Stelle i:

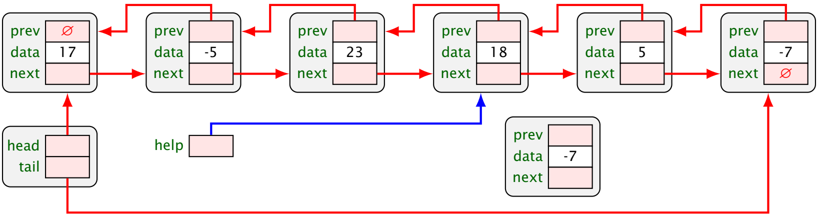
- Zugriff auf Element `i-1`
- umhängen von Referenzen



Operationen auf doppelt verketteter Liste

Einfügen von Element an Stelle i:

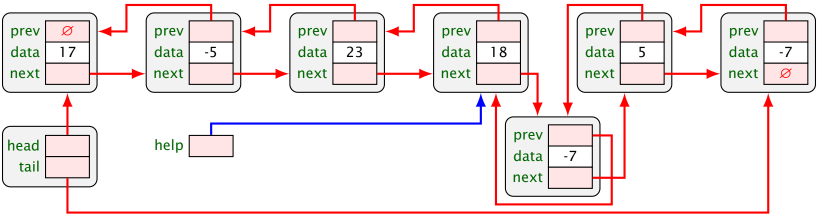
- Zugriff auf Element `i-1`
- umhängen von Referenzen



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

- `load(V)` liefert einen Wert
- `store(V, x)` wobei x ein Wert ist

und der Bedingung

- `load(V)` liefert immer den Wert x der letzten Operation `store(V, x)`

Beispiel abstrakter Datentyp: abstrakte Liste (Teil 1)

Abstrakte Liste `L` ist ein Datentyp

mit Operationen

- `pushFront(L, x)` liefert eine Liste
- `front(L)` liefert ein Element
- `rest(L)` liefert eine Liste

und den Bedingungen

- ist `x` Element, `L` Liste, dann liefert `front(pushFront(L, x))` das Element x.
- ist `x` Element, `L` Liste, dann liefert `rest(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()` $\neq$ `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

- `pop(S)` liefert einen Wert
- `push(S, x)` wobei `x` ein Wert

mit der Bedingung

- ist `x` Wert und `V` Variable, dann ist die Sequenz `push(S, x); V = pop(S);` äquivalent zu `V = x;`

Definition Stack (exakter)

Stack S ist ein abstrakter Datentyp mit Operationen

- `pop(S)` liefert einen Wert
- `push(S, x)` wobei `x` ein Wert

mit der Bedingung

- ist `x` Wert und `V` Variable, dann ist die Sequenz `push(S, x); V = pop(S);` äquivalent zu `V = x;`

sowie der Operation

- `top(S)` liefert einen Wert

mit der Bedingung

- ist `x` Wert und `V` Variable, dann ist die Sequenz `push(S, x); V = top(S);` äquivalent zu `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

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

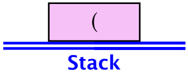
Beispiel: Auswertung arithmetischer Ausdrücke

Stack

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

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

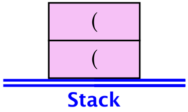
Beispiel: Auswertung arithmetischer Ausdrücke



(	(	(	3	+	1	)	·	7	)	+	(	14	−	3	)	)

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

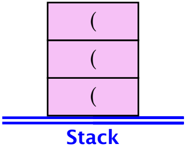
Beispiel: Auswertung arithmetischer Ausdrücke



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

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

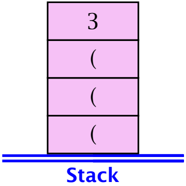
Beispiel: Auswertung arithmetischer Ausdrücke



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

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

Beispiel: Auswertung arithmetischer Ausdrücke



(	(	(	3	+	1	)	·	7	)	+	(	14	−	3	)	)

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

Beispiel: Auswertung arithmetischer Ausdrücke

+
3
(
(
(

Stack

(	(	(	3	**+**	1	)	·	7	)	+	(	14	−	3	)	)

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

Beispiel: Auswertung arithmetischer Ausdrücke

1
+
3
(
(
(

Stack

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

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

Beispiel: Auswertung arithmetischer Ausdrücke

1
+
3
(
(
(

Stack

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

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

Beispiel: Auswertung arithmetischer Ausdrücke

4
(
(

Stack

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

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

Beispiel: Auswertung arithmetischer Ausdrücke

$\cdot$
4
(
(

Stack

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

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

Beispiel: Auswertung arithmetischer Ausdrücke

7
$\cdot$
4
$($
$($

Stack

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

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

Beispiel: Auswertung arithmetischer Ausdrücke

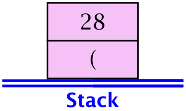
7
$\cdot$
4
(
(

Stack

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

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

Beispiel: Auswertung arithmetischer Ausdrücke



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

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

Beispiel: Auswertung arithmetischer Ausdrücke

$+$
28
$($

Stack

(	(	(	3	+	1	)	·	7	)	+	(	14	−	3	)	)

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

Beispiel: Auswertung arithmetischer Ausdrücke

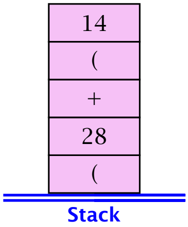
(
+
28
(

Stack

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

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

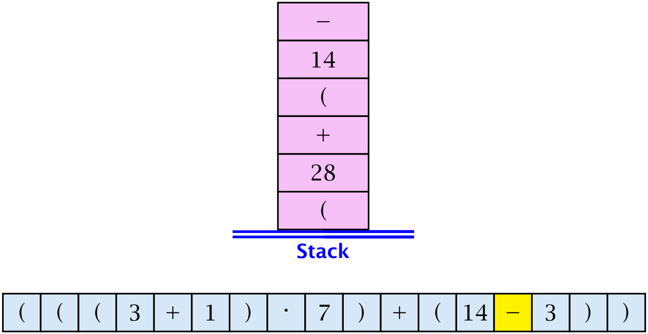
Beispiel: Auswertung arithmetischer Ausdrücke



(	(	(	3	+	1	)	·	7	)	+	(	14	−	3	)	)

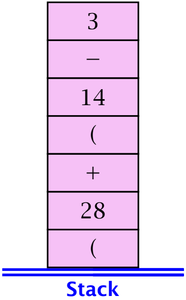
$$(((3 + 1) \cdot 7) + (14 - 3))$$

Beispiel: Auswertung arithmetischer Ausdrücke



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

Beispiel: Auswertung arithmetischer Ausdrücke



(	(	(	3	+	1	)	·	7	)	+	(	14	−	3	)	)

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

Beispiel: Auswertung arithmetischer Ausdrücke

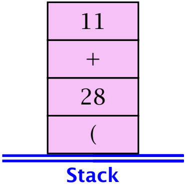
3
$-$
14
(
$+$
28
(

Stack

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

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

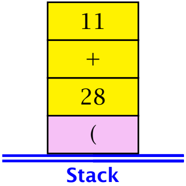
Beispiel: Auswertung arithmetischer Ausdrücke



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

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

Beispiel: Auswertung arithmetischer Ausdrücke



(	(	(	3	+	1	)	·	7	)	+	(	14	−	3	)	)

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

Beispiel: Auswertung arithmetischer Ausdrücke

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:

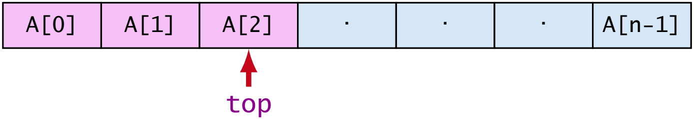
- Array
- verkettete Liste

Implementation Stack via Array

Stack-Elemente im Array (Länge n) speichern

oberstes Stack-Element merken mittels Variable top

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

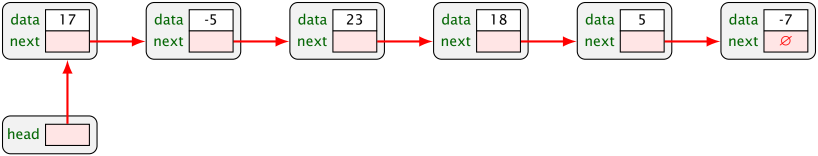


- `push(x)` inkrementiert `top` und speichert `x` in `A[top]`
- `pop()` liefert `A[top]` zurück und dekrementiert `top`
- `top()` liefert `A[top]` zurück

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

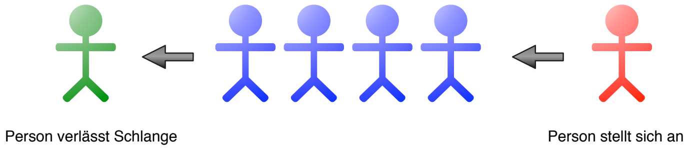
- dynamische Größe, aber Platz für Zeiger "verschwendet"
- push, pop effizient
- eventuell nicht cache-effizient

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()` $\neq$ `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()` $\neq$ `Q` für jede Queue `Q`
- `isEmpty(initialize()) == true`
- `isEmpty(enqueue(Q, x)) == false`

Definition Queue (exakter)

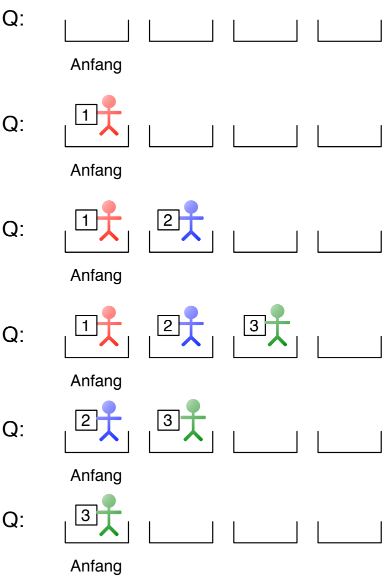
Queue `Q` ist ein abstrakter Datentyp mit Operationen

- `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()` $\neq$ `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

```
Input: Zeichenkette str mit Laenge n
Output: true falls str Palindrom; sonst false

Stack S;
Queue Q;

i = 0;
while (i<n)
    S.push(str[i]);
    Q.enqueue(str[i]);
    i++;
i = 0;
while (i<n)
    s = S.pop();
    q = Q.dequeue();
    if (s != q) return false;
    i++;
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:

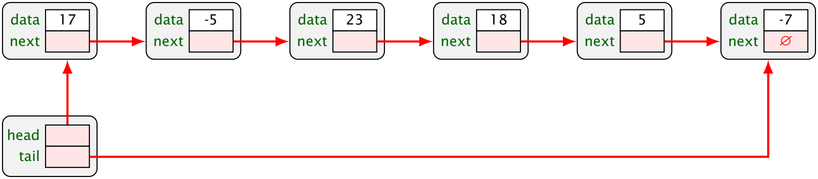
- verkettete Liste
- Array

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



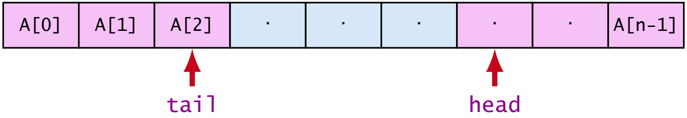
- enqueue(x) fügt Element bei head-Referenz ein
- dequeue() liefert Element bei tail-Referenz zurück und entfernt es

Implementation Queue via Array

Queueelemente in Array (Länge n) speichern

Anfang der Queue wird durch Index head markiert

Ende der Queue wird durch Index `tail` markiert



- `enqueue(x)` fügt Element bei Index `(tail+1)%n` ein
- dequeue liefert Element bei Index head zurück und entfernt es durch Inkrement von head (head=(head+1)%n)

Implementation Queue als zwei Stacks

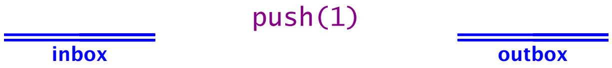
Queue `Q` kann mittels zwei Stacks implementiert werden

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

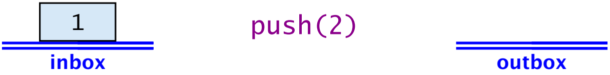
Queue `Q` kann mittels zwei Stacks implementiert werden

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

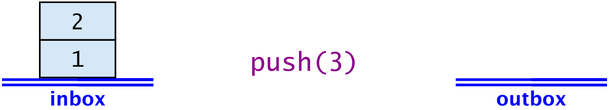
Queue `Q` kann mittels zwei Stacks implementiert werden

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

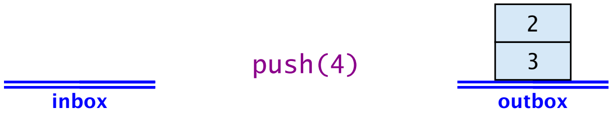
Queue `Q` kann mittels zwei Stacks implementiert werden

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

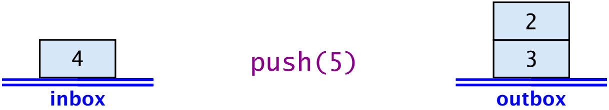
Queue `Q` kann mittels zwei Stacks implementiert werden

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

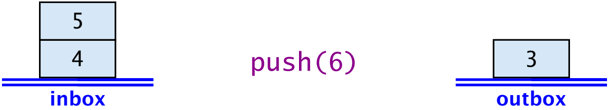
Queue `Q` kann mittels zwei Stacks implementiert werden

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

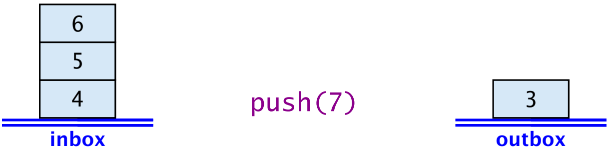
Queue `Q` kann mittels zwei Stacks implementiert werden

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



Implementation Queue als zwei Stacks

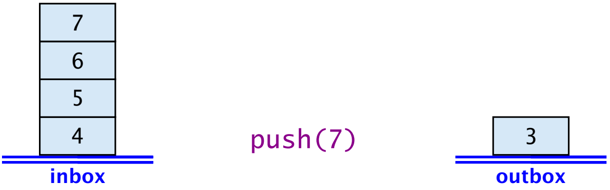
Queue `Q` kann mittels zwei Stacks implementiert werden

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.
  - Gibt [illegible] wie z.B. „dieser Algorithmus läuft immer in Zeit $\mathcal{O}(n^2)$“.
  - Üblicherweise wird der [illegible] 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 [illegible] wie z.B. „dieser Algorithmus läuft immer in Zeit $\mathcal{O}(n^2)$".
  - Üblicherweise wird der [illegible] 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.

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 [illegible] wie z.B. „dieser Algorithmus läuft immer in Zeit $O(n^2)$".
  - Üblicherweise wird der [illegible] 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.

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

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 $f : \mathbb{N} \to \mathbb{N}$ 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 $f : \mathbb{N} \to \mathbb{N}$ 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 $\{1, \ldots, N\}$ 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 $f : \mathbb{N} \to \mathbb{N}$ 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 $f : \mathbb{N} \to \mathbb{N}$ 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 $\{1, \ldots, N\}$ sollen sortiert werden. Wir sagen üblicherweise, dass die Eingabelänge n ist, anstatt z.B. $n \log N$, was der Anzahl an Bits entsprechen würde.

Rechenmodell

Wie messen wir

Rechenmodell

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.

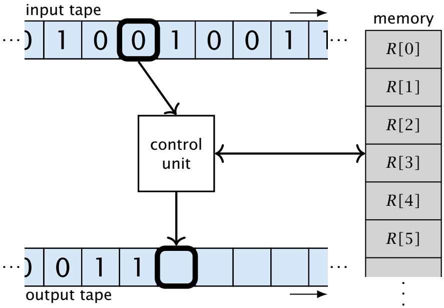
Rechenmodell

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)

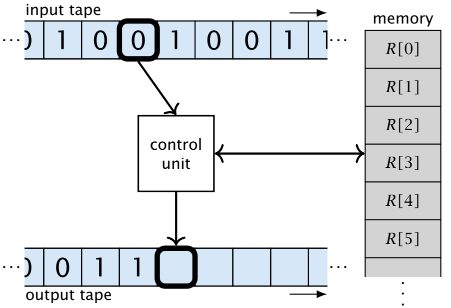
- Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- Speicher: unendlich viele Register $R[0], R[1], R[2], \ldots$.
- Register können beliebige Integer speichern.
- Indirekte Adressierung.



Ein- und Ausgabeband sind gerichtet und ein Lese- oder Schreibzugriff bewegt das entsprechend Band zur nächsten Position.

Random Access Machine (RAM)

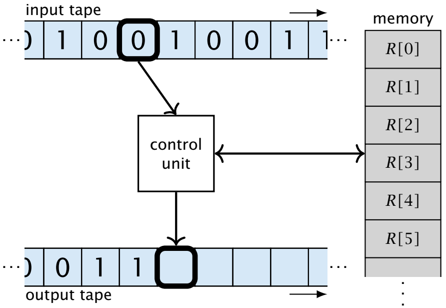
- Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- Speicher: unendlich viele Register $R[0], R[1], R[2], \ldots$.
- Register können beliebige Integer speichern.
- Indirekte Adressierung.



Ein- und Ausgabeband sind gerichtet und ein Lese- oder Schreibzugriff bewegt das entsprechend Band zur nächsten Position.

Random Access Machine (RAM)

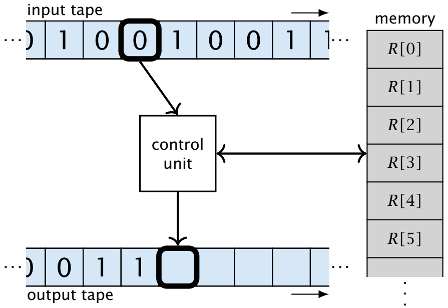
- Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- Speicher: unendlich viele Register $R[0], R[1], R[2], \ldots$.
- Register können beliebige Integer speichern.
- Indirekte Adressierung.



Ein- und Ausgabeband sind gerichtet und ein Lese- oder Schreibzugriff bewegt das entsprechend Band zur nächsten Position.

Random Access Machine (RAM)

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



Ein- und Ausgabeband sind gerichtet und ein Lese- oder Schreibzugriff bewegt das entsprechend Band zur nächsten Position.

Random Access Machine (RAM)

Operationen

- Eingabeoperationen (input tape $\to R[i]$)
  - READ i
- Ausgabeoperationen ($R[i] \to$ 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 Registes 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 → $R[i]$)
  - READ i
- Ausgabeoperationen ($R[i]$ → output tape)
- Registertransfers
- **indirekte** Adressierung

Random Access Machine (RAM)

Operationen

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

Random Access Machine (RAM)

Operationen

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - `READ` i
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - `WRITE` i

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 Registers 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

Random Access Machine (RAM)

Operationen

- Eingabeoperationen (input tape → $R[i]$)
  - `READ` i
- Ausgabeoperationen ($R[i]$ → 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 Registers 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 $\to R[i]$)
  - READ i
- Ausgabeoperationen ($R[i] \to$ 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 Registes 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 Registes 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 $\to R[i]$)
  - `READ` i
- Ausgabeoperationen ($R[i] \to$ 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 Registes 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: $+$, $-$, $\times$, $/$
  - $R[i]$:= $R[j]$ + $R[k]$;
    $R[i]$:= -$R[k]$;

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

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: $+$, $-$, $\times$, $/$

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

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: $+$, $-$, $\times$, $/$
  - $R[i] := R[j] + R[k];$
    $R[i] := -R[k];$

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

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: $+, -, \times, /$
  - $R[i] := R[j] + R[k];$
    $R[i] := -R[k];$

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

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: $+$, $-$, $\times$, $/$
  - $R[i]$:= $R[j] + R[k]$;
    $R[i]$:= $-R[k]$;

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

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: $+$, $-$, $\times$, $/$
  - $R[i]$ `:=` $R[j]$ `+` $R[k]$;
    $R[i]$ `:= -`$R[k]$;

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

Rechenmodell

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

Komplexitätsschranken

Es gibt unterschiedliche Komplexitätsschranken:

- best-case Komplexität:

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

  Normalerweise einfach zu analysieren; nicht sehr hilfreich

- worst-case Komplexität:

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

  Standard. Machmal zu pessimistisch.

- average case Komplexität:

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

  Manchmal schwierig zu analysieren.

$C(x)$	Kosten für Eingabe x
$\lvert x\rvert$	Eingabelänge von x
I_n	Menge der Eingaben mit Länge n

Komplexitätsschranken

Es gibt unterschiedliche Komplexitätsschranken:

- best-case Komplexität:

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

  Normalerweise einfach zu analysieren; nicht sehr hilfreich

- worst-case Komplexität:

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

  Standard. Machmal zu pessimistisch.

- average case Komplexität:

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

  Manchmal schwierig zu analysieren.

$C(x)$	Kosten für Eingabe x
$\lvert x\rvert$	Eingabelänge von x
I_n	Menge der Eingaben mit Länge n

Komplexitätsschranken

Es gibt unterschiedliche Komplexitätsschranken:

- best-case Komplexität:

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

  Normalerweise einfach zu analysieren; nicht sehr hilfreich

- worst-case Komplexität:

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

  Standard. Machmal zu pessimistisch.

- average case Komplexität:

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

  Manchmal schwierig zu analysieren.

$C(x)$	Kosten für Eingabe x
$\lvert x\rvert$	Eingabelänge von x
I_n	Menge der Eingaben mit Länge n

Komplexitätsschranken

Es gibt unterschiedliche Komplexitätsschranken:

- best-case Komplexität:

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

  Normalerweise einfach zu analysieren; nicht sehr hilfreich

- worst-case Komplexität:

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

  Standard. Machmal zu pessimistisch.

- average case Komplexität:

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

  Manchmal schwierig zu analysieren.

$C(x)$	Kosten für Eingabe x
$\lvert x\rvert$	Eingabelänge von x
I_n	Menge der Eingaben mit Länge n

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 $\mathbb{R}_0^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$

Asymptotische Notation

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

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim_{n\to\infty} \dfrac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim_{n\to\infty} \dfrac{g(n)}{f(n)} \leq \infty$

Asymptotische Notation

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

- $g \in \mathcal{O}(f)$: $\quad 0 \le \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \le \infty$
- $g \in \Theta(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$

Asymptotische Notation

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

- $g \in \mathcal{O}(f)$: $\quad 0 \le \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \le \infty$
- $g \in \Theta(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in o(f)$: $\quad \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Asymptotische Notation

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

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq \infty$
- $g \in \Theta(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in o(f)$: $\quad \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$
- $g \in \omega(f)$: $\quad \lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$

Asymptotische Notation

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 $f : \mathbb{N} \to \mathbb{R}^+, n \mapsto f(n)$, and $g : \mathbb{N} \to \mathbb{R}^+, n \mapsto g(n)$.
3. Man schreibt $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$, anstatt $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$.

Asymptotische Notation

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 $f : \mathbb{N} \to \mathbb{R}^+, n \mapsto f(n)$, and $g : \mathbb{N} \to \mathbb{R}^+, n \mapsto g(n)$.
3. Man schreibt $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$, anstatt $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$.

Asymptotische Notation

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 $f : \mathbb{N} \to \mathbb{R}^+, n \mapsto f(n)$, and $g : \mathbb{N} \to \mathbb{R}^+, n \mapsto g(n)$.
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

$$\Big\{f : \mathbb{N} \to \mathbb{R}^+ \mid f(n) = n^2 \cdot g(n) + h(n) \text{ mit } g(n) \in \mathcal{O}(n) \text{ und } h(n) \in \mathcal{O}(\log n)\Big\}$$

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 Ω und Θ.

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 Ω *und* Θ*.*

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 Ω *und* Θ*.*

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 Ω und Θ.

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 Ω und Θ.

Rechenregel für $\mathcal{O}$-Notation

Zu zeigen:

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

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

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

Rechenregel für $\mathcal{O}$-Notation

Zu zeigen:

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

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	***Eingabelänge n***							
$f(n)$	**10**	10^2	10^3	10^4	10^5	10^6	10^7	10^8
$\log n$	33ns	66ns	0.1µs	0.1µs	0.2µs	0.2µs	0.2µs	0.3µs
$\sqrt{n}$	32ns	0.1µs	0.3µs	1µs	3.1µs	10µs	31µs	0.1ms
n	100ns	1µs	10µs	0.1ms	1ms	10ms	0.1s	1s
$n \log n$	0.3µs	6.6µs	0.1ms	1.3ms	16ms	0.2s	2.3s	27s
$n^{3/2}$	0.3µs	10µs	0.3ms	10ms	0.3s	10s	5.2min	2.7h
n^2	1µs	0.1ms	10ms	1s	1.7min	2.8h	11d	3.2y
n^3	10µs	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	10µs	$4 \cdot 10^{14}$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

Sofern sich die angegebenen Laufzeiten auf ein Problem beziehen (und nicht auf einen konkreten Algorithmus zu Lösung eines Problems) handelt es sich um einen typischen Lösungsansatz für das jeweilige Problem. Zum Beispiel hat das Standardverfahren für die Matrixmultiplikation eine Laufzeit von $\Theta(n^3)$. Es gibt aber bessere Verfahren.

$\Theta(n \log n)$: "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

Ein wichtiges offenes Problem ist es ob es z.B. für das TSP-Problem einen Algorithmus mit polynomieller Laufzeit gibt.

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

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

Asymptotische Notation

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

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

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**:
  - Algorithmus A: Laufzeit $f(n) = 1000 \log n = O(\log n)$.
  - Algorithmus B: Laufzeit $g(n) = \log^2 n$.

  Es gilt $f = o(g)$. Aber solange $\log n \le 1000$ 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.

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!

$\Longrightarrow$ Sortieren durch Einfügen (InsertionSort)

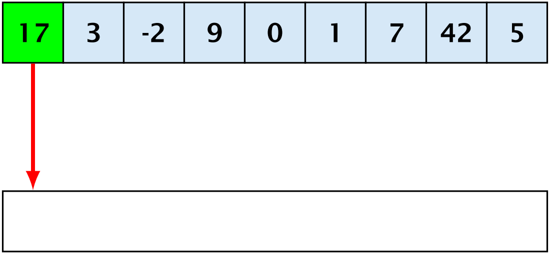
Beispiel

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

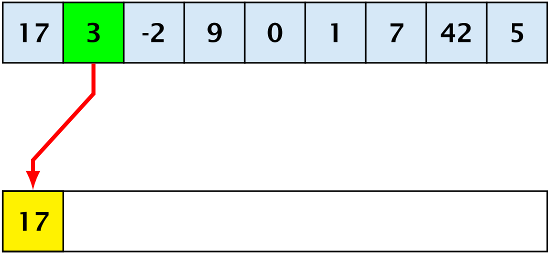
17	

Beispiel

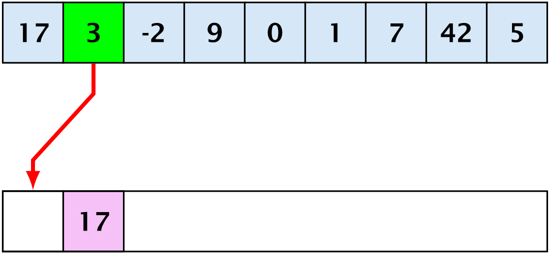
17	3	-2	9	0	1	7	42	5

17	

Beispiel



Beispiel

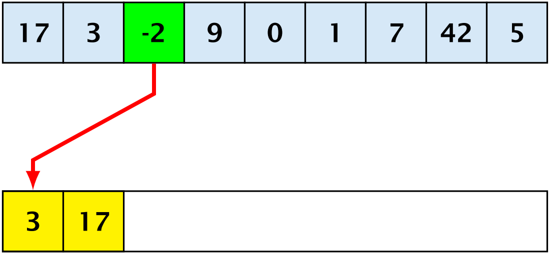


Beispiel

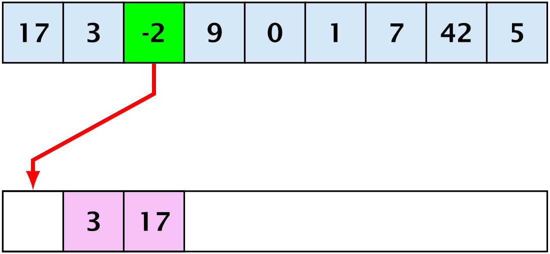
17	3	-2	9	0	1	7	42	5

3	17	

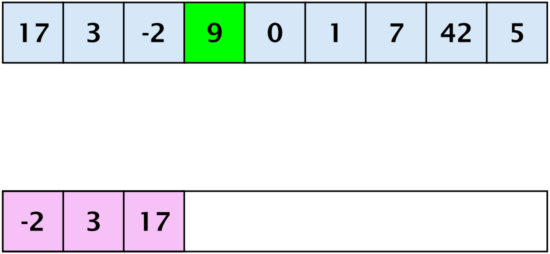
Beispiel



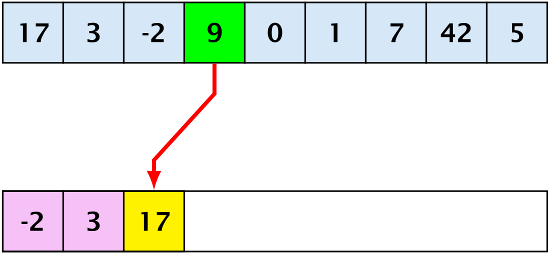
Beispiel



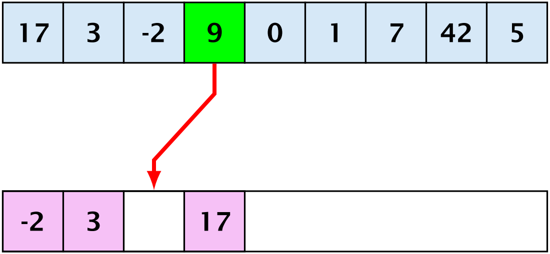
Beispiel



Beispiel



Beispiel

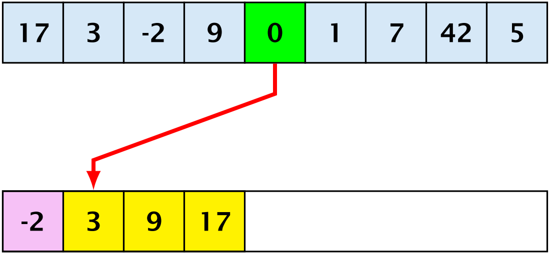


Beispiel

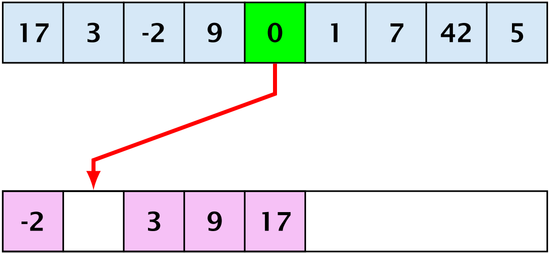
17	3	-2	9	0	1	7	42	5

-2	3	9	17	

Beispiel



Beispiel

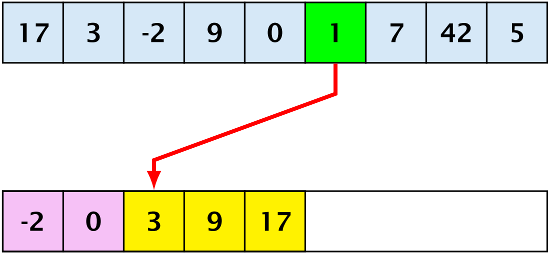


Beispiel

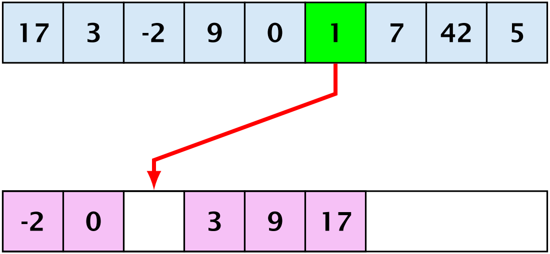
17	3	-2	9	0	1	7	42	5

-2	0	3	9	17				

Beispiel



Beispiel

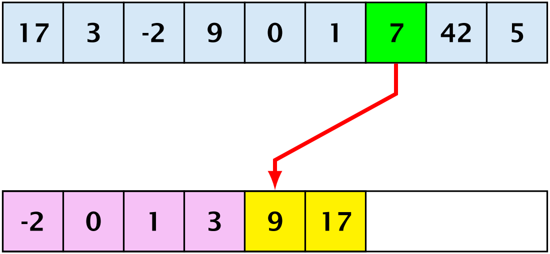


Beispiel

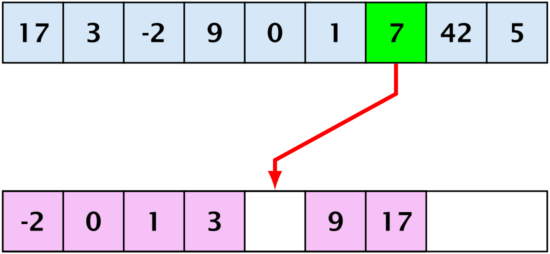
17	3	-2	9	0	1	7	42	5

-2	0	1	3	9	17			

Beispiel



Beispiel

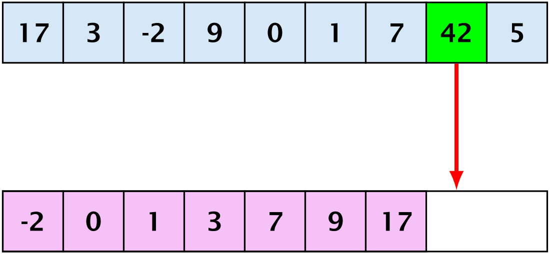


Beispiel

17	3	-2	9	0	1	7	42	5

-2	0	1	3	7	9	17		

Beispiel

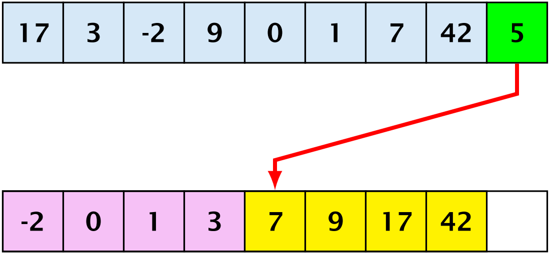


Beispiel

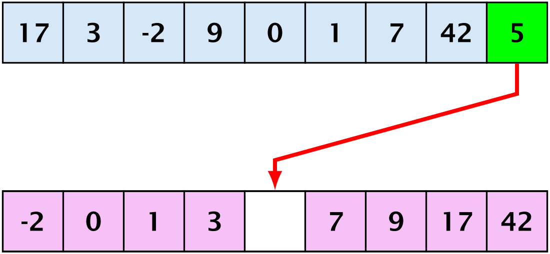
17	3	-2	9	0	1	7	42	5

-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:

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:

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:

3	17	-2	9	0	1	7	42	5

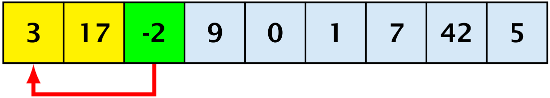
Beispiel

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:

-2	3	17	9	0	1	7	42	5

Beispiel

Wir brauchen kein zweites Array:

-2	3	17	9	0	1	7	42	5

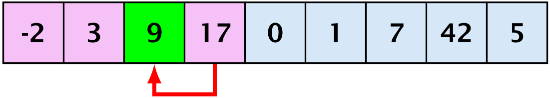
Beispiel

Wir brauchen kein zweites Array:

-2	3	17	9	0	1	7	42	5

Beispiel

Wir brauchen kein zweites Array:



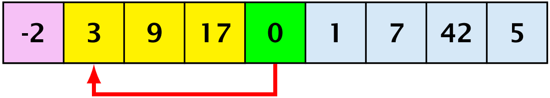
Beispiel

Wir brauchen kein zweites Array:

-2	3	9	17	0	1	7	42	5

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:

-2	0	3	9	17	1	7	42	5

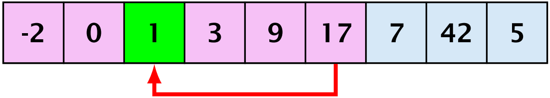
Beispiel

Wir brauchen kein zweites Array:

-2	0	3	9	17	1	7	42	5

Beispiel

Wir brauchen kein zweites Array:



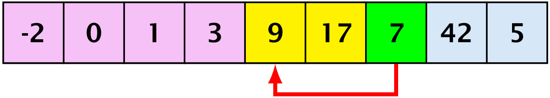
Beispiel

Wir brauchen kein zweites Array:

-2	0	1	3	9	17	7	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:

-2	0	1	3	7	9	17	42	5

Beispiel

Wir brauchen kein zweites Array:

-2	0	1	3	7	9	17	42	5

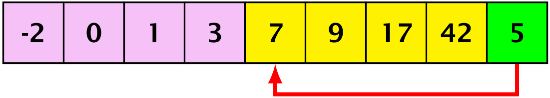
Beispiel

Wir brauchen kein zweites Array:

-2	0	1	3	7	9	17	42	5

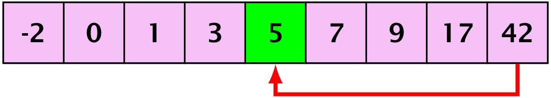
Beispiel

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

```
Input: Array A mit n Elementen
Output: Array A aufsteigend sortiert

InsertionSort(A)
    j = 0;
    while (j < n)
        key = A[j];
        i = j-1;
        while (i >= 0 && A[i] > key)
            A[i+1] = A[i];
            i--;
        A[i+1] = key;
        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

```
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++;
```

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

Laufzeit InsertionSort

$$\mathcal{O}(1) + (n+1)\mathcal{O}(1) + (n-1)\mathcal{O}(1) + \mathcal{O}(1)\sum_{j=0}^{n-1} t_j +$$

$$\mathcal{O}(1)\sum_{j=0}^{n-1}(t_j - 1) + \mathcal{O}(1)\sum_{j=0}^{n-1}(t_j - 1) + n\mathcal{O}(1) + n\mathcal{O}(1)$$

$$= \mathcal{O}(1)n + \mathcal{O}(1)\sum_{j=0}^{n-1} t_j$$

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

Die Laufzeit hängt stark vom Input ab.

Laufzeit InsertionSort

Best-case:
Wenn das Array sortiert wird ist, ist $t_j = 1$.
$\Longrightarrow$ Laufzeit: $\mathcal{O}(n)$.

Worst-case:
Wenn das Array absteigend sortiert ist, ist $t_j = j + 1$.
$\Longrightarrow$ Laufzeit: $\mathcal{O}(n^2)$.

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

```
Input:  Array A mit Laenge n; Element x
Output: i mit A[i] == x falls existent
        sonst -1

find(A,x)
    i = 0;
    while (i < n && A[i] != x)
        i++;
    if (i == n)
        return -1;
    else
        return i;
```

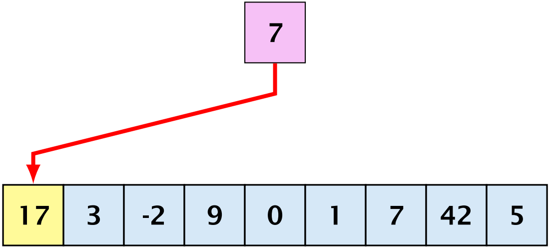
Naives Suchen

Beispiel

7

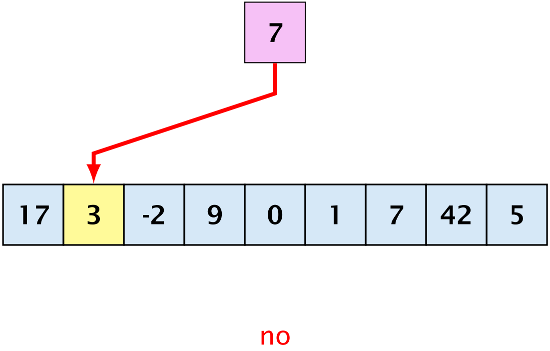
17	3	-2	9	0	1	7	42	5

Beispiel

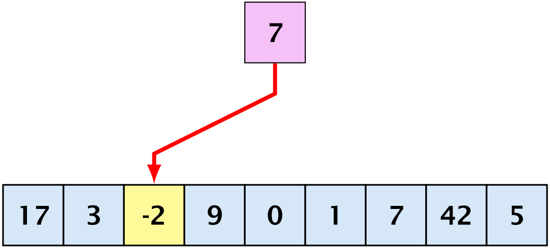


no

Beispiel

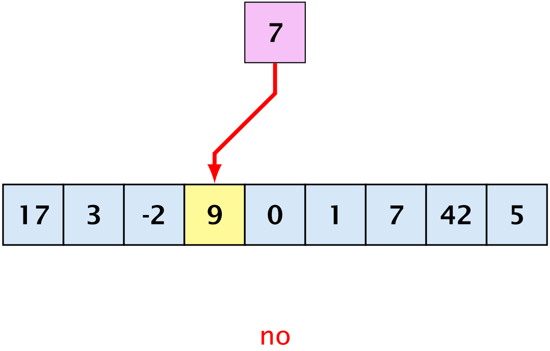


Beispiel

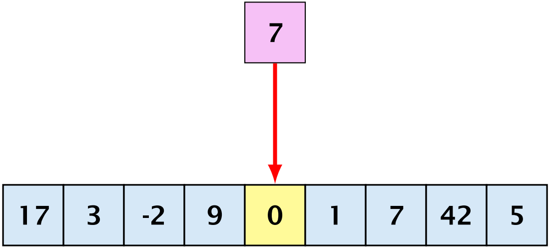


no

Beispiel

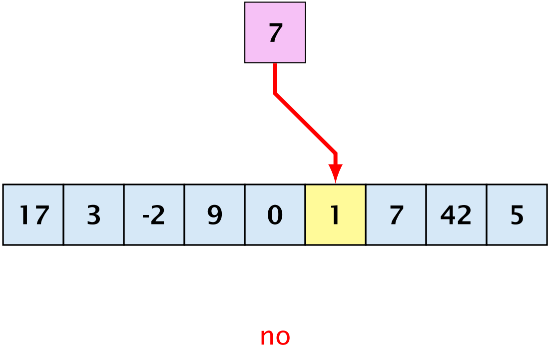


Beispiel

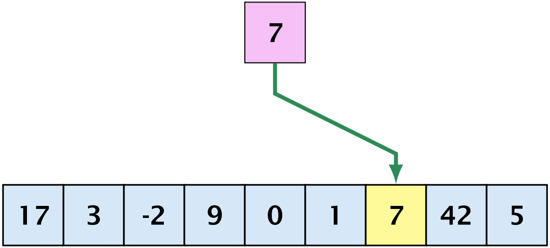


no

Beispiel



Beispiel



yes

Laufzeit Naives Suchen

Best-case:

Wenn x an Position 0.

$\Longrightarrow$ Laufzeit: $\mathcal{O}(1)$.

Worst-case:

Wenn x nicht vorkommt.

$\Longrightarrow$ 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.

$\Longrightarrow$ binäre Suche

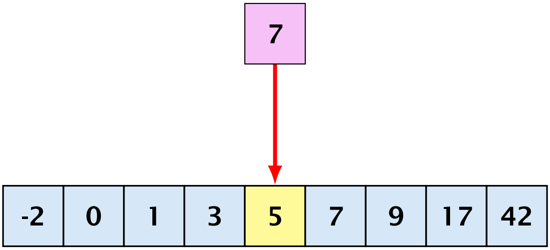
Beispiel

7

-2	0	1	3	5	7	9	17	42

- wir benötigen nur drei Vergleiche

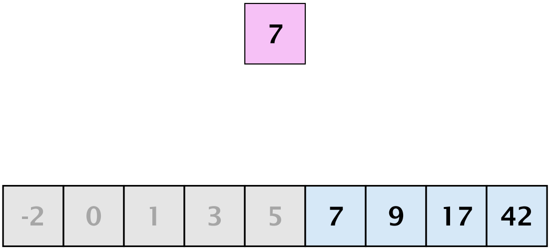
Beispiel



no

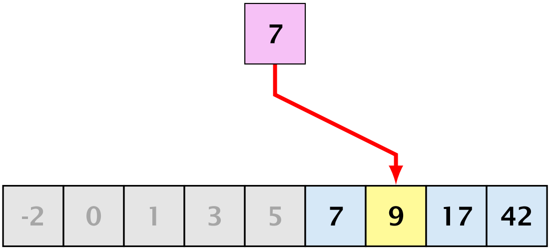
- wir benötigen nur drei Vergleiche

Beispiel



- wir benötigen nur drei Vergleiche

Beispiel



no

- wir benötigen nur drei Vergleiche

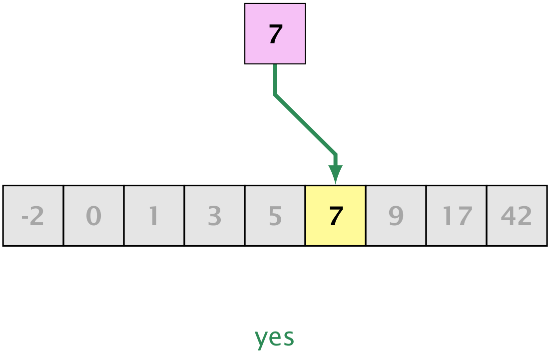
Beispiel

7

-2	0	1	3	5	7	9	17	42

- wir benötigen nur drei Vergleiche

Beispiel



- wir benötigen nur drei Vergleiche

Implementierung

```
Input:  sortiertes Array A; Element x;
        linker Index nl; rechter Index nr; nl<=nr
Output: Index i; nl ≤ i ≤ nr mit A[i]==x falls existent
        sonst -1
find(A, x, nl, nr) // Inputlaenge ist n=nr-nl+1
    t = (nl + nr) / 2;
    if (A[t] == x)
        return t;
    if (nl == nr)
        return -1;
    if (x > A[t])
        return find(A, x, t+1, nr);
    if (nl < t)
        return find(A, x, nl, t-1);
    return -1;
```

Laufzeit Binäre Suche

Laufzeit:

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

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.

Rekurrenzen

Wie finden wir einen geschlossenen Ausdruck für die Laufzeit?

Dafür müssen wir die Rekursionsgleichung lösen.

Rekurrenzen

Wie finden wir einen geschlossenen Ausdruck für die Laufzeit?

Dafür müssen wir die Rekursionsgleichung lösen.

Methoden

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.

Raten+Induktion

Zuerst müssen wir die $\mathcal{O}$-Notation entfernen:

$$T(n) \leq \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + c_1 n & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

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:

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 n & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

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

$$T(n) \le \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \le a \log n + b$.

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)

Raten+Induktion

$$T(n) \le \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \le a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$:

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 $b \geq c_2$.

Raten+Induktion

$$T(n) \le \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \le a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \ge c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

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** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für n:

Raten+Induktion

$$T(n) \leq \begin{cases} T\left(\frac{n}{2}\right) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \leq a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für n:

$$T(n) \leq T\left(\frac{n}{2}\right) + c_1$$

Raten+Induktion

$$T(n) \le \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \le a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \ge c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für n:

$$\begin{aligned} T(n) &\le T\left(\frac{n}{2}\right) + c_1 \\ &\le \left(a \log \frac{n}{2} + b\right) + c_1 \end{aligned}$$

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** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, 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** $1, \dots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \dots, n-1\}$, und $n > 1$. Wir beweisen sie für n:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + c_1 \\ &\leq \left(a \log \frac{n}{2} + b\right) + c_1 \\ &= a(\log n - 1) + b + c_1 \\ &= a \log n + (c_1 - a) + b \end{aligned}$$

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** $1, \dots, n-1 \to n$:

  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

$$T(n) \le \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \le a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \ge c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für n:

$$\begin{aligned} T(n) &\le T\left(\frac{n}{2}\right) + c_1 \\ &\le \left(a \log \frac{n}{2} + b\right) + c_1 \\ &= a(\log n - 1) + b + c_1 \\ &= a \log n + (c_1 - a) + b \\ &\le a \log n + b \end{aligned}$$

  Gilt falls $a \ge c_1$.

Mastertheorem

Lemma 3

Seien $a \geq 1, b \geq 1$ *und* $\epsilon > 0$ *Konstanten. Betrachte die Rekurrenz*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \ .$$

1. Fall:

Falls $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$ *gilt* $T(n) = \Theta(n^{\log_b a})$.

2. Fall:

Falls $f(n) = \Theta(n^{\log_b(a)} \log^k n)$ *gilt* $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$, $k \geq 0$.

Mastertheorem

Wir beweisen das Mastertheorem für den Fall $n = b^\ell$, und nehmen an, dass der nichtrekursive Fall für Problemgröße 1 Kosten 1 verursacht.

Der Rekursionsbaum

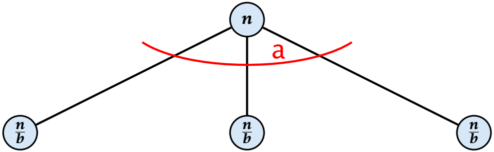
Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:

Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:

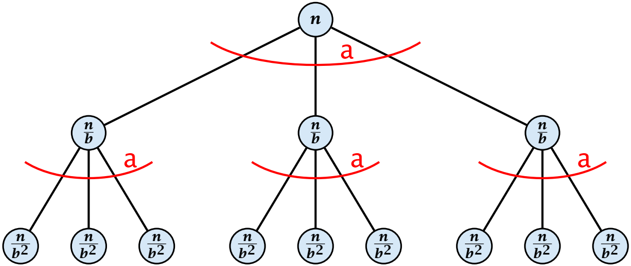
Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



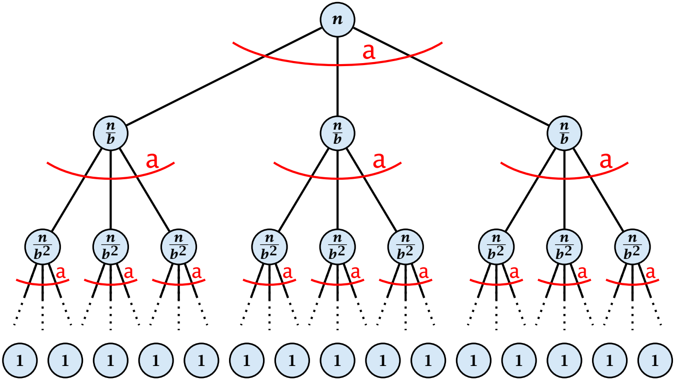
Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



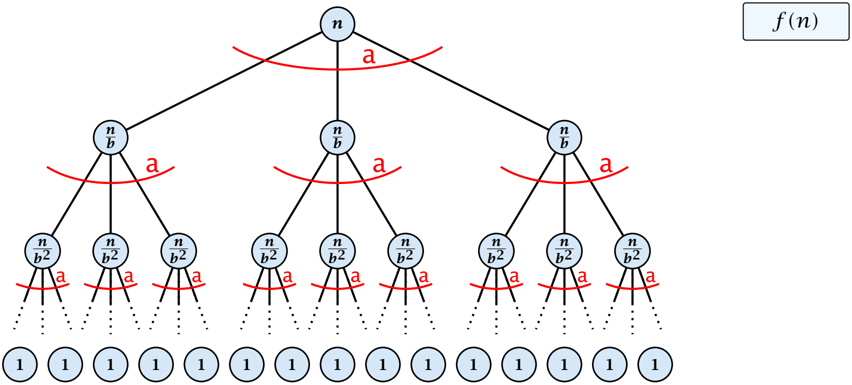
Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



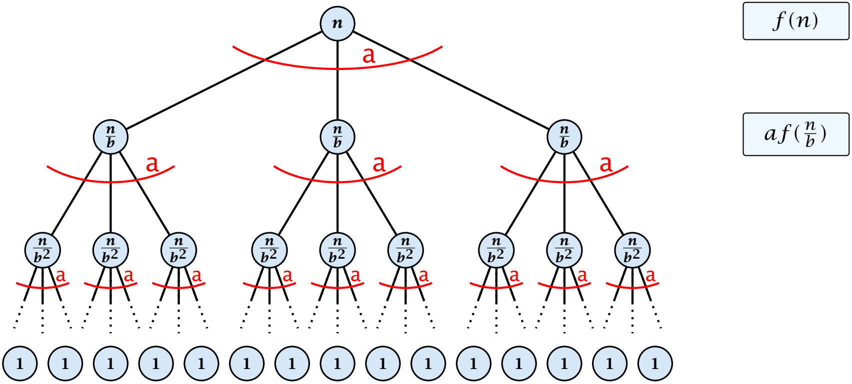
Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



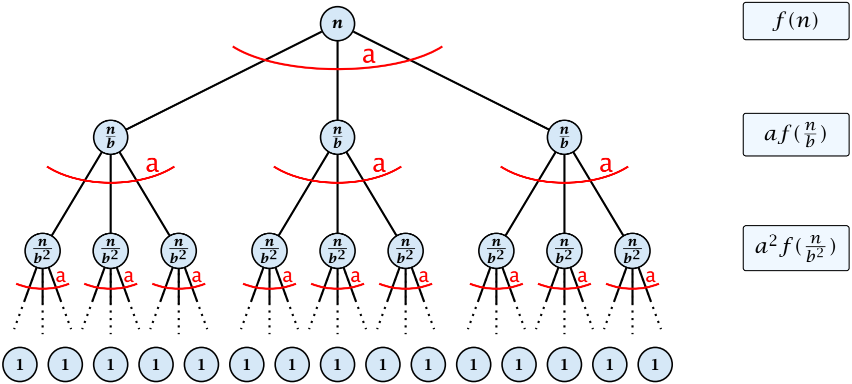
Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



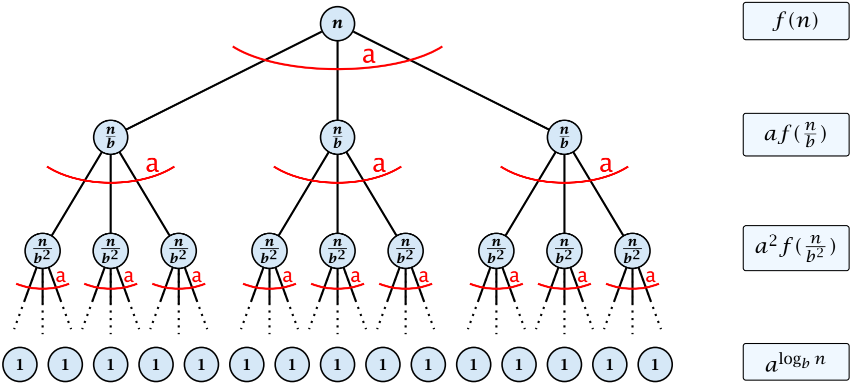
Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



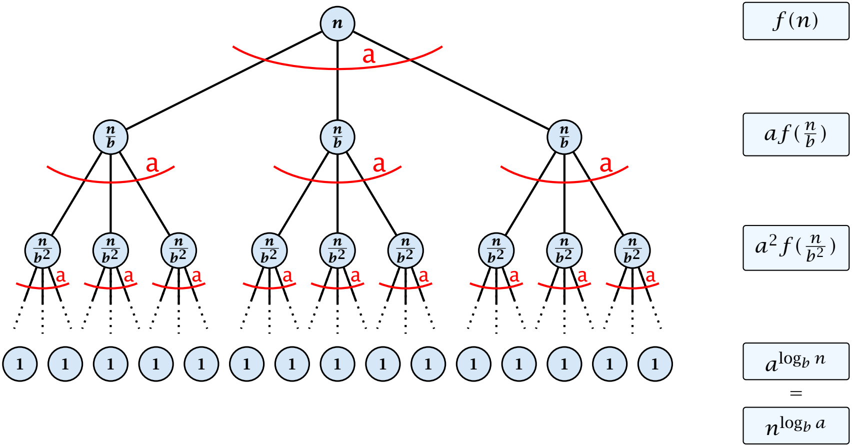
Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



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 $f(n) \leq cn^{\log_b a - \epsilon}$.

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 $f(n) \leq cn^{\log_b a-\epsilon}$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon} \end{aligned}$$

$$\boxed{b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}}$$

Fall 1. Sei $f(n) \leq cn^{\log_b a-\epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon} \\
&= cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} \left(b^{\epsilon}\right)^i
\end{aligned}
$$

$$\boxed{b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}}$$

Fall 1. Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \\
&= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} \left(b^{\epsilon}\right)^i
\end{aligned}
$$

$\boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}}$

$\boxed{\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}}$

Fall 1. Sei $f(n) \le cn^{\log_b a - \epsilon}$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\le c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \\ \boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}} \quad &= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} \left(b^{\epsilon}\right)^i \\ \boxed{\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}} \quad &= cn^{\log_b a - \epsilon}(b^{\epsilon \log_b n} - 1)/(b^{\epsilon} - 1) \end{aligned}$$

Fall 1. Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon}$$

$\boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}}$

$$= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} (b^{\epsilon})^i$$

$\boxed{\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}}$

$$= cn^{\log_b a - \epsilon}(b^{\epsilon \log_b n} - 1)/(b^{\epsilon} - 1)$$

$$= cn^{\log_b a - \epsilon}(n^{\epsilon} - 1)/(b^{\epsilon} - 1)$$

Fall 1. Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \\
\boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}} \quad &= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} (b^{\epsilon})^i \\
\boxed{\textstyle\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}} \quad &= cn^{\log_b a - \epsilon}(b^{\epsilon \log_b n} - 1)/(b^{\epsilon} - 1) \\
&= cn^{\log_b a - \epsilon}(n^{\epsilon} - 1)/(b^{\epsilon} - 1) \\
&= \frac{c}{b^{\epsilon} - 1} n^{\log_b a}(n^{\epsilon} - 1)/(n^{\epsilon})
\end{aligned}
$$

Fall 1. Sei $f(n) \leq cn^{\log_b a-\epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon} \\
\boxed{b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}} \quad &= cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} (b^\epsilon)^i \\
\boxed{\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}} \quad &= cn^{\log_b a-\epsilon}(b^{\epsilon \log_b n} - 1)/(b^\epsilon - 1) \\
&= cn^{\log_b a-\epsilon}(n^\epsilon - 1)/(b^\epsilon - 1) \\
&= \frac{c}{b^\epsilon - 1} n^{\log_b a}(n^\epsilon - 1)/(n^\epsilon)
\end{aligned}
$$

Also,

$$
T(n) \leq \left(\frac{c}{b^\epsilon - 1} + 1\right) n^{\log_b (a)}
$$

Fall 1. Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \\
\boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}} \quad &= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} \left(b^{\epsilon}\right)^i \\
\boxed{\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}} \quad &= cn^{\log_b a - \epsilon}(b^{\epsilon \log_b n} - 1)/(b^{\epsilon} - 1) \\
&= cn^{\log_b a - \epsilon}(n^{\epsilon} - 1)/(b^{\epsilon} - 1) \\
&= \frac{c}{b^{\epsilon} - 1} n^{\log_b a}(n^{\epsilon} - 1)/(n^{\epsilon})
\end{aligned}
$$

Also,

$$
T(n) \leq \left(\frac{c}{b^{\epsilon} - 1} + 1\right) n^{\log_b (a)} \qquad \boxed{\Rightarrow T(n) = \mathcal{O}(n^{\log_b a}).}
$$

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 $f(n) \leq cn^{\log_b a}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a}$$

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 $f(n) \le cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\le c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n-1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Fall 2. Sei $f(n) \leq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Also,

$$
T(n) = \mathcal{O}(n^{\log_b a} \log_b n)
$$

Fall 2. Sei $f(n) \le cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\le c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Also,

$$T(n) = \mathcal{O}(n^{\log_b a} \log_b n) \qquad \boxed{\Rightarrow T(n) = \mathcal{O}(n^{\log_b a} \log n).}$$

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 $f(n) \geq c n^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a}
\end{aligned}
$$

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,

$$
T(n) = \Omega(n^{\log_b a} \log_b n)
$$

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,

$$T(n) = \Omega(n^{\log_b a} \log_b n) \qquad \boxed{\Rightarrow T(n) = \Omega(n^{\log_b a} \log n).}$$

Fall 2. Sei $f(n) \leq c n^{\log_b a} (\log_b(n))^k$.

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a}$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$
$$\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k$$

$$\boxed{n = b^\ell \Rightarrow \ell = \log_b n}$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \\
\boxed{n = b^\ell \Rightarrow \ell = \log_b n} \quad &= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k
\end{aligned}
$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \\ &= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\ &= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k \end{aligned}$$

$\boxed{n = b^\ell \Rightarrow \ell = \log_b n}$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k$$

$\boxed{n = b^\ell \Rightarrow \ell = \log_b n}$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k$$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k$$

$$= cn^{\log_b a} \sum_{i=1}^{\ell} i^k$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \le cn^{\log_b a}(\log_b(n))^k$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\le c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k \\
&= cn^{\log_b a} \boxed{\sum_{i=1}^{\ell} i^k} \approx \frac{1}{k}\ell^{k+1}
\end{aligned}
$$

$\boxed{n = b^\ell \Rightarrow \ell = \log_b n}$

Beachte, dass $\sum_{i=1}^{\ell} i^k \le \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \ge \sum_{i=\ell/2}^{\ell} i^k \ge (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k \\
&= cn^{\log_b a} \sum_{i=1}^{\ell} i^k \\
&\approx \frac{c}{k} n^{\log_b a} \ell^{k+1}
\end{aligned}
$$

$\boxed{n = b^\ell \Rightarrow \ell = \log_b n}$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde. Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k \\
&= cn^{\log_b a} \sum_{i=1}^{\ell} i^k \\
&\approx \frac{c}{k} n^{\log_b a} \ell^{k+1}
\end{aligned}
$$

$\boxed{n = b^\ell \Rightarrow \ell = \log_b n}$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde. Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

$\boxed{\Rightarrow T(n) = \mathcal{O}(n^{\log_b a} \log^{k+1} n).}$

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

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:

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:

$$\begin{array}{ccccccccccc} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\ \hline \end{array}$$

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

$$
\begin{array}{ccccccccc|c}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & \boxed{1} & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & \boxed{1} & B \\
\hline
\end{array}
$$

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:

$$\begin{array}{ccccccccc|c|c} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1_{1} & 1 & & B \\ \hline & & & & & & & & 0 & & \end{array}$$

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:

$$
\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1_{1} & 1 & & B \\
\hline
 & & & & & & & & 0 & &
\end{array}
$$

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:

$$
\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
 & & & & & & {}_{1} & {}_{1} & & & \\
\hline
 & & & & & & & 0 & 0 & &
\end{array}
$$

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:

$$
\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
 & & & & & & {}_1 & {}_1 & & & \\
\hline
 & & & & & & & 0 & 0 & &
\end{array}
$$

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:

$$
\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & \boxed{1} & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & \boxed{0} & 1 & 1 & & B \\
 & & & & & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & \\
\hline
 & & & & & & \boxed{0} & 0 & 0 & &
\end{array}
$$

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:

$$
\begin{array}{ccccc|c|cccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & \quad A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & \quad B \\
 & & & & & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
\hline
 & & & & & & 0 & 0 & 0 &
\end{array}
$$

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:

$$
\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
 & & & & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & \\
\hline
 & & & & & 1 & 0 & 0 & 0 & &
\end{array}
$$

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:

$$\begin{array}{cccc|c|cccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
 & & & & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & \\
\hline
 & & & & & 1 & 0 & 0 & 0 & &
\end{array}$$

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:

$$
\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
 & & & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & \\
\hline
 & & & & 0 & 1 & 0 & 0 & 0 & &
\end{array}
$$

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:

$$\begin{array}{ccccccccccc}
1 & 1 & 0 & \boxed{1} & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
 & & & _{1} & _{0} & _{1} & _{1} & _{1} & & & \\
\hline
 & & & & 0 & 1 & 0 & 0 & 0 & &
\end{array}$$

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:

$$
\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
 & & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & \\
\hline
 & & & 0 & 0 & 1 & 0 & 0 & 0 & &
\end{array}
$$

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:

$$
\begin{array}{ccccccccccc}
1 & 1 & \boxed{0} & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
 & & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & \\
\hline
 & & & 0 & 0 & 1 & 0 & 0 & 0 & &
\end{array}
$$

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:

$$
\begin{array}{ccccccccc|c}
1 & 1 & \boxed{0} & 1 & 1 & 0 & 1 & 0 & 1 & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\
 & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
\hline
 & & \boxed{1} & 0 & 0 & 1 & 0 & 0 & 0 &
\end{array}
$$

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:

$$
\begin{array}{ccccccccc|c}
1 & \boxed{1} & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\
 & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
\hline
 & & 1 & 0 & 0 & 1 & 0 & 0 & 0 &
\end{array}
$$

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:

$$
\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
{\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & \\
\hline
 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & &
\end{array}
$$

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
0	0	1	1	0	1	1	1		
	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:

$$
\begin{array}{cccccccccccc}
 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
{\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & \\
\hline
 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & &
\end{array}
$$

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

$$
\begin{array}{c|ccccccccc|c}
 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\
 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\
{\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
\hline
 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 &
\end{array}
$$

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

$$
\begin{array}{cccccccccccc}
 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
{\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & \\
\hline
\boxed{1} & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & &
\end{array}
$$

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

$$
\begin{array}{cccccccccccc}
 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
 {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & \\
\hline
1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & &
\end{array}
$$

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.

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$\begin{array}{c} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \end{array}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$\begin{array}{ccccccccccc} 1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & \boxed{1} \\ \hline \end{array}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{ccccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & \boxed{1} \\
\hline
 & & & & & & 1 & 0 & 0 & 0 & 1
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{ccccccccccc}
 & & & & 1 & 0 & 0 & 0 & 1 & \times & 1\;0\;\boxed{1}\;1 \\
\hline
 & & & & & & 1 & 0 & 0 & 0 & 1
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & \boxed{1} & 1 \\
\hline
 & & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & & & & & & & 0
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & \boxed{1} & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0
\end{array}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & \boxed{0} & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{cccccccccccc}
 & 1 & 0 & 0 & 0 & 1 & \times & 1 & \boxed{0} & 1 & 1 \\
\hline
 & & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & & & & & & & 0 & 0
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{ccccccccccc}
 & & 1 & 0 & 0 & 0 & 1 & \times & 1 & \boxed{0} & 1 & 1 \\
\hline
 & & & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{ccccccccc}
1 & 0 & 0 & 0 & 1 & \times & \boxed{1} & 0 & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{rrrrrrrrrr}
 & & 1 & 0 & 0 & 0 & 1 & \times & \boxed{1} & 0 & 1 & 1 \\
\hline
 & & & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & 0 & 0 & 0
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{r}
1\ 0\ 0\ 0\ 1 \times \boxed{1}\ 0\ 1\ 1 \\
\hline
1\ 0\ 0\ 0\ 1 \\
1\ 0\ 0\ 0\ 1\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0 \\
1\ 0\ 0\ 0\ 1\ 0\ 0\ 0
\end{array}
$$

- Dies nennt man auch die „Schulmethode" für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{rrrrrrrrrr}
 & & 1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{r}
10001 \times 1011 \\
\hline
10001 \\
100010 \\
0000000 \\
10001000 \\
\hline
10111011
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \le n$) multiplizieren.

$$\begin{array}{r} 1\;0\;0\;0\;1 \times 1\;0\;1\;1 \\ \hline 1\;0\;0\;0\;1 \\ 1\;0\;0\;0\;1\;0 \\ 0\;0\;0\;0\;0\;0\;0 \\ 1\;0\;0\;0\;1\;0\;0\;0 \\ \hline 1\;0\;1\;1\;1\;0\;1\;1 \end{array}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \le 2n$ bits.

Laufzeit:

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{r}
10001 \times 1011 \\
\hline
10001 \\
100010 \\
0000000 \\
10001000 \\
\hline
10111011
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

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.

$$
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
 & & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebnisse haben höchstens $m + n \leq 2n$ bits.

Laufzeit:

- Zwischenergebnisse berechnen: $\mathcal{O}(nm)$.
- Addieren von m Zahlen der Länge $\leq 2n$: $\mathcal{O}((m + n)m) = \mathcal{O}(nm)$.

Beispiel: Integermultiplikation

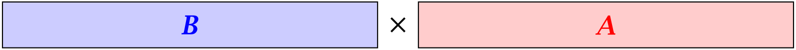
Ein rekursiver Ansatz:

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 $n = 2^k$.

$$\boxed{b_{n-1} \quad \cdots \quad b_0} \times \boxed{a_{n-1} \quad \cdots \quad a_0}$$

Beispiel: Integermultiplikation

Ein rekursiver Ansatz:

Angenommen die Integer A und B haben Länge $n = 2^k$.

$$\boxed{b_{n-1} \quad \cdots \quad b_{\frac{n}{2}} \; b_{\frac{n}{2}-1} \quad \cdots \quad b_0} \times \boxed{a_{n-1} \quad \cdots \quad a_{\frac{n}{2}} \; a_{\frac{n}{2}-1} \quad \cdots \quad a_0}$$

Beispiel: Integermultiplikation

Ein rekursiver Ansatz:

Angenommen die Integer A und B haben Länge $n = 2^k$.

B_1	B_0	$\times$	A_1	A_0

Beispiel: Integermultiplikation

Ein rekursiver Ansatz:

Angenommen die Integer A und B haben Länge $n = 2^k$.

B_1	B_0	$\times$	A_1	A_0

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 $n = 2^k$.

$$\underbrace{B_1}\ \underbrace{B_0} \times A_1\ A_0$$

$A_1 (2^{\frac{n}{2}})^1$ $\quad$ $A_0 \cdot (2^{\frac{n}{2}})^0$

Dann gilt

$$A = A_1 \cdot 2^{\frac{n}{2}} + A_0 \text{ und } B = \boxed{B_1 \cdot 2^{\frac{n}{2}} + B_0}$$

Also,

$\Theta(n)$

$$\underline{A \cdot B} = A_1 B_1 \cdot 2^n + (A_1 B_0 + A_0 B_1) \cdot 2^{\frac{n}{2}} + A_0 B_0$$

$\Theta(n)$ $\quad$ $O(n)$ $\quad$ $\Theta(n)$

$T(n) = 4 \cdot T(\frac{n}{2}) + \Theta(n)$ $\quad$ $\Theta(n)$

Beispiel: Integermultiplikation

```
Input:  Zahlen A, B, repraesentiert durch Bitarrays
        der Laenge n
Output: Bitarray, das A*B enthaelt

mult(A, B, n)
    if (n == 1)
        return new int(A[0]*B[0]);
    split(A,A0,A1);
    split(B,B0,B1);
    Z2 = mult(A1,B1,n/2);
    Z1 = mult(A0,B1,n/2) + mult(A1,B0,n/2);
    Z0 = mult(A0,B0,n/2);
    return Z2*2^n + Z1*2^{n/2} + Z0;
```

Die Addition + addiert Bitarrays. Das funktioniert in C++ nicht direkt, aber man kann stattdessen z.B. eine Funktion add(A,B) benutzen. Wenn man den Algorithmus nach C++ übertragen möchte ist das Speichermanagement etwas unübersichtlich, da man jedes Zwischenergebnis vor Funktionsende wieder von Hand löschen muss.

Wir erhalten folgende Rekurrenzgleichung:

$$T(n) = 4T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

Beispiel: Integermultiplikation

$4 \quad \to 2 \quad \to \Theta(n)$

Mastertheorem: Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Fall 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Fall 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

$\Theta(n) \subseteq \Theta(n^{\log_2 4 - \varepsilon}) = \Theta(n^{2-\varepsilon})$

$T(n) = \Theta(n^2)$

Beispiel: Integermultiplikation

Mastertheorem: Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Fall 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\qquad T(n) = \Theta(n^{\log_b a})$
- Fall 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\qquad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

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: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\qquad T(n) = \Theta(n^{\log_b a})$
- Fall 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\qquad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

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: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\qquad T(n) = \Theta(n^{\log_b a})$
- Fall 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\qquad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

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 $\mathcal{O}(n^2)$.

$\Longrightarrow$ 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:

$$
\begin{aligned}
Z_1 &= \boxed{A_1B_0 + A_0B_1} \\
&= (A_0 + A_1) \cdot (B_0 + B_1) - \underbrace{A_1B_1}_{= Z_2} - \underbrace{A_0B_0}_{= Z_0}
\end{aligned}
$$

$2 \cdot T(\frac{n}{2}) + n$

$T(\frac{n}{2}) + \frac{n}{2} + \frac{n}{2} + n + n$

$T(\frac{n}{2}) + 3n$

Beispiel: Integermultiplikation

We can use the following identity to compute Z_1:

$$
\begin{aligned}
Z_1 &= A_1B_0 + A_0B_1 \\
&= (A_0 + A_1)\cdot(B_0 + B_1) - \overbrace{A_1B_1}^{=\,Z_2} - \overbrace{A_0B_0}^{=\,Z_0}
\end{aligned}
$$

```
Input:  Zahlen A, B, repraesentiert durch Bitarrays
        der Laenge n
Output: Bitarray, das A*B enthaelt

mult(A, B, n)
    if (n == 1)
        return new int(A[0]*B[0]);
    split(A,A0,A1); Θ(n)
    split(B,B0,B1); Θ(n)
    Z2 = mult(A1,B1,n/2); T(n/2)
    Z1 = mult(A0+A1,B0+B1,n/2)-Z0-Z2; T(n/2+1) + Θ(n)
    Z0 = mult(A0,B0,n/2); T(n/2)
    return Z2*2^n + Z1*2^(n/2) + Z0;
```

$\Theta(n)$

Beispiel: Integermultiplikation

$(X_{\frac{n}{2}} \cdot 2^{n/2} + \bar{X})(Y_{\frac{n}{2}} \cdot 2^{n/2} + \bar{Y})$

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 $\bar{X}$ zerlegt kann man folgendes nutzen:

$T(\frac{n}{2}) + \Theta(n)$

```
...
 X = A0 + A1;
 Y = B0 + B1;
Z1 = X_{n/2}*Y_{n/2}*2^n + (X_{n/2}*Ȳ + Y_{n/2}*X̄)*2^{n/2} + mult(X̄,Ȳ) - Z0 - Z2;
...
```

Laufzeit hierfür ist $T(n/2) + \mathcal{O}(n)$.

$1 + n/2$ $1 + n/2$

$X_{\frac{n}{2}}$ $\bar{X}$ Y

Beispiel: Integermultiplikation

Wir erhalten folgende Rekurrenz:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

Master Theorem: Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

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: $T[n] = \overset{3}{a}\,T(\tfrac{n}{\underset{2}{b}}) + \underset{}{f(n)} \overset{}{=} \Theta(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \boxed{\Theta(n^{\log_b a})}$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

$$\Theta(n) = \underbrace{\Theta(n^{\log_2 3 - \varepsilon})}_{=\;\Theta(n^{\alpha})} \qquad \alpha > 1$$

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: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a-\epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a}\log^k n)$ $\quad T(n) = \Theta(n^{\log_b a}\log^{k+1} n)$

Wir sind im Fall 1. Laufzeit $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$.

Eine deutliche Verbesserung der „Schulmethode“.

$f(n) = \mathcal{O}(\log n^{\log_b a - \varepsilon})$

Beispiel: Integermultiplikation

Wir erhalten folgende Rekurrenz:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

Master Theorem: Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

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 **kreative** Tätigkeit

- “The Art of Computer Programming” (D. Knuth)

Unterstützung durch Algorithmenmuster

- 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

```
Input: Aufgabe A

DivideAndConquer(A)
    if (A klein)
        loese A explizit;
    else
        teile A in Teilaufgaben A1,...,An;
        DivideAndConquer(A1)
        ...
        DivideAndConquer(An)
        berechne Loesung fuer A aus Lsgn fuer A1,...,An
```

$\sum_{i=1}^{n} T(|A_i|)$

Divide and Conquer

- Berechnung der Fibonacii-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
- ...

Mergesort – Sortieren durch Mischen

Mergesort ist ein schneller Sortieralgorithmus der nach dem Divide and Conquer Prinzip arbeitet

John von Neumann (1945)

Divide and Conquer: MergeSort

$\overset{a}{2} \cdot T(\underset{b}{\tfrac{n}{2}}) + \boxed{\Theta(n)}$

$f(n)$

Sei L verkette Liste mit n natürlichen Zahlen $a_i \in \mathbb{N}$.

Aufgabe: sortiere L in aufsteigender Reihenfolge. $f(n) = \Theta(n^{\log_2 2})$ $-\varepsilon$

Lösung mit Divide and Conquer-Muster: **MergeSort** f

Idee:

- **Divide:** teile L auf in zwei gleich große Teillisten $\Theta(n)$
- **Rekursion:** rufe MergeSort rekursiv für die zwei Teillisten auf
- **Conquer:** setze die Teillisten zusammen (**merge** bzw. mischen) $\Theta(n)$ $=$ $\Theta(n)$

Wann ist Teilliste **"klein"**, d.h. wann löst man explizit?
→ Teilliste mit nur **einem** Element → sortiert!

$f(n) = \Theta(n)$ ✓

$f(n) = \Omega(n)$

$$f(n) \;=\; \Theta(n) \stackrel{?}{=} \Theta(n^{1-\varepsilon})$$

falsch

$1 = \Theta(n)$

$f(n) = \Omega(n \log^k n)$ für ein $k \geq 0$

$\| \; \| \, k=0$

$\Omega(n)$

2. Fall

$\Omega(n)$

$\Omega(n \cdot \log^{k+1} n)$

Divide and Conquer: MergeSort

```
List* mergeSort(List* a) {
    // returns a list with elements from a sorted
    // may delete the list pointed to by a
    if (a->length() <= 1) return a;
    List* b = a->half();
    a = mergeSort(a);
    b = mergeSort(b);
    return merge(a,b);
}
```

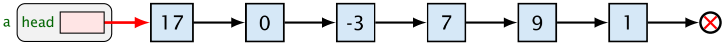
O(n

divide

Divide and Conquer: MergeSort

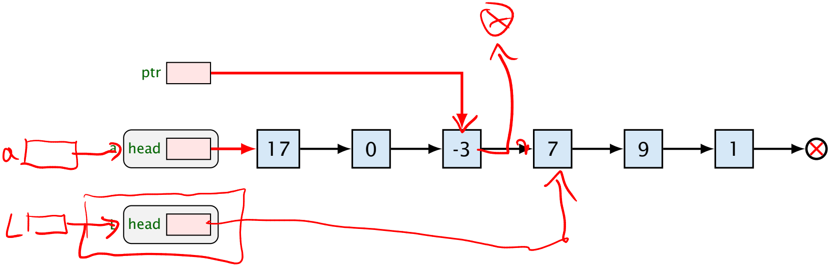
```
List* half() {
    // removes elements from second half of list
    // and returns these as a new list
    Node* ptr = head;   l = length()
    for (int i=0; i<length()/2-1; i++)
        ptr = ptr->next;
    List* L = new List(ptr->next);
    ptr->next = NULL;
    return L;
}
```

Beispiel – Halbieren

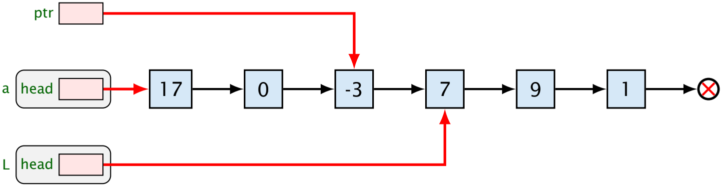


`a.half()`

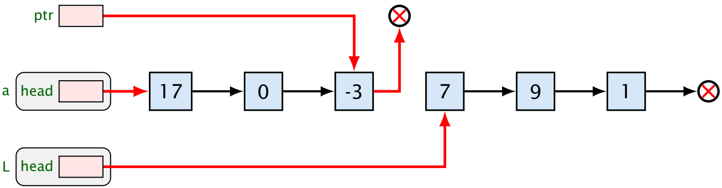
Beispiel – Halbieren



Beispiel – Halbieren

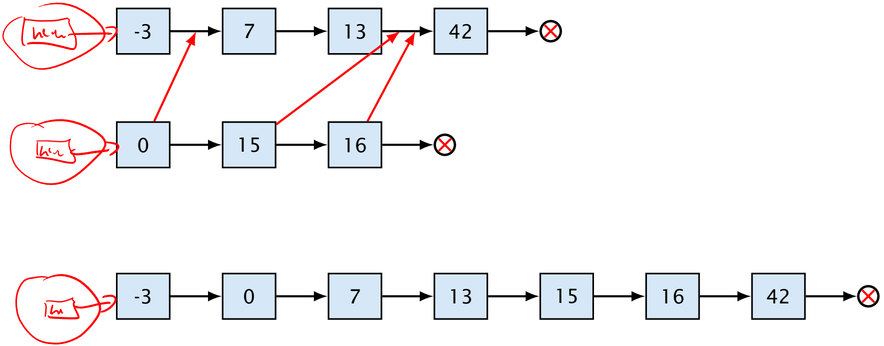


Beispiel – Halbieren

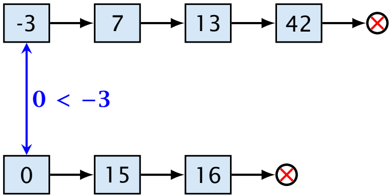


Beispiel – Mischen

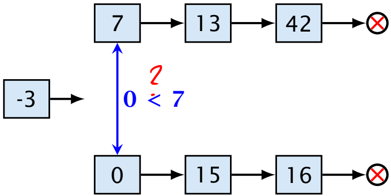
Hier benutzen wir das Symbol ⊗ für das `null`-Objekt.



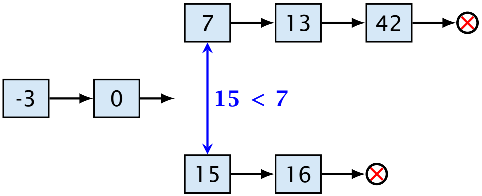
Beispiel – Mischen



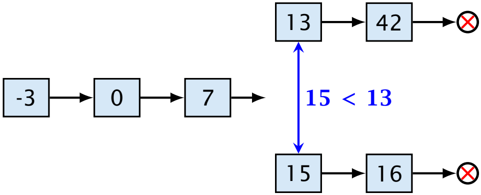
Beispiel – Mischen



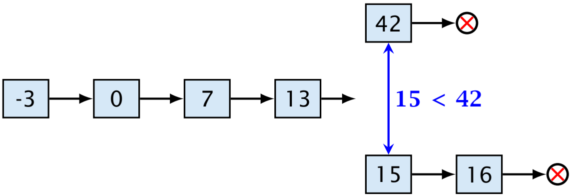
Beispiel – Mischen



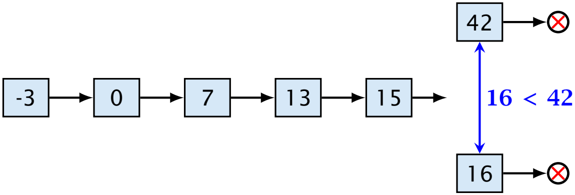
Beispiel – Mischen



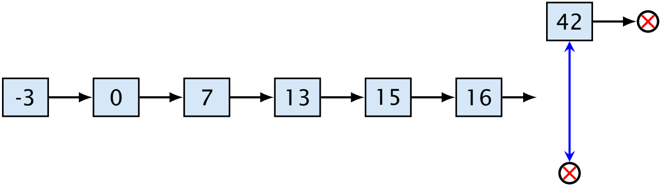
Beispiel – Mischen



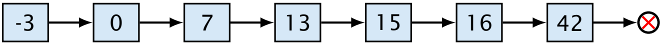
Beispiel – Mischen



Beispiel – Mischen



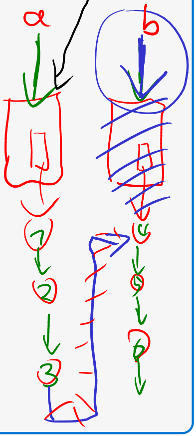
Beispiel – Mischen

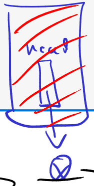


Divide and Conquer: MergeSort

```
List* merge(List* a, List* b) {
    // returns a list with elements from a and b
    // the lists a and b may be deleted
    if (a->empty()) { delete a; return b; }
    if (b->empty()) { delete b; return a; }
    List* h;
    if (a->elementAt(0) < b->elementAt(0))
        h = a;
    else
        h = b;
    // remove element from h and recurse
    int d = h->removeFront();
    List* L = merge(a,b);
    L->insertFront(d);
    return L;
}
```

h

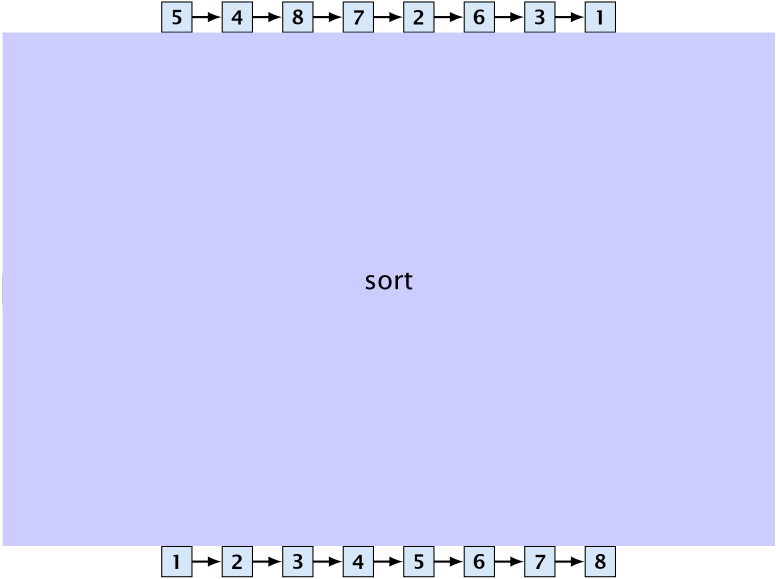




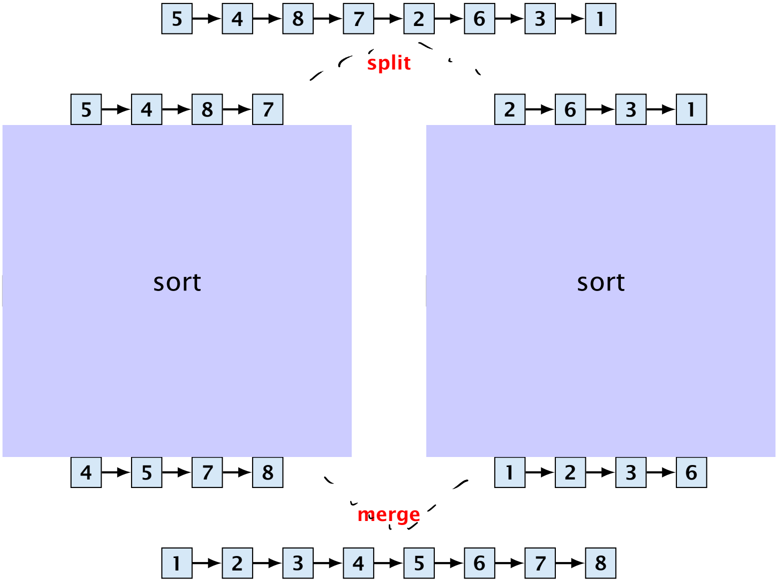
1 → 4 → 5

2 → 3 → 7 → → → →

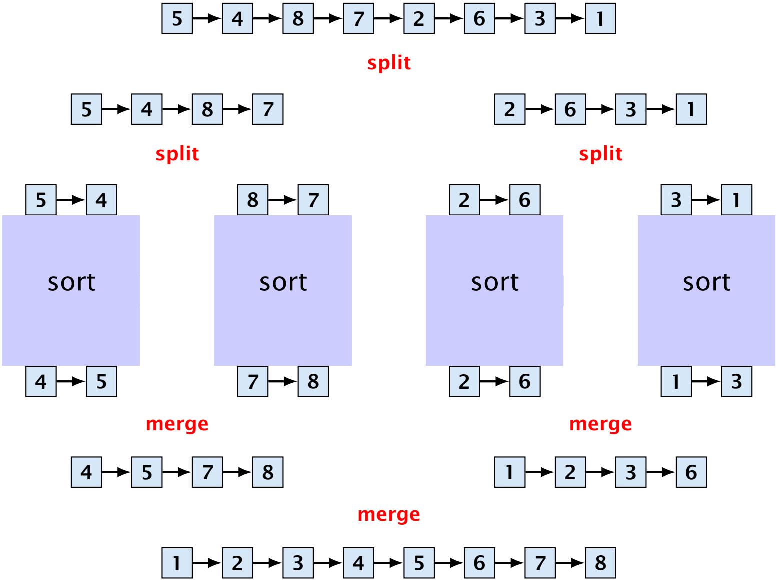
Mergesort



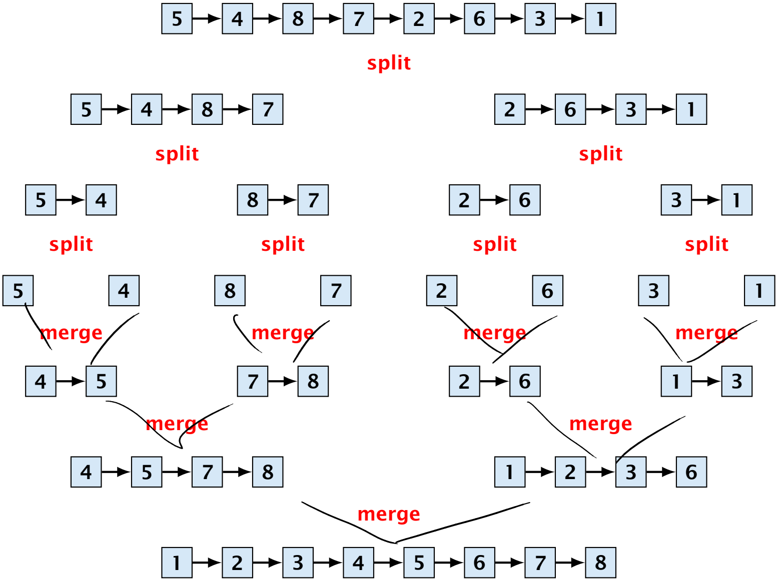
Mergesort


5 4 8 7 2 6 3 1
split
5 4 8 7
2 6 3 1
sort
sort
4 5 7 8
1 2 3 6
merge
1 2 3 4 5 6 7 8

Mergesort


5 4 8 7 2 6 3 1
split
5 4 8 7
2 6 3 1
split
split
5 4
8 7
2 6
3 1
sort
sort
sort
sort
4 5
7 8
2 6
1 3
merge
merge
4 5 7 8
1 2 3 6
merge
1 2 3 4 5 6 7 8

Mergesort



5 4 8 7 2 6 3 1
split
5 4 8 7
2 6 3 1
split
split
5 4
8 7
2 6
3 1
split
split
split
split
5
4
8
7
2
6
3
1
merge
merge
merge
merge
4 5
7 8
2 6
1 3
merge
merge
4 5 7 8
1 2 3 6
merge
1 2 3 4 5 6 7 8

MergeSort

Eigenschaften

- Merge Sort benötigt zusätzlichen Speicher in Funktion `merge`; insgesamt n zusätzliche Elemente falls A Länge n
- best und worst case sind identisch
- die meiste Arbeit steckt in **merge**; ein Aufruf von merge auf zwei Listen der Länge $n/2$ Kosten Zeit $\mathcal{O}(n)$.

MergeSort

Rekurrenzgleichung:

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n) \quad .$$

Laufzeit: $\mathcal{O}(n \log n)$

Sortieralgorithmen Zusammenfassung

Insertion Sort

- in-place
- Komplexität $\mathcal{O}(n^2)$, best case: $\mathcal{O}(n)$
- Komplexität $\mathcal{O}(hn)$ falls jedes Element nur h Positionen von Zielposition entfernt

MergeSort

- benötigt zusätzlichen Speicher
- Komplexität $\mathcal{O}(n \log n)$

QuickSort

- in-place
- Komplexität im Mittel $\mathcal{O}(n \log n)$, worst case: $\mathcal{O}(n^2)$

Algorithmenmuster: Greedy

greedy = “gierig”, “gefräßig”

Greedy Prinzip:

- Lösung eines Problems durch schrittweise Erweiterung der Lösung ausgehend von Startlösung
- in jedem Schritt wähle den bestmöglichen Schritt (ohne Berücksichtigung zukünftiger Schritte) ⇒ **greedy**

gefundene Lösung muss nicht immer optimal sein!

Algorithmenmuster: Greedy

```
Input: Aufgabe A

Greedy(A)
    S = {}; // Loesung
    while (S keine Loesung)
        waehle bestmoeglichen Erweiterungschritt s
        erweitere S mit s
```

Greedy: Beispiel Wechselgeld I

Problem: Herausgabe von Wechselgeld

- **Voraussetzung:** übliche Euro-Münzen 2€, 1€, 50ct, 20ct, 10ct, 5ct, 2ct und 1ct
- **Aufgabe:** Wechselgeld-Herausgabe mit möglichst wenig Münzen

Beispiel: Preis €1.11, bezahlt mit 2€-Münze. Wechselgeld: $\boxed{\text{89ct}}$

Minimum Anzahl Münzen: 6

$$89\text{ct} = 50\text{ct} + 20\text{ct} + 10\text{ct} + 5\text{ct} + 2\text{ct} + 2\text{ct}$$

Greedy: Beispiel Wechselgeld II

```
Input: Betrag b

Wechselgeld(b)
    printf("%d = ",b);
    count = 0;
    while (count < b)
        // make greedy choice
        waehle groesste Muenze s mit count + s <= b
        printf("%d ",s);
        count += s;
```

Achtung: Abhängig vom Geldsystem liefert dieser Algorithmus nicht immer optimale Lösung!

- **Beispiel:** Münzen: 5ct, 4ct, 1ct. **Betrag:** 8ct
- **Greedy-Lösung:** 8ct = 5ct + 1ct + 1ct + 1ct
- **Optimale Lösung:** 8ct = 4ct + 4ct

Von Greedy lösbare Probleme

Voraussetzungen für Anwendbarkeit von Greedy:

- Lösungen lassen sich schrittweise durch Hinzufügen von Elementen aufbauen, beginnend bei leerer Lösung
- Bewertungsfunktion für partielle und vollständige Lösung
- Gesucht wird eine/die optimale Lösung

Anwendung Greedy: Glasfasernetz

Problemstellung: Aufbau von möglichst billigem Glasfasernetz zwischen n Knoten $K_1, \ldots, K_n$, so dass alle Knoten miteinander verbunden sind (u.U. mit Umweg)

Anwendung Greedy: Glasfasernetz

Problemstellung: Aufbau von möglichst billigem Glasfasernetz zwischen n Knoten $K_1, \ldots, K_n$, so dass alle Knoten miteinander verbunden sind (u.U. mit Umweg)

Input:

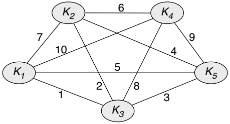
- Knoten $K_1, \ldots, K_n$
- Kosten $d_{ij} > 0$ für direkte Verbindung zwischen K_i und K_j für $i \neq j$, $i, j \in \{1, \ldots, n\}$

Anwendung Greedy: Glasfasernetz

Problemstellung: Aufbau von möglichst billigem Glasfasernetz zwischen n Knoten $K_1, \ldots, K_n$, so dass alle Knoten miteinander verbunden sind (u.U. mit Umweg)

Input:

- Knoten $K_1, \ldots, K_n$
- Kosten $d_{ij} > 0$ für direkte Verbindung zwischen K_i und K_j für $i \neq j$, $i, j \in \{1, \ldots, n\}$

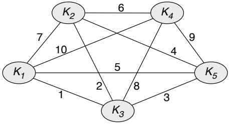


Anwendung Greedy: Glasfasernetz

Problemstellung: Aufbau von möglichst billigem Glasfasernetz zwischen n Knoten $K_1, \ldots, K_n$, so dass alle Knoten miteinander verbunden sind (u.U. mit Umweg)

Input:

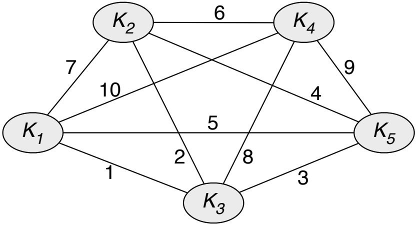
- Knoten $K_1, \ldots, K_n$
- Kosten $d_{ij} > 0$ für direkte Verbindung zwischen K_i und K_j für $i \neq j$, $i, j \in \{1, \ldots, n\}$



Output: Teilmenge aller Verbindungen, so dass

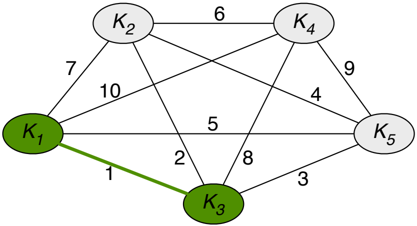
- alle Knoten verbunden sowie
- minimale Kosten

Glasfasernetz: Beispiel I



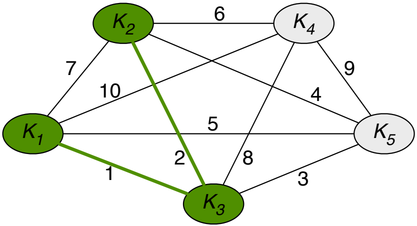
- Knoten $K_1, \ldots, K_5$
- Kosten d_{ij}
- repräsentiert als **gewichteter, ungerichteter Graph**

Glasfasernetz: Beispiel II



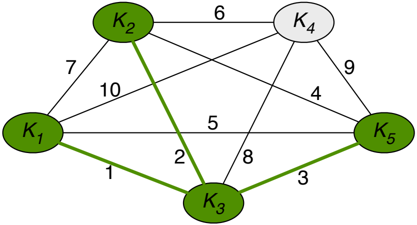
- Startknoten: K_1
- beste Verbindung: zu K_3, Kosten 1
  - andere Kosten: 7 (zu K_2), 10 (zu K_4), 5 (zu K_5)

Glasfasernetz: Beispiel III



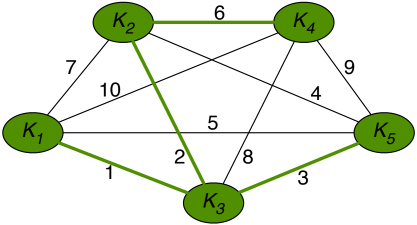
- nächstbeste Verbindung von $\{K_1, K_3\}$: zu K_2, Kosten 2
  - andere Kosten: 8 (zu K_4), 3 (zu K_5), 7 (zu K_2)

Glasfasernetz: Beispiel IV



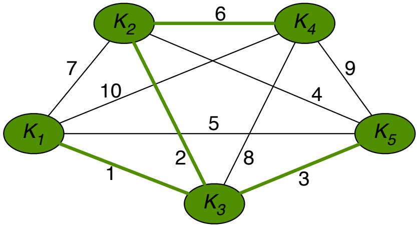
- nächstbeste Verbindung von $\{K_1, K_2, K_3\}$: zu K_5, Kosten 3
  - andere Kosten: 6 bzw. 8 (zu K_4)

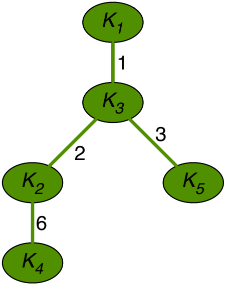
Glasfasernetz: Beispiel V



- nächstbeste Verbindung von $\{K_1, K_2, K_3, K_5\}$: zu K_4, Kosten 6
- $\Rightarrow$ alle Knoten behandelt, Algorithmus fertig

Glasfasernetz: Beispiel VI





Ergebnis: ein sog. minimaler Spannbaum (minimum spanning tree, MST) des Graphen

Glasfasernetz: Algorithmus

```
Input:  Array K von n Knoten, Kostenfunktion d(i,j)
Output: minimaler Spannbaum B

Glasfasernetz(K, d)
    B = K[1];
    while (B nicht Spannbaum)
        suche billigste Kante aus B;
        fuege Kante und Knoten zu B hinzu;
```

Komplexität naiver Implementation: $O(n^3)$

⇒ geht besser, (später in der Vorlesung)

Algorithmenmuster: Brute Force

Brute Force:

- erzeuge all in Frage kommenden Lösungskandidaten
- überprüfe für jeden Kandidaten ob es eine zulässige Lösung ist
- ggfs. (bei Optimierungsproblemen) bestimme zulässige Lösung mit minimalen Kosten/maximalem Profit

Eigenschaften:

- sehr einfach zu implementieren
- häufig sehr schlechte Laufzeit

Ein Programmierer sollte wissen ob man ein Problem via Brute-Force lösen kann, oder ob die Laufzeit dafür zu schlecht ist.

Algorithmenmuster: Brute Force

```
Input:  Problem P
Output: zulaessige Loesung fuer P

BruteForce(P)
    S = first(P)
    while (S != NULL)
        if (S valid for P)
            return S
        S = next(P)
```

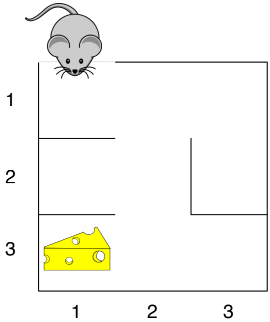
Algorithmenmuster: Backtracking

Backtracking: systematische Suchtechnik, um vorgegebenen Lösungsraum vollständig abzuarbeiten

Algorithmenmuster: Backtracking

Backtracking: systematische Suchtechnik, um vorgegebenen Lösungsraum vollständig abzuarbeiten

Paradebeispiel: Labyrinth. Wie findet Maus den Käse?



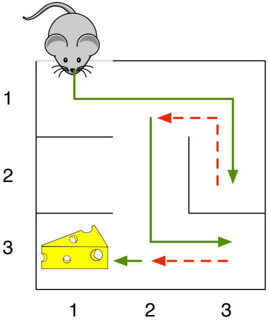
Backtracking: Labyrinth I

Problem: Wie findet Maus den Käse?

Lösung:

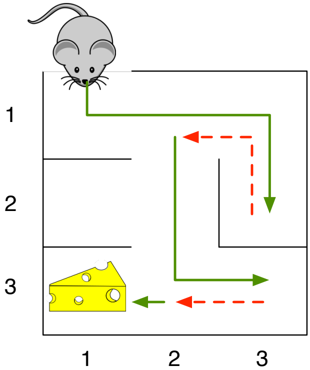
- systematisches Abgehen des Labyrinths
- Zurückgehen falls Sackgasse (daher: **Backtracking**)

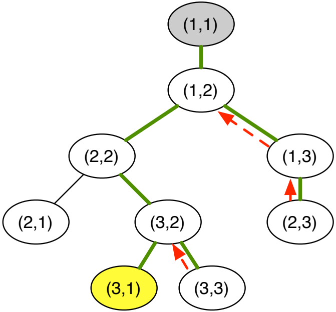
  ⇒ "trial and error"



Backtracking: Labyrinth II

Mögliche Wege repräsentiert als **Baum**:





Algorithmenmuster: Backtracking

Voraussetzungen:

- Lösungs(teil)raum repräsentiert als **Konfiguration** K
- K_0 ist Startkonfiguration
- jede Konfiguration K_i kann **direkt erweitert** werden
- für jede Konfiguration ist entscheidbar, ob Lösung

```
Input: Konfiguration K

Backtrack(K)
    if (K Loesung)
        gib K aus;
    else
        foreach (Erweiterung K’ von K)
            Backtrack(K’)
```

Algorithmenmuster: Backtracking

Voraussetzungen:

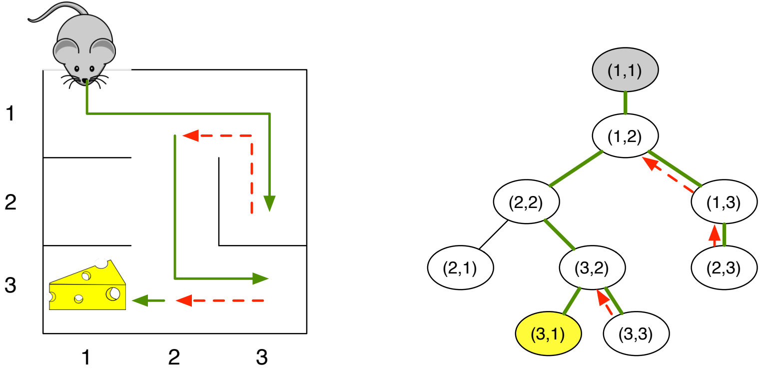
- Lösungs(teil)raum repräsentiert als **Konfiguration** K
- K_0 ist Startkonfiguration
- jede Konfiguration K_i kann **direkt erweitert** werden
- für jede Konfiguration ist entscheidbar, ob Lösung

```
Input: Konfiguration K

Backtrack(K)
    if (K Loesung)
        gib K aus;
    else
        foreach (Erweiterung K' von K)
            Backtrack(K')
```

⇒ initialer Aufruf mittels Backtrack(K_0)

Backtracking: Konfigurationen



Konfiguration z.B. repräsentiert als **Pfad** im Baum

Backtracking: Eigenschaften

Terminierung von Backtracking:

- nur wenn Lösungsraum **endlich**
- nur wenn sichergestellt dass Konfigurationen **nicht wiederholt getestet** werden

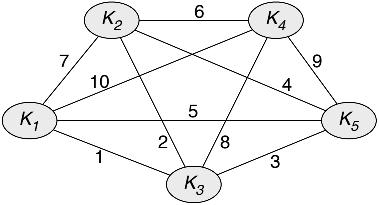
Komplexität von Backtracking:

- direkt abhängig von Größe des Lösungsraums
- meist **exponentiell**, also $O(2^n)$, oder schlimmer!
- nur für kleine Probleme wirklich anwendbar

Backtracking Beispiel: Traveling Salesman

Traveling Salesman Problem:

- n Städte
- finde **kürzeste Rundreise**, die alle Städte exakt einmal besucht
  - ausser Start- und Zielort (identisch)



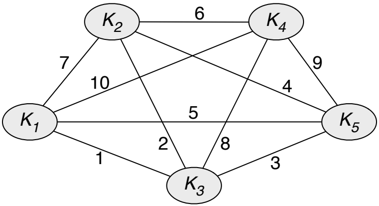
→ Lösung z.B. mit Algorithmen-Muster Backtracking

Traveling Salesman Problem: Algorithmus mit Backtracking

```
Input: n Staedte, Rundreise trip

TSP(trip)
    if (trip besucht jede Stadt)
        erweitere trip um Reise zum Standort;
        gebe trip und Kosten aus;
    else
        foreach (bislang unbesuchte Stadt s)
            trip' = trip erweitert um s
            TSP(trip');
```

Traveling Salesman Problem: Beispiel



bei n Städten mit fixiertem Start-/Zielort gibt es $(n-1)!$ Rundreisen

- hier: 5 Städte $\rightarrow 4! = 24$ Rundreisen

Laufzeit von **TSP** hier ist $O\big((n-1)!\big)$

hier: kürzeste Rundreise hat Länge 23

- z.B. über Route $K_1 \rightarrow K_3 \rightarrow K_2 \rightarrow K_4 \rightarrow K_5 \rightarrow K_1$

Backtracking Beispiel: Acht-Damen-Problem

Acht-Damen-Problem:

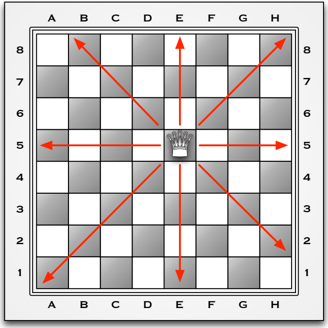
- suche alle Konfigurationen von 8 Damen auf Schachbrett
- so dass keine Dame eine andere bedroht

Backtracking Beispiel: Acht-Damen-Problem

Acht-Damen-Problem:

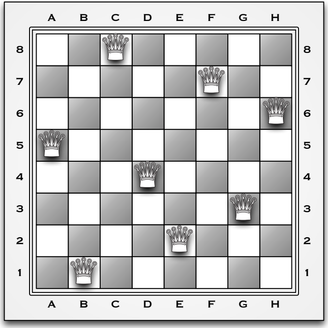
- suche alle Konfigurationen von 8 Damen auf Schachbrett
- so dass keine Dame eine andere bedroht

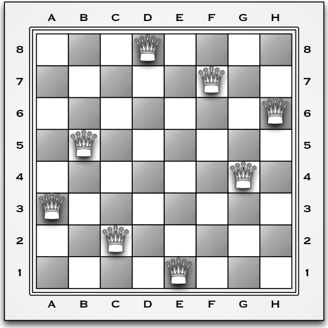
Dame auf Schachbrett:



Acht-Damen-Problem

Zwei der möglichen Lösungen:





Beobachtung: jeweils nur eine Dame pro Zeile/Spalte

→ Lösung z.B. mit Algorithmen-Muster Backtracking

Acht-Damen-Problem: Algorithmus mit Backtracking

```
Input: Zeilenindex i

AchtDamen(i)
    if (i == 9)
        gib Loesung aus;
        return;
    for h=1 to 8
        if (Feld in Zeile i, Spalte h nicht bedroht)
            setze Dame auf Feld (i,h);
            AchtDamen(i+1);
            entferne Dame von Feld (i,h);
```

Acht-Damen-Problem

- es gibt **92 Lösungen** für das Acht-Damen-Problem
- das Problem lässt sich auf n Damen auf einem $n \times n$ Schachbrett ausweiten
  - Anzahl Lösungen wächst stark
  - z.B. für $n = 13$ gibt es 73712 Lösungen
- ähnliche Spiele, wie z.B. **Sudoku**, lassen sich entsprechend lösen

Dynamisches Programmieren

Dynamisches Programmieren

- einsetzbar für Probleme, deren optimale Lösung sich aus optimalen Lösungen von Teilproblemen zusammensetzt (z.B. Rekursion)

Prinzip:

- statt Rekursion berechnet man vom kleinsten Teilproblem **“aufwärts”**
- Zwischenergebnisse werden in **Tabellen** gespeichert

Beispiel: Fibonacci Zahlen

Fibonacci Folge

Die **Fibonacci Folge** ist eine Folge natürlicher Zahlen $f_1, f_2, f_3, \ldots$, für die gilt

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

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

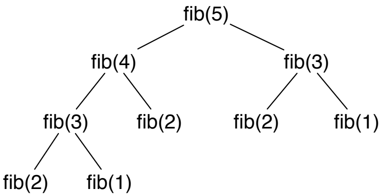
Beispiel: Fibonacci Funktion

```
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):



Fibonacci Funktion: dynamisch programmiert

```
Input:  Index n der Fibonaccifolge
Output: F_n

FibDyn(n)
    fib = new long[n+1];
    fib[1] = 1;
    fib[2] = 2;
    for (k=3; k <= n; k++)
        fib[k] = fib[k-1] + fib[k-2];
    res = fib[n];
    delete fib;
    return fib[n];
```

- Komplexität dynamisch programmiert: $O(n)$