

# Datenstrukturen und Algorithmen

Vorlesung am D-Math (CSE) der ETH Zürich

Felix Friedrich

FS 2019

# Willkommen!

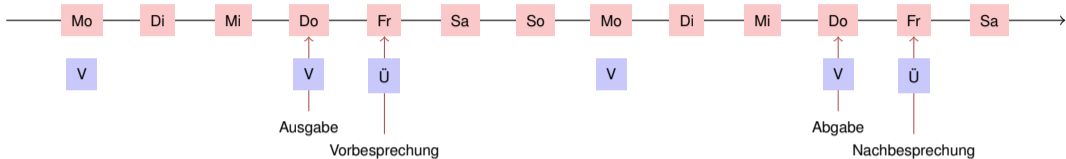
Vorlesungshomepage:

<http://lec.inf.ethz.ch/DA/2019>

Das Team:

Assistenten	Philippe Schlattner Jan Stratmann Robin Worreby Robin Vogtland
Back-Office	Aritra Dhar Pesho Ivanov
Dozent	Felix Friedrich

# Übungsbetrieb



- Übungsblattausgabe zur Vorlesung (online).
- Vorbesprechung in der folgenden Übung.
- Bearbeitung der Übung bis spätestens am Tag vor der nächsten Übungsstunde (23:59h).
- Nachbesprechung der Übung in der nächsten Übungsstunde. Feedback zu den Abgaben innerhalb einer Woche nach Nachbesprechung.

# Zu den Übungen

- Bearbeitung der wöchentlichen Übungsserien ist freiwillig, wird aber *dringend* empfohlen!

# Es ist so einfach!

Für die Übungen verwenden wir eine Online-Entwicklungsumgebung, benötigt lediglich einen Browser, Internetverbindung und Ihr ETH Login.

Falls Sie keinen Zugang zu einem Computer haben: in der ETH stehen an vielen Orten öffentlich Computer bereit.

# Literatur

**Algorithmen und Datenstrukturen**, *T. Ottmann, P. Widmayer*, Spektrum-Verlag, 5. Auflage, 2011

**Algorithmen - Eine Einführung**, *T. Cormen, C. Leiserson, R. Rivest, C. Stein*, Oldenbourg, 2010

**Introduction to Algorithms**, *T. Cormen, C. Leiserson, R. Rivest, C. Stein*, 3rd ed., MIT Press, 2009

**The C++ Programming Language**, *B. Stroustrup*, 4th ed., Addison-Wesley, 2013.

**The Art of Multiprocessor Programming**, *M. Herlihy, N. Shavit*, Elsevier, 2012.

# Relevantes für die Prüfung

Prüfungsstoff für die Endprüfung schliesst ein

- Vorlesungsinhalt (Vorlesung, Handout) und
- Übungsinhalte (Übungsstunden, Übungsblätter).

Prüfung (120 min) ist schriftlich. Hilfsmittel: vier A4-Seiten (bzw. 2 A4-Blätter doppelseitig) entweder handgeschrieben oder mit Fontgrösse minimal 11 Punkt.

# Unser Angebot

- Bearbeitung der wöchentlichen Übungsserien → Bonus von maximal 0.25 Notenpunkten für die Prüfung.
- Bonus proportional zur erreichten Punktzahl von **speziell markierten Bonus-Aufgaben**. Volle Punktzahl  $\hat{=}$  0.25.
- **Zulassung** zu speziell markierten Bonusaufgaben kann von der erfolgreichen Absolvierung anderer Übungsaufgaben abhängen.



# Unser Angebot (Konkret)

- Insgesamt 4 Bonusaufgaben; 3/4 der Punkte reichen für 0.25 Bonuspunkte für die Prüfung
- Sie können also z.B. 3 Bonusaufgaben zu 100% lösen, oder 4 Bonusaufgaben zu je 75%, oder ...
- Bonusaufgaben müssen durch erfolgreich gelöste Übungsserien freigeschaltet (→ Experience Points) werden
- Es müssen wiederum nicht alle Übungsserien vollständig gelöst werden, um eine Bonusaufgabe freizuschalten
- Details: Übungsstunden, Online-Übungssystem (Code Expert)

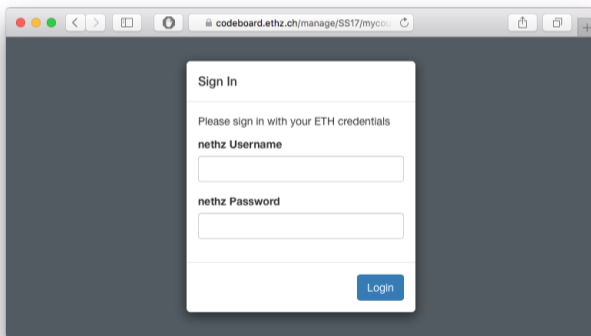
# Akademische Lauterkeit

**Regel:** Sie geben nur eigene Lösungen ab, welche Sie selbst verfasst und verstanden haben.

Wir prüfen das (zum Teil automatisiert) nach und behalten uns disziplinarische Massnahmen vor.

# Einschreibung in Übungsgruppen - I

- Besuchen Sie <http://expert.ethz.ch/enroll/SS19/da>
- Loggen Sie sich mit Ihrem nethz Account ein.

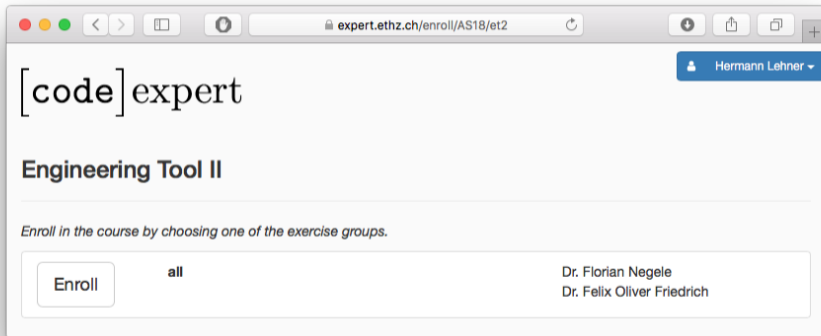


The image shows a browser window with the address bar containing `codeboard.ethz.ch/manage/SS17/myco...`. The main content area displays a 'Sign In' form with the following elements:

- Title: Sign In
- Instruction: Please sign in with your ETH credentials
- Label: nethz Username
- Input field: A text box for the username.
- Label: nethz Password
- Input field: A text box for the password.
- Button: A blue button labeled 'Login'.

# Einschreibung in Übungsgruppen - II

Schreiben Sie sich im folgenden Dialog in eine Übungsgruppe ein.



# Übersicht

## [code]expert

Felix Oliver Friedrich ▾

Autum 2017 ▾

Enrolled Courses

My Exercise Groups

My Courses

### Demo Course Demo Group - Dr. Hermann Lehner [change...](#)

#### Coding Demo Exercise

Earned XP

Submissions

Handout Date

Due Date

[Tasks](#) [Solutions](#)

1,000 / 1,000

9. Sep. 2017 00:00

31. Dez. 2027 00:00

[Quadratic Equations In C++](#)

1,000 ✓

100%

[Hand in now](#)

#### Markdown Editor Manual

Submissions

Handout Date

Due Date

[Tasks](#) [Solutions](#)

1. Aug. 2017 00:00

1. Aug. 2017 00:01

[Basic Markdown Syntax](#)

[Code Blocks and Inline Code](#)

# Programmierübung

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min;
6     max = min;
7     for (int i = 0; i < 8; ++i){ // (there is
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << "/" << max << std::endl;
14 }
```

Project Files  
main.cpp

Minimax - Student Attempt

Felix Oliver Friedrich  
Status Not submitted yet

Task  
History

D: Beschreibung  
E: History

imum and  
maximum of a series of ten integers.

- Input format: 10 consecutive integers  
number:int , example:  
0 100250 45 0 0 1 -1000001 45 -25065 1
- Expected output format: minimum:int  
"/" maximum:int , example:  
-1000001/100251

A: compile  
B: run  
C: test

> Console

# Testen und Abgeben

The screenshot displays a C++ IDE interface with the following components:

- Project Files:** A sidebar on the left showing the file structure with `main.cpp`.
- Code Editor:** The main area shows the source code for `Minimax - Student Attempt`. The code is as follows:

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min; std::cin >> max;
6     max = min-1;
7     for (int i = 0; i < 8; ++i){
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << "/" << max << std::endl;
14 }
```
- Test Results:** A console window below the code shows the output of running tests:

```
Running tests.....
min_first passed
min_last passed
min_middle passed
max_first failed
input:
100251 -25065 45 -1000001 1 0 0 45 100250 0
expected output:
-1000001/100251
actual output:
-1000001/100250
-----
max_last passed
max_middle passed
unique passed

Tests result: passed 6 of 7 / score: 86% [ ]
```
- Submission Panel:** On the right side, the user `Felix Oliver Friedrich` is logged in. It shows a status `is Not submitted yet` and a green button labeled `Create new Submission`. A red arrow points from a pink box labeled `Abgeben` to this button.
- Snapshot History:** Below the submission button, there is a section for snapshots with a `Create Snapshot` button and a list of snapshots: `First Working Version` (2 minutes ago) and `Initial Snapshot` (13 minutes ago), each with navigation icons.

# Wo ist der Save Knopf?

- Das Filesystem ist transaktionsbasiert und es wird laufend gespeichert („autosave“). Beim Öffnen eines Projektes findet man immer den zuletzt gesehenen Zustand wieder.
- Der derzeitige Stand kann als (benannter) *Snapshot* festgehalten werden. Zu gespeicherten Snapshots kann jederzeit zurückgekehrt werden.
- Der aktuelle Stand kann als Snapshot abgegeben (submitted) werden. Zudem kann jeder gespeicherte Snapshot abgegeben werden.



# Snapshots

The image shows a code editor interface with a C++ program and its test results. The code is as follows:

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min; std::cin >> max;
6     max = min;
7     for (int i = 0; i < 8; ++i){ // (there is a bug here)
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << endl;
14 }
```

The test results show that all tests passed:

```
Running tests.....
min_first passed
min_last passed
min_middle passed
max_first passed
max_last passed
max_middle passed
unique passed

Tests result: passed 7 of 7 / score: 100% [██████████]
```

The sidebar on the right shows the snapshot history:

- Really Working Version (2 minutes ago)
- First Working Version (6 minutes ago)
- Initial Snapshot (16 minutes ago)

Red boxes and arrows highlight key features:

- Snapshot betrachten**: Points to the 'First Working Version' entry.
- Abgabe**: Points to the 'Initial Snapshot' entry.
- Zurück zu**: Points to the 'Initial Snapshot' entry.

# Wenn es Probleme gibt ...

- mit dem Kursinhalt
  - unbedingt alle Übungen besuchen
  - dort Fragen stellen
  - und/oder Übungsleiter kontaktieren
- alle weiteren Probleme
  - Email an Dozenten (Felix Friedrich)
- Wir helfen gerne!

# 1. Einführung

Überblick, Algorithmen und Datenstrukturen, Korrektheit, erstes Beispiel

# Ziele des Kurses

- Verständnis des Entwurfs und der Analyse grundlegender Algorithmen und Datenstrukturen.
- Vertiefter Einblick in ein modernes Programmiermodell (mit C++).
- Wissen um Chancen, Probleme und Grenzen des parallelen und nebenläufigen Programmierens.

# Inhalte der Vorlesung

## Datenstrukturen / Algorithmen

Begriff der Invariante, Kostenmodell, Landau Symbole

Algorithmenentwurf, Induktion

Suchen und Auswahl, Sortieren

Amortisierte Analyse

Dynamic Programming

Minimale Spannbäume, Fibonacci Heaps

Kürzeste Wege, Maximaler Fluss

Fundamentale Algorithmen auf Graphen

Wörterbücher: Hashing und Suchbäume, AVL

van-Emde-Boas-Bäume, Splay-Bäume

## Programmieren mit C++

RAII, Move Konstruktion, Smart Pointers,

Templates und Generische Programmierung

Exceptions

Funktoren und Lambdas

Promises and Futures

Threads, Mutexs and Monitors

## Parallel Programming

Parallelität vs. Concurrency, Speedup (Amdahl/Gustavson), Races, Memory Reordering, Atomic Registers, RMW (CAS, TAS), Deadlock/Starvation

## 1.2 Algorithmen

[Cormen et al, Kap. 1; Ottman/Widmayer, Kap. 1.1]

# Algorithmus

Algorithmus: wohldefinierte Berechnungsvorschrift, welche aus Eingabedaten (*input*) Ausgabedaten (*output*) berechnet.

# Beispielproblem

**Input:** Eine Folge von  $n$  Zahlen  $(a_1, a_2, \dots, a_n)$

**Output:** Eine Permutation  $(a'_1, a'_2, \dots, a'_n)$  der Folge  $(a_i)_{1 \leq i \leq n}$ , so dass  
 $a'_1 \leq a'_2 \leq \dots \leq a'_n$

## Mögliche Eingaben

$(1, 7, 3), (15, 13, 12, -0.5), (1) \dots$

Jedes Beispiel erzeugt eine *Probleminstance*.

Die Performanz (Geschwindigkeit) des Algorithmus hängt üblicherweise ab von der Probleminstance. Es gibt oft “gute” und “schlechte” Instanzen.



# Beispiele für Probleme in der Algorithmik

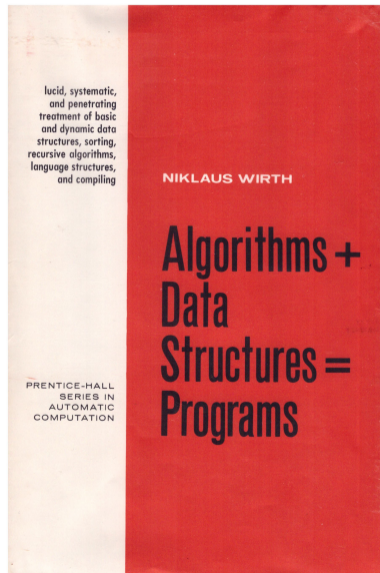
- **Tabellen und Statistiken**: Suchen, Auswählen und Sortieren
- **Routenplanung**: Kürzeste Wege Algorithmus, Heap Datenstruktur
- **DNA Matching**: Dynamic Programming
- **Auswertungsreihenfolge**: Topologische Sortierung
- **Autovervollständigung**: Wörterbücher/Bäume
- **Schnelles Nachschlagen** : Hash-Tabellen
- **Der Handlungsreisende**: Dynamische Programmierung, Minimal aufspannender Baum, Simulated Annealing,

# Charakteristik

- Extrem grosse Anzahl potentieller Lösungen
- Praktische Anwendung

# Datenstrukturen

- Eine Datenstruktur *organisiert Daten* so in einem Computer, dass man sie (in den darauf operierenden Algorithmen) *effizient nutzen* kann.
- Programme = Algorithmen + Datenstrukturen.



# Effizienz

Illusion:

- Wären Rechner unendlich schnell und hätten unendlich viel Speicher ...
- ... dann bräuchten wir die Theorie der Algorithmen (nur) für Aussagen über Korrektheit (incl. Terminierung).

Realität: Ressourcen sind beschränkt und nicht umsonst:

- Rechenzeit → Effizienz
- Speicherplatz → Effizienz

**Eigentlich geht es in diesem Kurs nur um Effizienz.**

# Schwierige Probleme

- NP-vollständige Probleme: Keine bekannte effiziente Lösung (Existenz einer effizienten Lösung ist zwar sehr unwahrscheinlich – es ist aber unbewiesen, dass es keine gibt!)
- Beispiel: Travelling Salesman Problem

**In diesem Kurs beschäftigen wir uns *hauptsächlich* mit Problemen, die effizient (in Polynomialzeit) lösbar sind.**

## 2. Effizienz von Algorithmen

Effizienz von Algorithmen, Random Access Machine Modell,  
Funktionenwachstum, Asymptotik [Cormen et al, Kap. 2.2,3,4.2-4.4 |  
Ottman/Widmayer, Kap. 1.1]

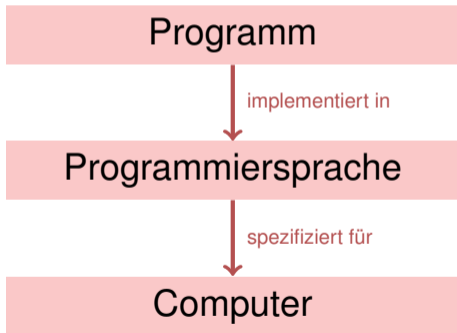
# Effizienz von Algorithmen

## Ziele

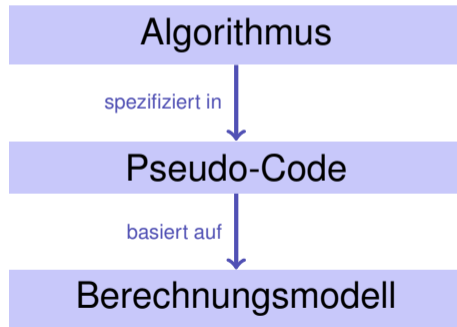
- Laufzeitverhalten eines Algorithmus maschinenunabhängig quantifizieren.
- Effizienz von Algorithmen vergleichen.
- Abhängigkeit von der Eingabegrösse verstehen.

# Programme und Algorithmen

## Technologie



## Abstraktion





# Technologiemodell

## *Random Access Machine (RAM)*

- Ausführungsmodell: Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt.
- Speichermodell: Konstante Zugriffszeit (grosses Array)
- Elementare Operationen: Rechenoperation ( $+$ ,  $-$ ,  $\cdot$ , ...) , Vergleichsoperationen, Zuweisung / Kopieroperation auf Maschinenworten (Registern), Flusskontrolle (Sprünge)
- Einheitskostenmodell: elementare Operation hat Kosten 1.
- Datentypen: Fundamentaltypen wie grössenbeschränkte Ganzzahl oder Fließkommazahl.

# Grösse der Eingabedaten

Typisch: Anzahl Eingabeobjekte (von fundamentalem Typ).

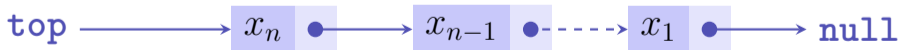
Oftmals: Anzahl Bits für eine *vernünftige / kostengünstige* Repräsentation der Daten.

Annahme: fundamentale Typen passen in Maschinenwort (*word*) mit Grösse :  $w \geq \log(\text{sizeof}(\text{mem}))$  Bits.

# Pointer Machine Modell

Wir nehmen an

- Objekte beschränkter Grösse können dynamisch erzeugt werden in konstanter Zeit 1.
- Auf Felder (mit Wortgrösse) der Objekte kann in konstanter Zeit 1 zugegriffen werden.



# Asymptotisches Verhalten

Genauere Laufzeit eines Algorithmus lässt sich selbst für kleine Eingabedaten kaum voraussagen.

- Betrachten das asymptotische Verhalten eines Algorithmus.
- Ignorieren alle konstanten Faktoren.

## Beispiel

Eine Operation mit Kosten 20 ist genauso gut wie eine mit Kosten 1.  
Lineares Wachstum mit Steigung 5 ist genauso gut wie lineares Wachstum mit Steigung 1.

# Algorithmen, Programme und Laufzeit

Programm: Konkrete Implementation eines Algorithmus.

Laufzeit des Programmes: messbarer Wert auf einer konkreten Maschine. Kann sowohl nach oben, wie auch nach unten abgeschätzt werden.

## Beispiel

Rechner mit 3 GHz. Maximale Anzahl Operationen pro Taktzyklus (z.B. 8).  $\Rightarrow$  untere Schranke.

Einzelne Operation dauert mit Sicherheit nie länger als ein Tag  $\Rightarrow$  obere Schranke.

Hinsichtlich des *asymptotischen Verhaltens* des Programmes spielen die Schranken keine Rolle.

## 2.2 Funktionenwachstum

$\mathcal{O}$ ,  $\Theta$ ,  $\Omega$  [Cormen et al, Kap. 3; Ottman/Widmayer, Kap. 1.1]

# Oberflächlich

Verwende die asymptotische Notation zur Kennzeichnung der Laufzeit von Algorithmen

Wir schreiben  $\Theta(n^2)$  und meinen, dass der Algorithmus sich für grosse  $n$  wie  $n^2$  verhält: verdoppelt sich die Problemgrösse, so vervierfacht sich die Laufzeit.

# Genauer: Asymptotische obere Schranke

Gegeben: Funktion  $g : \mathbb{N} \rightarrow \mathbb{R}$ .

Definition:<sup>1</sup>

$$\begin{aligned} \mathcal{O}(g) = \{ f : \mathbb{N} \rightarrow \mathbb{R} \mid \\ \exists c > 0, \exists n_0 \in \mathbb{N} : \\ \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n) \} \end{aligned}$$

Schreibweise:

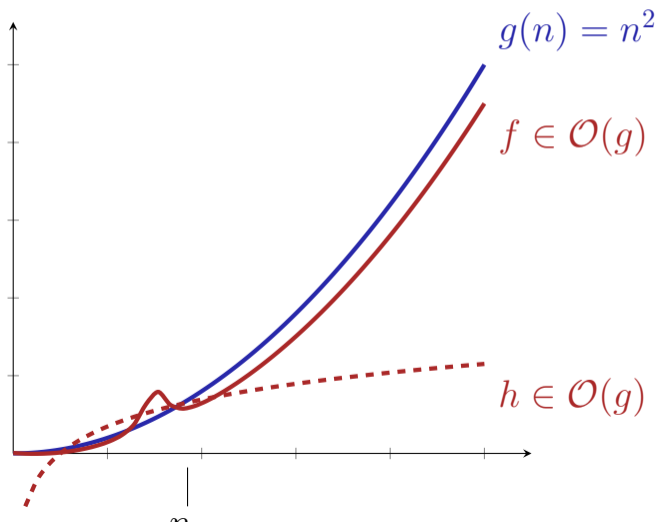
$$\mathcal{O}(g(n)) := \mathcal{O}(g(\cdot)) = \mathcal{O}(g).$$

---

<sup>1</sup>Ausgesprochen: Menge aller reellwertiger Funktionen  $f : \mathbb{N} \rightarrow \mathbb{R}$  für die gilt: es gibt ein (reellwertiges)  $c > 0$  und ein  $n_0 \in \mathbb{N}$  so dass  $0 \leq f(n) \leq c \cdot g(n)$  für alle  $n \geq n_0$ .



# Anschauung



# Beispiele

$$\mathcal{O}(g) = \{f : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$$

$f(n)$	$f \in \mathcal{O}(?)$	Beispiel
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$	$\mathcal{O}(n)$	$c = 2, n_0 = 0$
$n^2 + 100n$	$\mathcal{O}(n^2)$	$c = 2, n_0 = 100$
$n + \sqrt{n}$	$\mathcal{O}(n)$	$c = 2, n_0 = 1$

# Eigenschaft

$$f_1 \in \mathcal{O}(g), f_2 \in \mathcal{O}(g) \Rightarrow f_1 + f_2 \in \mathcal{O}(g)$$

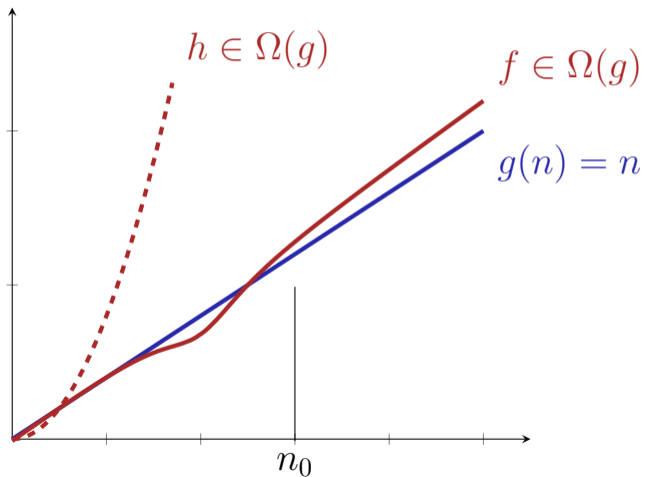
# Umkehrung: Asymptotische untere Schranke

Gegeben: Funktion  $g : \mathbb{N} \rightarrow \mathbb{R}$ .

Definition:

$$\Omega(g) = \{f : \mathbb{N} \rightarrow \mathbb{R} \mid \\ \exists c > 0, \exists n_0 \in \mathbb{N} : \\ \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\}$$

# Beispiel



# Asymptotisch scharfe Schranke

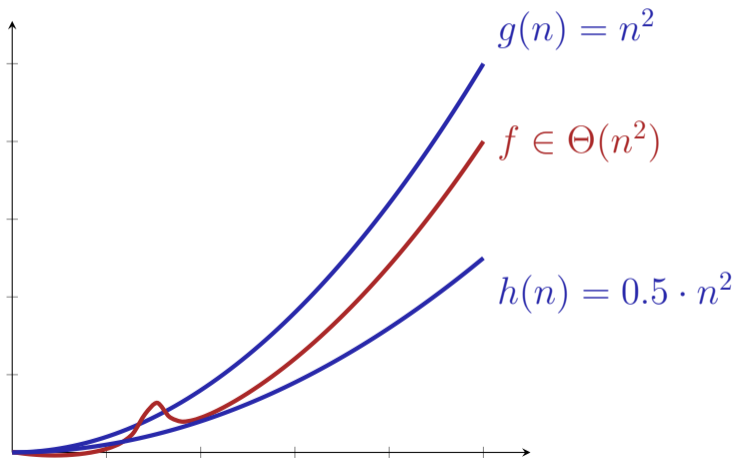
Gegeben Funktion  $g : \mathbb{N} \rightarrow \mathbb{R}$ .

Definition:

$$\Theta(g) := \Omega(g) \cap \mathcal{O}(g).$$

Einfache, geschlossene Form: Übung.

# Beispiel

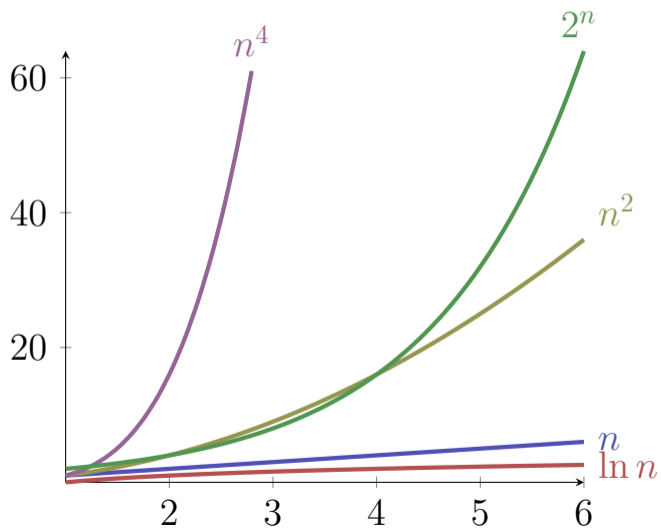


# Wachstumsbezeichnungen

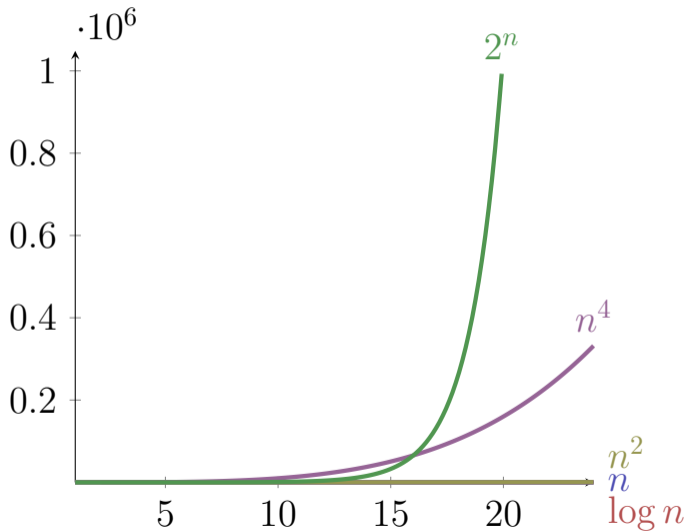
$\mathcal{O}(1)$	beschränkt	Array-Zugriff
$\mathcal{O}(\log \log n)$	doppelt logarithmisch	Binäre sortierte Suche interpoliert
$\mathcal{O}(\log n)$	logarithmisch	Binäre sortierte Suche
$\mathcal{O}(\sqrt{n})$	wie die Wurzelfunktion	Primzahltest (naiv)
$\mathcal{O}(n)$	linear	Unsortierte naive Suche
$\mathcal{O}(n \log n)$	superlinear / loglinear	Gute Sortieralgorithmen
$\mathcal{O}(n^2)$	quadratisch	Einfache Sortieralgorithmen
$\mathcal{O}(n^c)$	polynomial	Matrixmultiplikation
$\mathcal{O}(2^n)$	exponentiell	Travelling Salesman Dynamic Programming
$\mathcal{O}(n!)$	faktoriell	Travelling Salesman naiv



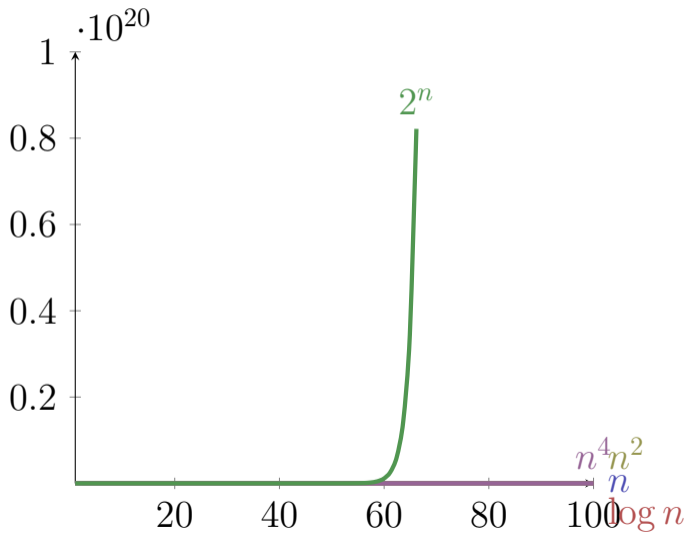
# Kleine $n$



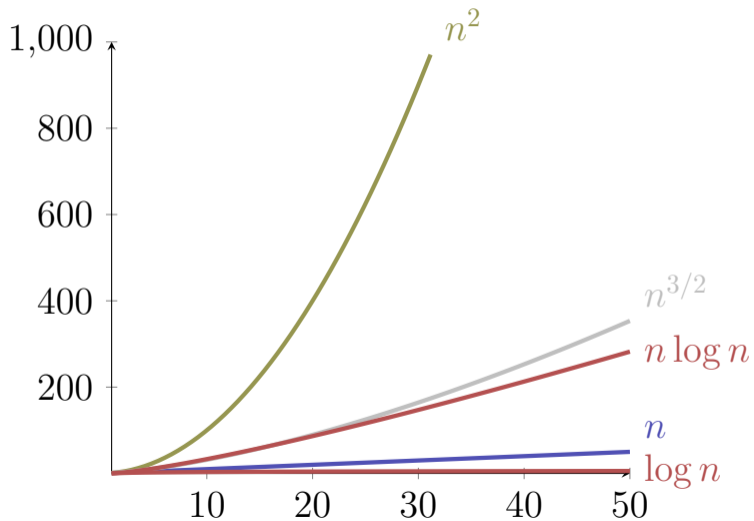
# Grössere $n$



# “Grosse” $n$



# Logarithmen!



# Zeitbedarf

Annahme: 1 Operation =  $1\mu s$ .

Problemgrösse	1	100	10000	$10^6$	$10^9$
$\log_2 n$	$1\mu s$	$7\mu s$	$13\mu s$	$20\mu s$	$30\mu s$
$n$	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 Minuten
$n \log_2 n$	$1\mu s$	$700\mu s$	$13/100\mu s$	$20s$	8.5 Stunden
$n^2$	$1\mu s$	$1/100s$	1.7 Minuten	11.5 Tage	317 Jahrhund.
$2^n$	$1\mu s$	$10^{14}$ Jahrh.	$\approx \infty$	$\approx \infty$	$\approx \infty$

## Theorem

Seien  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$  zwei Funktionen. Dann gilt:

- 1  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in \mathcal{O}(g), \mathcal{O}(f) \subsetneq \mathcal{O}(g).$
- 2  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = C > 0$  ( $C$  konstant)  $\Rightarrow f \in \Theta(g).$
- 3  $\frac{f(n)}{g(n)} \xrightarrow{n \rightarrow \infty} \infty \Rightarrow g \in \mathcal{O}(f), \mathcal{O}(g) \subsetneq \mathcal{O}(f).$

# Zur Notation

Übliche informelle Schreibweise

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

ist zu verstehen als  $f \in \mathcal{O}(g)$ .

Es gilt nämlich

$$f_1 = \mathcal{O}(g), f_2 = \mathcal{O}(g) \not\Rightarrow f_1 = f_2!$$

## Beispiel

$n = \mathcal{O}(n^2), n^2 = \mathcal{O}(n^2)$  aber natürlich  $n \neq n^2$ .

*Wir vermeiden diese Schreibweise, wo sie zu Mehrdeutigkeiten führen könnte.*

# Erinnerung: Effizienz: Arrays vs. Verkettete Listen

- Speicher: Unser `avec` belegt ungefähr  $n$  ints (Vektorgrösse  $n$ ), unser `llvec` ungefähr  $3n$  ints (ein Zeiger belegt i.d.R. 8 Byte)
- Laufzeit (mit `avec = std::vector`, `llvec = std::list`):

```
prepending (insert at front) [100,000x]:
  ▶ avec:    675 ms
  ▶ llvec:   10 ms
appending (insert at back) [100,000x]:
  ▶ avec:    2 ms
  ▶ llvec:   9 ms
removing first [100,000x]:
  ▶ avec:    675 ms
  ▶ llvec:   4 ms
removing last [100,000x]:
  ▶ avec:    0 ms
  ▶ llvec:   4 ms
removing randomly [10,000x]:
  ▶ avec:    3 ms
  ▶ llvec:  113 ms
inserting randomly [10,000x]:
  ▶ avec:    16 ms
  ▶ llvec:  117 ms
fully iterate sequentially (5000 elements) [5,000x]:
  ▶ avec:   354 ms
  ▶ llvec:  525 ms
```



# Asymptotische Laufzeiten

Mit unserer neuen Sprache ( $\Omega$ ,  $\mathcal{O}$ ,  $\Theta$ ) können wir das *Verhalten der Datenstrukturen und ihrer Algorithmen präzisieren*.

Typische Asymptotische Laufzeiten (Vorgriff!)

Datenstruktur	Wahlfreier Zugriff	Einfügen	Nächstes	Einfügen nach Element	Suchen
<code>std::vector</code>	$\Theta(1)$	$\Theta(1) A$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
<code>std::list</code>	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
<code>std::set</code>	–	$\Theta(\log n)$	$\Theta(\log n)$	–	$\Theta(\log n)$
<code>std::unordered_set</code>	–	$\Theta(1) P$	–	–	$\Theta(1) P$

$A$  = amortisiert,  $P$  = erwartet, sonst schlechtester Fall („worst case“)

# Komplexität

*Komplexität* eines Problems  $P$ : minimale (asymptotische) Kosten über alle Algorithmen  $A$ , die  $P$  lösen.

Komplexität der Elementarmultiplikation zweier Zahlen der Länge  $n$  ist  $\Omega(n)$  und  $\mathcal{O}(n^{\log_3 2})$  (Karatsuba Ofman).

## Exemplarisch:

Problem	Komplexität	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
		↑	↑	↑
Algorithmus	Kosten <sup>2</sup>	$3n - 4$	$\mathcal{O}(n)$	$\Theta(n^2)$
		↓	↕	↕
Programm	Laufzeit	$\Theta(n)$	$\mathcal{O}(n)$	$\Theta(n^2)$

---

<sup>2</sup>Anzahl Elementaroperationen

# 3. Beispiele

Korrektheit zeigen, Rekursion und Rekurrenzen  
[Literaturangaben bei den Beispielen]

## 3.1 Altägyptische Multiplikation

Altägyptische Multiplikation – Ein Beispiel, wie man Korrektheit von Algorithmen zeigen kann.

# Altägyptische Multiplikation<sup>3</sup>

Berechnung von  $11 \cdot 9$

11		9
<del>22</del>		<del>4</del>
<del>44</del>		<del>2</del>
88		1
99		—

9		11
18		5
<del>36</del>		<del>2</del>
72		1
99		—

- 1 Links verdoppeln, rechts ganzzahlig halbieren.
- 2 Gerade Zahl rechts  $\Rightarrow$  Zeile streichen.
- 3 Übrige Zeilen links addieren.

<sup>3</sup>Auch bekannt als Russische Bauernmultiplikation

# Vorteile

- Kurze Beschreibung, einfach zu verstehen.
- Effizient für Computer im Dualsystem: Verdoppeln = Left Shift, Halbieren = Right Shift

## Beispiel

*left shift*     $9 = 01001_2 \rightarrow 10010_2 = 18$

*right shift*     $9 = 01001_2 \rightarrow 00100_2 = 4$

# Fragen

- Für welche Eingaben liefert der Algorithmus das richtige Resultat (in endlicher Zeit)?
- Wie beweist man seine Korrektheit?
- Was ist ein gutes Mass für seine Effizienz?

# Die Essenz

Wenn  $b > 1$ ,  $a \in \mathbb{Z}$ , dann:

$$a \cdot b = \begin{cases} 2a \cdot \frac{b}{2} & \text{falls } b \text{ gerade,} \\ a + 2a \cdot \frac{b-1}{2} & \text{falls } b \text{ ungerade.} \end{cases}$$



# Terminierung

$$a \cdot b = \begin{cases} a & \text{falls } b = 1, \\ 2a \cdot \frac{b}{2} & \text{falls } b \text{ gerade,} \\ a + 2a \cdot \frac{b-1}{2} & \text{falls } b \text{ ungerade.} \end{cases}$$

# Rekursiv funktional notiert

$$f(a, b) = \begin{cases} a & \text{falls } b = 1, \\ f(2a, \frac{b}{2}) & \text{falls } b \text{ gerade,} \\ a + f(2a, \frac{b-1}{2}) & \text{falls } b \text{ ungerade.} \end{cases}$$

# Als Funktion programmiert

```
// pre: b>0
// post: return a*b
int f(int a, int b){
    if(b==1)
        return a;
    else if (b%2 == 0)
        return f(2*a, b/2);
    else
        return a + f(2*a, (b-1)/2);
}
```

# Korrektheit: Mathematischer Beweis

$$f(a, b) = \begin{cases} a & \text{falls } b = 1, \\ f(2a, \frac{b}{2}) & \text{falls } b \text{ gerade,} \\ a + f(2a \cdot \frac{b-1}{2}) & \text{falls } b \text{ ungerade.} \end{cases}$$

Zu zeigen:  $f(a, b) = a \cdot b$  für  $a \in \mathbb{Z}$ ,  $b \in \mathbb{N}^+$ .

# Korrektheit: Mathematischer Beweis per Induktion

Sei  $a \in \mathbb{Z}$ , zu zeigen  $f(a, b) = a \cdot b \quad \forall b \in \mathbb{N}^+$ .

**Anfang:**  $f(a, 1) = a = a \cdot 1$

**Hypothese:**  $f(a, b') = a \cdot b' \quad \forall 0 < b' \leq b$

**Schritt:**  $f(a, b') = a \cdot b' \quad \forall 0 < b' \leq b \stackrel{!}{\Rightarrow} f(a, b + 1) = a \cdot (b + 1)$

$$f(a, b + 1) = \begin{cases} f(2a, \overbrace{\frac{b+1}{2}}^{0 < \cdot \leq b}) \stackrel{i.H.}{=} a \cdot (b + 1) & \text{falls } b > 0 \text{ ungerade,} \\ a + f(2a, \underbrace{\frac{b}{2}}_{0 < \cdot < b}) \stackrel{i.H.}{=} a + a \cdot b & \text{falls } b > 0 \text{ gerade.} \end{cases}$$



# [Code-Umformung: Endrekursion]

Die Rekursion lässt sich *endrekursiv* schreiben

```
// pre: b>0
// post: return a*b
int f(int a, int b){
    if(b==1)
        return a;
    else if (b%2 == 0)
        return f(2*a, b/2);
    else
        return a + f(2*a, (b-1)/2);
}
```



```
// pre: b>0
// post: return a*b
int f(int a, int b){
    if(b==1)
        return a;
    int z=0;
    if (b%2 != 0){
        --b;
        z=a;
    }
    return z + f(2*a, b/2);
}
```

# [Code-Umformung: Endrekursion $\Rightarrow$ Iteration]

```
// pre: b>0
// post: return a*b
int f(int a, int b){
    if(b==1)
        return a;
    int z=0;
    if (b%2 != 0){
        --b;
        z=a;
    }
    return z + f(2*a, b/2);
}
```



```
int f(int a, int b) {
    int res = 0;
    while (b != 1) {
        int z = 0;
        if (b % 2 != 0){
            --b;
            z = a;
        }
        res += z;
        a *= 2; // neues a
        b /= 2; // neues b
    }
    res += a; // Basisfall b=1
    return res;
}
```

# [Code-Umformung: Vereinfachen]

```
int f(int a, int b) {  
    int res = 0;  
    while (b != 1) {  
        int z = 0;  
        if (b % 2 != 0){  
            --b; → Teil der Division  
            z = a; → Direkt in res  
        }  
        res += z;  
        a *= 2;  
        b /= 2;  
    }  
    res += a; → in den Loop  
    return res;  
}
```



```
// pre: b>0  
// post: return a*b  
int f(int a, int b) {  
    int res = 0;  
    while (b > 0) {  
        if (b % 2 != 0)  
            res += a;  
        a *= 2;  
        b /= 2;  
    }  
    return res;  
}
```



# Korrektheit: Argumentation mit Invarianten!

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
    int res = 0;
    -----
    while (b > 0) {
        if (b % 2 != 0){
            -----
            res += a;
            --b;
            -----
        }
        -----
        a *= 2;
        b /= 2;
        -----
    }
    -----
    return res;
}
```

Sei  $x := a \cdot b$ .

Hier gilt  $x = a \cdot b + res$

Wenn hier  $x = a \cdot b + res \dots$

$\dots$  dann auch hier  $x = a \cdot b + res$   
 $b$  gerade

Hier gilt  $x = a \cdot b + res$

Hier gilt  $x = a \cdot b + res$  und  $b = 0$

Also  $res = x$ .

# Zusammenfassung

Der Ausdruck  $a \cdot b + res$  ist eine *Invariante*.

- Werte von  $a$ ,  $b$ ,  $res$  ändern sich, aber die Invariante bleibt "im Wesentlichen" unverändert: Invariante vorübergehend durch eine Anweisung zerstört, aber dann darauf wieder hergestellt. Betrachtet man solche Aktionsfolgen als atomar, bleibt der Wert tatsächlich invariant
- Insbesondere erhält die Schleife die Invariante (*Schleifeninvariante*), sie wirkt dort wie der Induktionsschritt bei der vollständigen Induktion
- Invarianten sind offenbar mächtige Beweishilfsmittel!

# [Weiteres Kürzen]

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
    int res = 0;
    while (b > 0) {
        if (b % 2 != 0){
            res += a;
            --b;
        }
        a *= 2;
        b /= 2;
    }
    return res;
}
```



```
// pre: b>0
// post: return a*b
int f(int a, int b) {
    int res = 0;
    while (b > 0) {
        res += a * (b%2);
        a *= 2;
        b /= 2;
    }
    return res;
}
```

# [Analyse]

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
    int res = 0;
    while (b > 0) {
        res += a * (b%2);
        a *= 2;
        b /= 2;
    }
    return res;
}
```

Altägyptische Multiplikation entspricht der Schulmethode zur Basis 2.

$$\begin{array}{r} 1\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \phantom{1\ 0\ 0\ 1} 1\ 0\ 0\ 1 \quad (9) \\ \phantom{1\ 0\ 0\ 1} 1\ 0\ 0\ 1 \quad (18) \\ \hline \phantom{1\ 0\ 0\ 1} 1\ 1\ 0\ 1\ 1 \\ \phantom{1\ 0\ 0\ 1} 1\ 0\ 0\ 1 \quad (72) \\ \hline 1\ 1\ 0\ 0\ 0\ 1\ 1 \quad (99) \end{array}$$

# Effizienz

Frage: Wie lange dauert eine Multiplikation von  $a$  und  $b$ ?

- Mass für die Effizienz

- Gesamtzahl der elementaren Operationen: Verdoppeln, Halbieren, Test auf "gerade", Addition
- Im rekursiven wie im iterativen Code: maximal 6 Operationen pro Aufruf bzw. Durchlauf

- Wesentliches Kriterium:

- Anzahl rekursiver Aufrufe oder
  - Anzahl Schleifendurchläufe(im iterativen Fall)
- $\frac{b}{2^n} \leq 1$  gilt für  $n \geq \log_2 b$ . Also nicht mehr als  $6 \lceil \log_2 b \rceil$  elementare Operationen.

## **3.2 Schnelle Multiplikation von Zahlen**

[Ottman/Widmayer, Kap. 1.2.3]

## Beispiel 2: Multiplikation grosser Zahlen

Primarschule:

$$\begin{array}{rcccc|l} & a & b & & c & d & & & & & \\ & 6 & 2 & \cdot & 3 & 7 & & & & & \\ \hline & & & & & 1 & 4 & & & & d \cdot b \\ & & & & & 4 & 2 & & & & d \cdot a \\ & & & & & & 6 & & & & c \cdot b \\ & & & & & 1 & 8 & & & & c \cdot a \\ \hline = & & 2 & 2 & 9 & 4 & & & & & \end{array}$$

$2 \cdot 2 = 4$  einstellige Multiplikationen.  $\Rightarrow$  Multiplikation zweier  $n$ -stelliger Zahlen:  $n^2$  einstellige Multiplikationen

# Beobachtung

$$\begin{aligned}ab \cdot cd &= (10 \cdot a + b) \cdot (10 \cdot c + d) \\&= 100 \cdot a \cdot c + 10 \cdot a \cdot c \\&\quad + 10 \cdot b \cdot d + b \cdot d \\&\quad + 10 \cdot (a - b) \cdot (d - c)\end{aligned}$$



# Verbesserung?

$a$	$b$		$c$	$d$	
6	2	·	3	7	
<hr/>					
			1	4	$d \cdot b$
			1	4	$d \cdot b$
			1	6	$(a - b) \cdot (d - c)$
			1	8	$c \cdot a$
	1	8			$c \cdot a$
<hr/>					
=	2	2	9	4	

→ 3 einstellige Multiplikationen.

# Grosse Zahlen

$$6237 \cdot 5898 = \underbrace{62}_{a'} \underbrace{37}_{b'} \cdot \underbrace{58}_{c'} \underbrace{98}_{d'}$$

Rekursive / induktive Anwendung:  $a' \cdot c'$ ,  $a' \cdot d'$ ,  $b' \cdot c'$  und  $c' \cdot d'$  wie oben berechnen.

→  $3 \cdot 3 = 9$  statt 16 einstellige Multiplikationen.

# Verallgemeinerung

Annahme: zwei  $n$ -stellige Zahlen,  $n = 2^k$  für ein  $k$ .

$$\begin{aligned}(10^{n/2}a + b) \cdot (10^{n/2}c + d) &= 10^n \cdot a \cdot c + 10^{n/2} \cdot a \cdot c \\ &+ 10^{n/2} \cdot b \cdot d + b \cdot d \\ &+ 10^{n/2} \cdot (a - b) \cdot (d - c)\end{aligned}$$

Rekursive Anwendung dieser Formel: Algorithmus von Karatsuba und Ofman (1962).

# Analyse

$M(n)$ : Anzahl einstelliger Multiplikationen.

Rekursive Anwendung des obigen Algorithmus  $\Rightarrow$   
Rekursionsgleichung:

$$M(2^k) = \begin{cases} 1 & \text{falls } k = 0, \\ 3 \cdot M(2^{k-1}) & \text{falls } k > 0. \end{cases}$$

# Teleskopieren

Iteratives Einsetzen der Rekursionsformel zum Lösen der Rekursionsgleichung.

$$\begin{aligned}M(2^k) &= 3 \cdot M(2^{k-1}) = 3 \cdot 3 \cdot M(2^{k-2}) = 3^2 \cdot M(2^{k-2}) \\ &= \dots \\ &\stackrel{!}{=} 3^k \cdot M(2^0) = 3^k.\end{aligned}$$

# Beweis: Vollständige Induktion

*Hypothese H:*

$$M(2^k) = 3^k.$$

*Induktionsanfang ( $k = 0$ ):*

$$M(2^0) = 3^0 = 1. \quad \checkmark$$

*Induktionsschritt ( $k \rightarrow k + 1$ ):*

$$M(2^{k+1}) \stackrel{\text{def}}{=} 3 \cdot M(2^k) \stackrel{\text{H}}{=} 3 \cdot 3^k = 3^{k+1}.$$



# Vergleich

Primarschulmethode:  $n^2$  einstellige Multiplikationen.

Karatsuba/Ofman:

$$M(n) = 3^{\log_2 n} = (2^{\log_2 3})^{\log_2 n} = 2^{\log_2 3 \log_2 n} = n^{\log_2 3} \approx n^{1.58}.$$

Beispiel: 1000-stellige Zahl:  $1000^2/1000^{1.58} \approx 18$ .

# Bestmöglicher Algorithmus?

Wir kennen nun eine obere Schranke  $n^{\log_2 3}$ .

Es gibt praktisch (für grosses  $n$ ) relevante, schnellere Algorithmen.

Beispiel: Schönhage-Strassen-Algorithmus (1971) basierend auf schneller Fouriertransformation mit Laufzeit  $\mathcal{O}(n \log n \cdot \log \log n)$ .

Die beste obere Schranke ist nicht bekannt.

Untere Schranke:  $n$ . Jede Ziffer muss zumindest einmal angeschaut werden.



# Anhang: Asymptotik mit Additionen und Shifts

Bei jeder Multiplikation zweier  $n$ -stelliger Zahlen kommt auch noch eine konstante Anzahl Additionen, Subtraktionen und Shifts dazu

Additionen, Subtraktionen und Shifts von  $n$  stelligen Zahlen kosten  $\mathcal{O}(n)$

Daher ist die asymptotische Laufzeit eigentlich (mit geeignetem  $c > 1$ ) bestimmt durch die folgende Rekurrenz

$$T(n) = \begin{cases} 3 \cdot T\left(\frac{1}{2}n\right) + c \cdot n & \text{falls } n > 1 \\ 1 & \text{sonst} \end{cases}$$

# Anhang: Asymptotik mit Additionen und Shifts

Annahme:  $n = 2^k$ ,  $k > 0$

$$\begin{aligned}T(2^k) &= 3 \cdot T(2^{k-1}) + c \cdot 2^k \\&= 3 \cdot (3 \cdot T(2^{k-2}) + c \cdot 2^{k-1}) + c \cdot 2^k \\&= 3 \cdot (3 \cdot (3 \cdot T(2^{k-3}) + c \cdot 2^{k-2}) + c \cdot 2^{k-1}) + c \cdot 2^k \\&= 3 \cdot (3 \cdot (\dots(3 \cdot T(2^{k-k}) + c \cdot 2^1)\dots) + c \cdot 2^{k-1}) + c \cdot 2^k \\&= 3^k \cdot T(1) + c \cdot 3^{k-1}2^1 + c \cdot 3^{k-2}2^2 + \dots + c \cdot 3^02^k \\&\leq c \cdot 3^k \cdot (1 + 2/3 + (2/3)^2 + \dots + (2/3)^k)\end{aligned}$$

Die geometrische Reihe  $\sum_{i=0}^k \varrho^i$  mit  $\varrho = 2/3$  konvergiert für  $k \rightarrow \infty$  gegen  $\frac{1}{1-\varrho} = 3$ .

Somit  $T(2^k) \leq c \cdot 3^k \cdot 3 \in \Theta(3^k) = \Theta(3^{\log_2 n}) = \Theta(n^{\log_2 3})$ .

## 3.3 Maximum Subarray Problem

Algorithmenentwurf – Maximum Subarray Problem [Ottman/Widmayer, Kap. 1.3]  
Divide and Conquer [Ottman/Widmayer, Kap. 1.2.2. S.9; Cormen et al, Kap. 4-4.1]

# Algorithmenentwurf

Induktive Entwicklung eines Algorithmus: Zerlegung in Teilprobleme, Verwendung der Lösungen der Teilproblem zum Finden der endgültigen Lösung.

*Ziel:* Entwicklung des asymptotisch effizientesten (korrekten) Algorithmus.

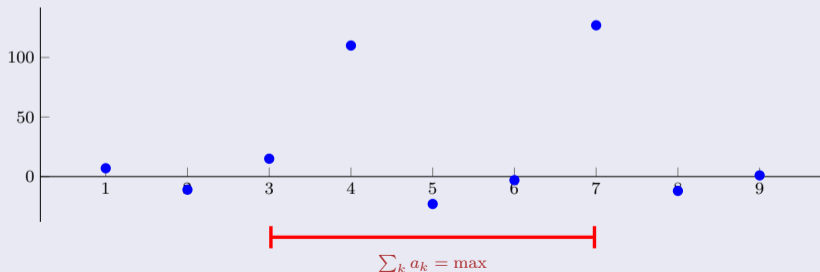
*Effizienz* hinsichtlich der Laufzeitkosten (# Elementaroperationen) oder / und Speicherbedarf.

# Maximum Subarray Problem

*Gegeben:* ein Array von  $n$  reellen Zahlen  $(a_1, \dots, a_n)$ .

*Gesucht:* Teilstück  $[i, j]$ ,  $1 \leq i \leq j \leq n$  mit maximaler positiver Summe  $\sum_{k=i}^j a_k$ .

Beispiel:  $a = (7, -11, 15, 110, -23, -3, 127, -12, 1)$



# Naiver Maximum Subarray Algorithmus

**Input:** Eine Folge von  $n$  Zahlen  $(a_1, a_2, \dots, a_n)$

**Output:**  $I, J$  mit  $\sum_{k=I}^J a_k$  maximal.

$M \leftarrow 0; I \leftarrow 1; J \leftarrow 0$

**for**  $i \in \{1, \dots, n\}$  **do**

**for**  $j \in \{i, \dots, n\}$  **do**

$m = \sum_{k=i}^j a_k$

**if**  $m > M$  **then**

$M \leftarrow m; I \leftarrow i; J \leftarrow j$

**return**  $I, J$

# Analyse

## Theorem

*Der naive Algorithmus für das Maximum Subarray Problem führt  $\Theta(n^3)$  Additionen durch.*

Beweis:

$$\begin{aligned}\sum_{i=1}^n \sum_{j=i}^n (j - i + 1) &= \sum_{i=1}^n \sum_{j=0}^{n-i} (j + 1) = \sum_{i=1}^n \sum_{j=1}^{n-i+1} j = \sum_{i=1}^n \frac{(n - i + 1)(n - i + 2)}{2} \\ &= \sum_{i=0}^n \frac{i \cdot (i + 1)}{2} = \frac{1}{2} \left( \sum_{i=1}^n i^2 + \sum_{i=1}^n i \right) \\ &= \frac{1}{2} \left( \frac{n(2n + 1)(n + 1)}{6} + \frac{n(n + 1)}{2} \right) = \frac{n^3 + 3n^2 + 2n}{6} = \Theta(n^3).\end{aligned}$$



# Beobachtung

$$\sum_{k=i}^j a_k = \underbrace{\left( \sum_{k=1}^j a_k \right)}_{S_j} - \underbrace{\left( \sum_{k=1}^{i-1} a_k \right)}_{S_{i-1}}$$

*Präfixsummen*

$$S_i := \sum_{k=1}^i a_k.$$



# Maximum Subarray Algorithmus mit Präfixsummen

**Input:** Eine Folge von  $n$  Zahlen  $(a_1, a_2, \dots, a_n)$

**Output:**  $I, J$  mit  $\sum_{k=I}^J a_k$  maximal.

$S_0 \leftarrow 0$

**for**  $i \in \{1, \dots, n\}$  **do** // Präfixsumme

└  $S_i \leftarrow S_{i-1} + a_i$

$M \leftarrow 0; I \leftarrow 1; J \leftarrow 0$

**for**  $i \in \{1, \dots, n\}$  **do**

└ **for**  $j \in \{i, \dots, n\}$  **do**

└└  $m = S_j - S_{i-1}$

└└ **if**  $m > M$  **then**

└└└  $M \leftarrow m; I \leftarrow i; J \leftarrow j$

# Analyse

## Theorem

*Der Präfixsummen Algorithmus für das Maximum Subarray Problem führt  $\Theta(n^2)$  Additionen und Subtraktionen durch.*

Beweis:

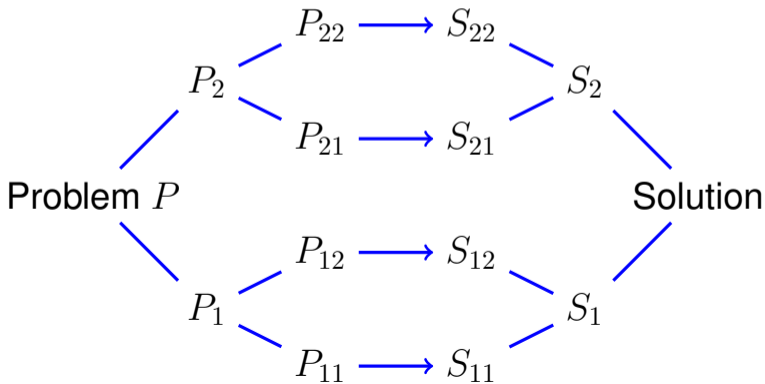
$$\sum_{i=1}^n 1 + \sum_{i=1}^n \sum_{j=i}^n 1 = n + \sum_{i=1}^n (n - i + 1) = n + \sum_{i=1}^n i = \Theta(n^2)$$



# divide et impera

Teile und (be)herrsche (engl. divide and conquer)

Zerlege das Problem in Teilprobleme, deren Lösung zur vereinfachten Lösung des Gesamtproblems beitragen.



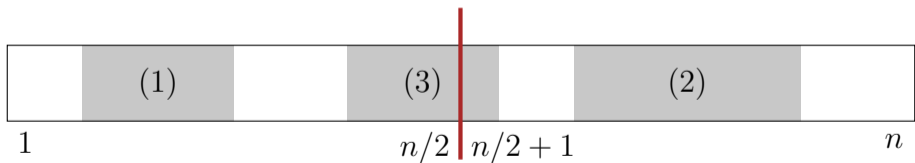
# Maximum Subarray – Divide

- Divide: Teile das Problem in zwei (annähernd) gleiche Hälften auf:  
 $(a_1, \dots, a_n) = (a_1, \dots, a_{\lfloor n/2 \rfloor}, a_{\lfloor n/2 \rfloor + 1}, \dots, a_n)$
- Vereinfachende Annahme:  $n = 2^k$  für ein  $k \in \mathbb{N}$ .

# Maximum Subarray – Conquer

Sind  $i, j$  die Indizes einer Lösung  $\Rightarrow$  Fallunterscheidung:

- 1 Lösung in linker Hälfte  $1 \leq i \leq j \leq n/2 \Rightarrow$  Rekursion (linke Hälfte)
- 2 Lösung in rechter Hälfte  $n/2 < i \leq j \leq n \Rightarrow$  Rekursion (rechte Hälfte)
- 3 Lösung in der Mitte  $1 \leq i \leq n/2 < j \leq n \Rightarrow$  Nachfolgende Beobachtung



# Maximum Subarray – Beobachtung

Annahme: Lösung in der Mitte  $1 \leq i \leq n/2 < j \leq n$

$$\begin{aligned} S_{\max} &= \max_{\substack{1 \leq i \leq n/2 \\ n/2 < j \leq n}} \sum_{k=i}^j a_k = \max_{\substack{1 \leq i \leq n/2 \\ n/2 < j \leq n}} \left( \sum_{k=i}^{n/2} a_k + \sum_{k=n/2+1}^j a_k \right) \\ &= \max_{1 \leq i \leq n/2} \sum_{k=i}^{n/2} a_k + \max_{n/2 < j \leq n} \sum_{k=n/2+1}^j a_k \\ &= \max_{1 \leq i \leq n/2} \underbrace{S_{n/2} - S_{i-1}}_{\text{Suffixsumme}} + \max_{n/2 < j \leq n} \underbrace{S_j - S_{n/2}}_{\text{Präfixsumme}} \end{aligned}$$

# Maximum Subarray Divide and Conquer Algorithmus

**Input:** Eine Folge von  $n$  Zahlen  $(a_1, a_2, \dots, a_n)$

**Output:** Maximales  $\sum_{k=i'}^{j'} a_k$ .

**if**  $n = 1$  **then**

**return**  $\max\{a_1, 0\}$

**else**

    Unterteile  $a = (a_1, \dots, a_n)$  in  $A_1 = (a_1, \dots, a_{n/2})$  und  
     $A_2 = (a_{n/2+1}, \dots, a_n)$

    Berechne rekursiv beste Lösung  $W_1$  in  $A_1$

    Berechne rekursiv beste Lösung  $W_2$  in  $A_2$

    Berechne grösste Suffixsumme  $S$  in  $A_1$

    Berechne grösste Präfixsumme  $P$  in  $A_2$

    Setze  $W_3 \leftarrow S + P$

**return**  $\max\{W_1, W_2, W_3\}$

## Theorem

*Der Divide and Conquer Algorithmus für das Maximum Subarray Sum Problem führt  $\Theta(n \log n)$  viele Additionen und Vergleiche durch.*



# Analyse

**Input:** Eine Folge von  $n$  Zahlen  $(a_1, a_2, \dots, a_n)$

**Output:** Maximales  $\sum_{k=i'}^{j'} a_k$ .

**if**  $n = 1$  **then**

$\Theta(1)$  **return**  $\max\{a_1, 0\}$

**else**

$\Theta(1)$  Unterteile  $a = (a_1, \dots, a_n)$  in  $A_1 = (a_1, \dots, a_{n/2})$  und  
 $A_2 = (a_{n/2+1}, \dots, a_n)$

$T(n/2)$  Berechne rekursiv beste Lösung  $W_1$  in  $A_1$

$T(n/2)$  Berechne rekursiv beste Lösung  $W_2$  in  $A_2$

$\Theta(n)$  Berechne grösste Suffixsumme  $S$  in  $A_1$

$\Theta(n)$  Berechne grösste Präfixsumme  $P$  in  $A_2$

$\Theta(1)$  Setze  $W_3 \leftarrow S + P$

$\Theta(1)$  **return**  $\max\{W_1, W_2, W_3\}$

# Analyse

Rekursionsgleichung

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ 2T(\frac{n}{2}) + a \cdot n & \text{falls } n > 1 \end{cases}$$

# Analyse

Mit  $n = 2^k$ :

$$\bar{T}(k) = \begin{cases} c & \text{falls } k = 0 \\ 2\bar{T}(k-1) + a \cdot 2^k & \text{falls } k > 0 \end{cases}$$

Lösung:

$$\bar{T}(k) = 2^k \cdot c + \sum_{i=0}^{k-1} 2^i \cdot a \cdot 2^{k-i} = c \cdot 2^k + a \cdot k \cdot 2^k = \Theta(k \cdot 2^k)$$

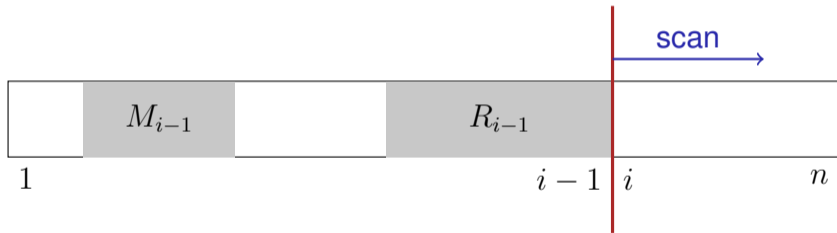
also

$$T(n) = \Theta(n \log n)$$



# Maximum Subarray Sum Problem – Induktiv

Annahme: Maximaler Wert  $M_{i-1}$  der Subarraysumme für  $(a_1, \dots, a_{i-1})$  ( $1 < i \leq n$ ) bekannt.



$a_i$ : erzeugt höchstens Intervall am Rand (Präfixsumme).

$$R_{i-1} \Rightarrow R_i = \max\{R_{i-1} + a_i, 0\}$$

# Induktiver Maximum Subarray Algorithmus

**Input:** Eine Folge von  $n$  Zahlen  $(a_1, a_2, \dots, a_n)$ .

**Output:**  $\max\{0, \max_{i,j} \sum_{k=i}^j a_k\}$ .

$M \leftarrow 0$

$R \leftarrow 0$

**for**  $i = 1 \dots n$  **do**

$R \leftarrow R + a_i$

**if**  $R < 0$  **then**

$R \leftarrow 0$

**if**  $R > M$  **then**

$M \leftarrow R$

**return**  $M$ ;

## Theorem

*Der induktive Algorithmus für das Maximum Subarray Sum Problem führt  $\Theta(n)$  viele Additionen und Vergleiche durch.*

# Komplexität des Problems?

Geht es besser als  $\Theta(n)$ ?

Jeder korrekte Algorithmus für das Maximum Subarray Sum Problem muss jedes Element im Algorithmus betrachten.

Annahme: der Algorithmus betrachtet nicht  $a_i$ .

- 1 Lösung des Algorithmus enthält  $a_i$ . Wiederholen den Algorithmus mit genügend kleinem  $a_i$ , so dass die Lösung den Punkt nicht enthalten hätte dürfen.
- 2 Lösung des Algorithmus enthält  $a_i$  nicht. Wiederholen den Algorithmus mit genügend grossem  $a_i$ , so dass die Lösung  $a_i$  hätten enthalten müssen.

# Komplexität des Maximum Subarray Sum Problems

## Theorem

*Das Maximum Subarray Sum Problem hat Komplexität  $\Theta(n)$ .*

Beweis: Induktiver Algorithmus mit asymptotischer Laufzeit  $\mathcal{O}(n)$ .

Jeder Algorithmus hat Laufzeit  $\Omega(n)$ .

Somit ist die Komplexität  $\Omega(n) \cap \mathcal{O}(n) = \Theta(n)$ . ■



## 3.4 Anhang

Herleitung einiger mathematischen Formeln

# Summen

$$\sum_{i=0}^n i^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

Trick:

$$\sum_{i=1}^n i^3 - (i-1)^3 = \sum_{i=0}^n i^3 - \sum_{i=0}^{n-1} i^3 = n^3$$

$$\sum_{i=1}^n i^3 - (i-1)^3 = \sum_{i=1}^n i^3 - i^3 + 3i^2 - 3i + 1 = n - \frac{3}{2}n \cdot (n+1) + 3 \sum_{i=0}^n i^2$$

$$\Rightarrow \sum_{i=0}^n i^2 = \frac{1}{6}(2n^3 + 3n^2 + n) \in \Theta(n^3)$$

Kann einfach verallgemeinert werden:  $\sum_{i=1}^n i^k \in \Theta(n^{k+1})$ .

# Geometrische Reihe

$$\sum_{i=0}^n \rho^i \stackrel{!}{=} \frac{1 - \rho^{n+1}}{1 - \rho}$$

$$\begin{aligned} \sum_{i=0}^n \rho^i \cdot (1 - \rho) &= \sum_{i=0}^n \rho^i - \sum_{i=0}^n \rho^{i+1} = \sum_{i=0}^n \rho^i - \sum_{i=1}^{n+1} \rho^i \\ &= \rho^0 - \rho^{n+1} = 1 - \rho^{n+1}. \end{aligned}$$

Für  $0 \leq \rho < 1$ :

$$\sum_{i=0}^{\infty} \rho^i = \frac{1}{1 - \rho}$$

# 4. Suchen

Lineare Suche, Binäre Suche, (Interpolationssuche,) Untere Schranken [Ottman/Widmayer, Kap. 3.2, Cormen et al, Kap. 2: Problems 2.1-3,2.2-3,2.3-5]

# Das Suchproblem

Gegeben

- Menge von Datensätzen.

## Beispiele

Telefonverzeichnis, Wörterbuch, Symboltabelle

- Jeder Datensatz hat einen Schlüssel  $k$ .
- Schlüssel sind vergleichbar: eindeutige Antwort auf Frage  $k_1 \leq k_2$  für Schlüssel  $k_1, k_2$ .

Aufgabe: finde Datensatz nach Schlüssel  $k$ .

# Suche in Array

Gegeben

- Array  $A$  mit  $n$  Elementen ( $A[1], \dots, A[n]$ ).
- Schlüssel  $b$

Gesucht: Index  $k$ ,  $1 \leq k \leq n$  mit  $A[k] = b$  oder "nicht gefunden".

22	20	32	10	35	24	42	38	28	41
1	2	3	4	5	6	7	8	9	10

# Lineare Suche

Durchlaufen des Arrays von  $A[1]$  bis  $A[n]$ .

- *Bestenfalls* 1 Vergleich.
- *Schlimmstenfalls*  $n$  Vergleiche.
- Annahme: Jede Anordnung der  $n$  Schlüssel ist gleichwahrscheinlich. *Erwartete* Anzahl Vergleiche für die erfolgreiche Suche:

$$\frac{1}{n} \sum_{i=1}^n i = \frac{n+1}{2}.$$

# Suche im sortierten Array

Gegeben

- Sortiertes Array  $A$  mit  $n$  Elementen  $(A[1], \dots, A[n])$  mit  $A[1] \leq A[2] \leq \dots \leq A[n]$ .
- Schlüssel  $b$

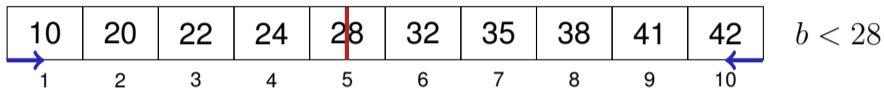
Gesucht: Index  $k$ ,  $1 \leq k \leq n$  mit  $A[k] = b$  oder "nicht gefunden".

10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10



# Divide and Conquer!

Suche  $b = 23$ .



# Binärer Suchalgorithmus

## BSearch( $A[l..r]$ , $b$ )

**Input:** Sortiertes Array  $A$  von  $n$  Schlüsseln. Schlüssel  $b$ . Bereichsgrenzen

$1 \leq l \leq r \leq n$  oder  $l > r$  beliebig.

**Output:** Index des gefundenen Elements. 0, wenn erfolglos.

$m \leftarrow \lfloor (l + r) / 2 \rfloor$

**if**  $l > r$  **then** // erfolglose Suche

**return** *NotFound*

**else if**  $b = A[m]$  **then** // gefunden

**return**  $m$

**else if**  $b < A[m]$  **then** // Element liegt links

**return** BSearch( $A[l..m - 1]$ ,  $b$ )

**else** //  $b > A[m]$ : Element liegt rechts

**return** BSearch( $A[m + 1..r]$ ,  $b$ )

# Analyse (schlechtester Fall)

Rekurrenz ( $n = 2^k$ )

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Teleskopieren:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c = \dots \\ &= T\left(\frac{n}{2^i}\right) + i \cdot c \\ &= T\left(\frac{n}{n}\right) + \log_2 n \cdot c = d + c \cdot \log_2 n \in \Theta(\log n) \end{aligned}$$

# Analyse (schlechtester Fall)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

**Vermutung** :  $T(n) = d + c \cdot \log_2 n$

**Beweis durch Induktion:**

- Induktionsanfang:  $T(1) = d$ .
- Hypothese:  $T(n/2) = d + c \cdot \log_2 n/2$
- Schritt ( $n/2 \rightarrow n$ )

$$T(n) = T(n/2) + c = d + c \cdot (\log_2 n - 1) + c = d + c \log_2 n.$$

## Theorem

*Der Algorithmus zur binären sortierten Suche benötigt  $\Theta(\log n)$  Elementarschritte.*

# Iterativer binärer Suchalgorithmus

**Input:** Sortiertes Array  $A$  von  $n$  Schlüsseln. Schlüssel  $b$ .

**Output:** Index des gefundenen Elements. 0, wenn erfolglos.

$l \leftarrow 1; r \leftarrow n$

**while**  $l \leq r$  **do**

$m \leftarrow \lfloor (l + r)/2 \rfloor$

**if**  $A[m] = b$  **then**

**return**  $m$

**else if**  $A[m] < b$  **then**

$l \leftarrow m + 1$

**else**

$r \leftarrow m - 1$

**return** *NotFound*;

# Korrektheit

Algorithmus bricht nur ab, falls  $A[l..r]$  leer oder  $b$  gefunden.

**Invariante:** Falls  $b$  in  $A$ , dann im Bereich  $A[l..r]$

## Beweis durch Induktion

- Induktionsanfang:  $b \in A[1..n]$  (oder nicht)
- Hypothese: Invariante gilt nach  $i$  Schritten
- Schritt:
  - $b < A[m] \Rightarrow b \in A[l..m - 1]$
  - $b > A[m] \Rightarrow b \in A[m + 1..r]$

# [Geht es noch besser?]

Annahme: Gleichverteilung der *Werte* im Array.

## Beispiel

Name "Becker" würde man im Telefonbuch vorne suchen.

"Wawrinka" wohl ziemlich weit hinten.

Binäre Suche vergleicht immer zuerst mit der Mitte.

Binäre Suche setzt immer  $m = \lfloor l + \frac{r-l}{2} \rfloor$ .



# [Interpolationssuche]

Erwartete relative Position von  $b$  im Suchintervall  $[l, r]$

$$\rho = \frac{b - A[l]}{A[r] - A[l]} \in [0, 1].$$

Neue "Mitte":  $l + \rho \cdot (r - l)$

Anzahl Vergleiche im Mittel  $\mathcal{O}(\log \log n)$  (ohne Beweis).

❓ Ist Interpolationssuche also immer zu bevorzugen?

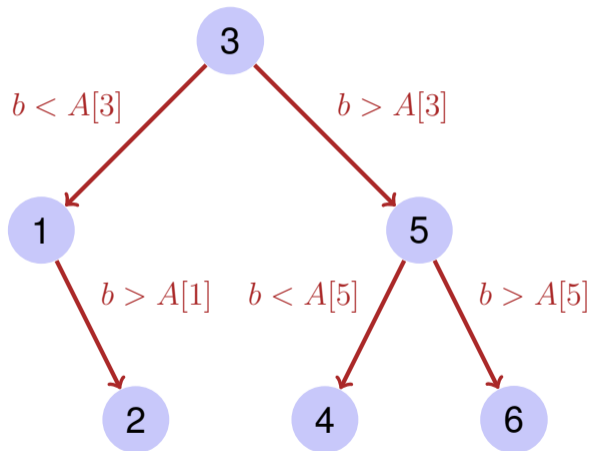
❗ Nein: Anzahl Vergleiche im schlimmsten Fall  $\Omega(n)$ .

# Untere Schranke

Binäre Suche (im schlechtesten Fall):  $\Theta(\log n)$  viele Vergleiche.

Gilt für *jeden* Suchalgorithmus in sortiertem Array (im schlechtesten Fall): Anzahl Vergleiche =  $\Omega(\log n)$ ?

# Entscheidungsbaum



- Für jede Eingabe  $b = A[i]$  muss Algorithmus erfolgreich sein  $\Rightarrow$  Baum enthält mindestens  $n$  Knoten.
- Anzahl Vergleiche im schlechtesten Fall = Höhe des Baumes = maximale Anzahl Knoten von Wurzel zu Blatt.

# Entscheidungsbaum

Binärer Baum der Höhe  $h$  hat höchstens  $2^0 + 2^1 + \dots + 2^{h-1} = 2^h - 1 < 2^h$  Knoten.

$$2^h > n \Rightarrow h > \log_2 n$$

Entscheidungsbaum mit  $n$  Knoten hat mindestens Höhe  $\log_2 n$ .

Anzahl Entscheidungen =  $\Omega(\log n)$ .

## Theorem

*Jeder Algorithmus zur vergleichsbasierten Suche in sortierten Daten der Länge  $n$  benötigt im schlechtesten Fall  $\Omega(\log n)$  Vergleichsschritte.*

# Untere Schranke für Suchen in unsortiertem Array

## Theorem

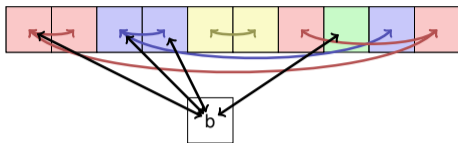
*Jeder vergleichsbasierte Algorithmus zur Suche in **un**sortierten Daten der Länge  $n$  benötigt im schlechtesten Fall  $\Omega(n)$  Vergleichsschritte.*

❓ Korrekt?

”Beweis”: Um  $b$  in  $A$  zu finden, muss  $b$  mit jedem Element  $A[i]$  ( $1 \leq i \leq n$ ) verglichen werden.

❗ Falsch! Vergleiche zwischen Elementen von  $A$  möglich!

# Besseres Argument



- Unterteilung der Vergleiche: Anzahl Vergleiche mit  $b$  :  $e$  Anzahl Vergleiche untereinander ohne  $b$ :  $i$
- Vergleiche erzeugen  $g$  Gruppen. Initial:  $g = n$ .
- Vereinigen zweier Gruppen benötigt mindestens einen (internen Vergleich):  $n - g \leq i$ .
- Mindestens ein Element pro Gruppe muss mit  $b$  verglichen werden:  $e \geq g$ .
- Anzahl Vergleiche  $i + e \geq n - g + g = n$ .

# 5. Auswählen

Das Auswahlproblem, Randomisierte Berechnung des Medians,  
Lineare Worst-Case Auswahl [Ottman/Widmayer, Kap. 3.1, Cormen  
et al, Kap. 9]



# Das Auswahlproblem

Eingabe

- Unsortiertes Array  $A = (A_1, \dots, A_n)$  paarweise verschiedener Werte
- Zahl  $1 \leq k \leq n$ .

Ausgabe:  $A[i]$  mit  $|\{j : A[j] < A[i]\}| = k - 1$

## Spezialfälle

$k = 1$ : Minimum: Algorithmus mit  $n$  Vergleichsoperationen trivial.

$k = n$ : Maximum: Algorithmus mit  $n$  Vergleichsoperationen trivial.

$k = \lfloor n/2 \rfloor$ : Median.

# Naiver Algorithmus

Wiederholt das Minimum entfernen / auslesen:  $\Theta(k \cdot n)$ .  
→ Median in  $\Theta(n^2)$

# Min und Max

② Separates Finden von Minimum und Maximum in  $(A[1], \dots, A[n])$  benötigt insgesamt  $2n$  Vergleiche. (Wie) geht es mit weniger als  $2n$  Vergleichen für beide gemeinsam?

① Es geht mit  $\frac{3}{2}n$  Vergleichen: Vergleiche jeweils 2 Elemente und deren kleineres mit Min und grösseres mit Max.<sup>4</sup> Possible with  $\frac{3}{2}n$  comparisons: compare 2 elements each and then the smaller one with min and the greater one with max.<sup>5</sup>

---

<sup>4</sup>Das liefert einen Hinweis darauf, dass der naive Algorithmus verbessert werden kann.

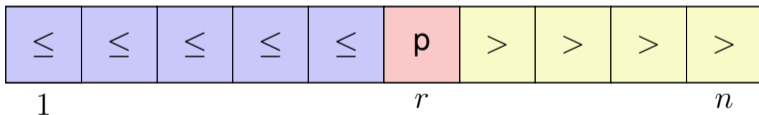
<sup>5</sup>An indication that the naive algorithm can be improved.

# Bessere Ansätze

- Sortieren (kommt bald):  $\Theta(n \log n)$
- Pivotieren:  $\Theta(n)$  !

# Pivotieren

- 1 Wähle ein (beliebiges) Element  $p$  als Pivotelement
- 2 Teile  $A$  in zwei Teile auf, bestimme dabei den Rang von  $p$ , indem die Anzahl der Indizes  $i$  mit  $A[i] \leq p$  gezählt werden.
- 3 Rekursion auf dem relevanten Teil. Falls  $k = r$ , dann gefunden.



# Algorithmus Partition( $A[l..r], p$ )

**Input:** Array  $A$ , welches den Pivot  $p$  im Intervall  $[l, r]$  mindestens einmal enthält.

**Output:** Array  $A$  partitioniert in  $[l..r]$  um  $p$ . Rückgabe der Position von  $p$ .

**while**  $l \leq r$  **do**

**while**  $A[l] < p$  **do**

$l \leftarrow l + 1$

**while**  $A[r] > p$  **do**

$r \leftarrow r - 1$

    swap( $A[l], A[r]$ )

**if**  $A[l] = A[r]$  **then**

$l \leftarrow l + 1$

**return**  $l-1$

# Korrektheit: Invariante

Invariante  $I$ :  $A_i \leq p \forall i \in [0, l), A_i \geq p \forall i \in (r, n], \exists k \in [l, r] : A_k = p$ .

**while**  $l \leq r$  **do**

**while**  $A[l] < p$  **do**

$l \leftarrow l + 1$

**while**  $A[r] > p$  **do**

$r \leftarrow r - 1$

$\text{swap}(A[l], A[r])$

**if**  $A[l] = A[r]$  **then**

$l \leftarrow l + 1$

$I$

$I$  und  $A[l] \geq p$

$I$  und  $A[r] \leq p$

$I$  und  $A[l] \leq p \leq A[r]$

$I$

**return**  $l-1$

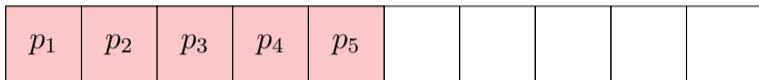
# Korrektheit: Fortschritt

```
while  $l \leq r$  do  
  while  $A[l] < p$  do           Fortschritt wenn  $A[l] < p$   
     $l \leftarrow l + 1$   
  while  $A[r] > p$  do           Fortschritt wenn  $A[r] > p$   
     $r \leftarrow r - 1$   
  swap( $A[l], A[r]$ )              Fortschritt wenn  $A[l] > p$  oder  $A[r] < p$   
  if  $A[l] = A[r]$  then          Fortschritt wenn  $A[l] = A[r] = p$   
     $l \leftarrow l + 1$   
return  $l-1$ 
```

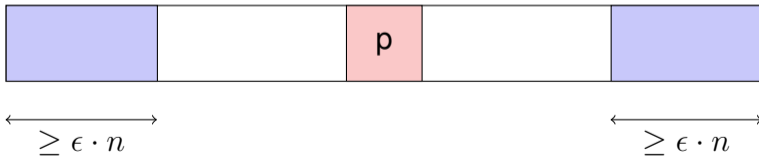


# Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case  $\Theta(n^2)$



Ein guter Pivot hat linear viele Elemente auf beiden Seiten.



# Analyse

Unterteilung mit Faktor  $q$  ( $0 < q < 1$ ): zwei Gruppen mit  $q \cdot n$  und  $(1 - q) \cdot n$  Elementen (ohne Einschränkung  $q \geq 1 - q$ ).

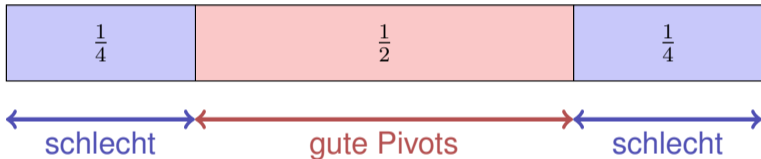
$$T(n) \leq T(q \cdot n) + c \cdot n$$

$$= c \cdot n + q \cdot c \cdot n + T(q^2 \cdot n) = \dots = c \cdot n \sum_{i=0}^{\log_q(n)-1} q^i + T(1)$$

$$\leq c \cdot n \underbrace{\sum_{i=0}^{\infty} q^i}_{\text{geom. Reihe}} + d = c \cdot n \cdot \frac{1}{1 - q} + d = \mathcal{O}(n)$$

# Wie bekommen wir das hin?

Der Zufall hilft uns (Tony Hoare, 1961). Wähle in jedem Schritt einen zufälligen Pivot.



Wahrscheinlichkeit für guten Pivot nach einem Versuch:  $\frac{1}{2} =: \rho$ .

Wahrscheinlichkeit für guten Pivot nach  $k$  Versuchen:  $(1 - \rho)^{k-1} \cdot \rho$ .

Erwartete Anzahl Versuche:  $1/\rho = 2$  (Erwartungswert der geometrischen Verteilung:)

# Algorithmus Quickselect ( $A[l..r], k$ )

**Input:** Array  $A$  der Länge  $n$ . Indizes  $1 \leq l \leq k \leq r \leq n$ , so dass für alle  $x \in A[l..r] : |\{j | A[j] \leq x\}| \geq l$  und  $|\{j | A[j] \leq x\}| \leq r$ .

**Output:** Wert  $x \in A[l..r]$  mit  $|\{j | A[j] \leq x\}| \geq k$  und  $|\{j | x \leq A[j]\}| \geq n - k + 1$

**if**  $l=r$  **then**

  | return  $A[l]$ ;

$x \leftarrow \text{RandomPivot}(A[l..r])$

$m \leftarrow \text{Partition}(A[l..r], x)$

**if**  $k < m$  **then**

  | return QuickSelect( $A[l..m - 1], k$ )

**else if**  $k > m$  **then**

  | return QuickSelect( $A[m + 1..r], k$ )

**else**

  | **return**  $A[k]$

# Algorithmus RandomPivot ( $A[l..r]$ )

**Input:** Array  $A$  der Länge  $n$ . Indizes  $1 \leq l \leq i \leq r \leq n$

**Output:** Zufälliger “guter” Pivot  $x \in A[l..r]$

**repeat**

    wähle zufälligen Pivot  $x \in A[l..r]$

$p \leftarrow l$

**for**  $j = l$  **to**  $r$  **do**

**if**  $A[j] \leq x$  **then**  $p \leftarrow p + 1$

**until**  $\lfloor \frac{3l+r}{4} \rfloor \leq p \leq \lceil \frac{l+3r}{4} \rceil$

**return**  $x$

*Dieser Algorithmus ist nur von theoretischem Interesse und liefert im Erwartungswert nach 2 Durchläufen einen guten Pivot. Praktisch kann man im Algorithmus Quickselect direkt einen zufälligen Pivot uniformverteilt ziehen oder einen deterministischen Pivot wählen, z.B. den Median von drei Elementen.*

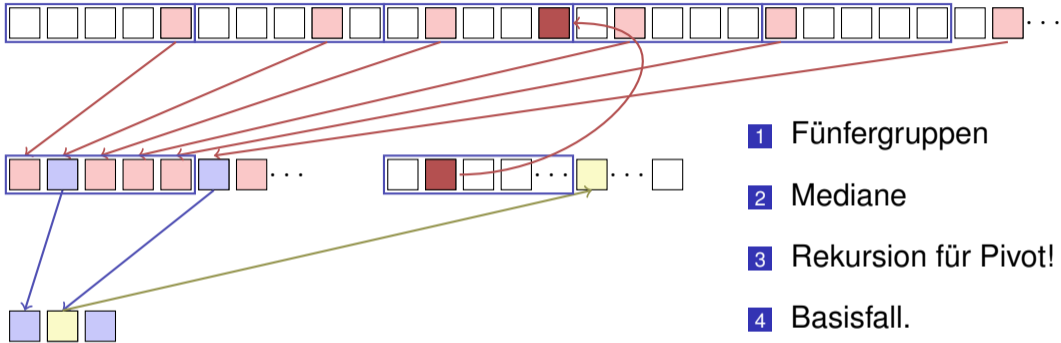
# Median der Mediane

Ziel: Finde einen Algorithmus, welcher im schlechtesten Fall nur linear viele Schritte benötigt.

Algorithmus Select ( $k$ -smallest)

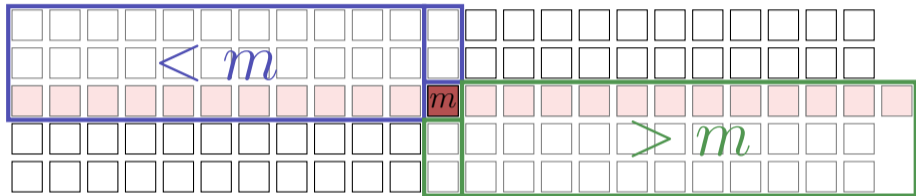
- Fünfergruppen bilden.
- Median jeder Gruppe bilden (naiv).
- Select rekursiv auf den Gruppenmedianen.
- Partitioniere das Array um den gefundenen Median der Mediane.  
Resultat:  $i$
- Wenn  $i = k$ , Resultat. Sonst: Select rekursiv auf der richtigen Seite.

# Median der Mediane



- 1 Fünfergruppen
- 2 Mediane
- 3 Rekursion für Pivot!
- 4 Basisfall.
- 5 Pivot (Level 1)
- 6 Partition (Level 1)
- 7 Median = Pivot Level 0
- 8 2. Rekursion startet

# Was bringt das?



Anzahl Punkte links / rechts vom Median der Mediane (ohne Mediengruppe und ohne Restgruppe)  $\geq 3 \cdot (\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2) \geq \frac{3n}{10} - 6$

Zweiter Aufruf mit maximal  $\lceil \frac{7n}{10} + 6 \rceil$  Elementen.



# Analyse

Rekursionsungleichung:

$$T(n) \leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\left\lceil \frac{7n}{10} + 6 \right\rceil\right) + d \cdot n.$$

mit einer Konstanten  $d$ .

Behauptung:

$$T(n) = \mathcal{O}(n).$$

# Beweis

Induktionsanfang: Wähle  $c$  so gross, dass

$$T(n) \leq c \cdot n \text{ für alle } n \leq n_0.$$

Induktionsannahme:

$$T(i) \leq c \cdot i \text{ für alle } i < n.$$

Induktionsschritt:

$$\begin{aligned} T(n) &\leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\left\lceil \frac{7n}{10} + 6 \right\rceil\right) + d \cdot n \\ &= c \cdot \left\lceil \frac{n}{5} \right\rceil + c \cdot \left\lceil \frac{7n}{10} + 6 \right\rceil + d \cdot n. \end{aligned}$$

# Beweis

Induktionsschritt:

$$\begin{aligned} T(n) &\leq c \cdot \left\lceil \frac{n}{5} \right\rceil + c \cdot \left\lceil \frac{7n}{10} + 6 \right\rceil + d \cdot n \\ &\leq c \cdot \frac{n}{5} + c + c \cdot \frac{7n}{10} + 6c + c + d \cdot n = \frac{9}{10} \cdot c \cdot n + 8c + d \cdot n. \end{aligned}$$

Wähle  $c \geq 80 \cdot d$  und  $n_0 = 91$ .

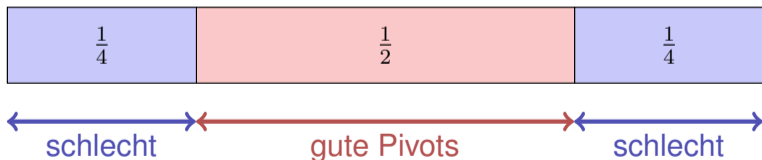
$$T(n) \leq \frac{72}{80} \cdot c \cdot n + 8c + \frac{1}{80} \cdot c \cdot n = c \cdot \underbrace{\left( \frac{73}{80}n + 8 \right)}_{\leq n \text{ für } n > n_0} \leq c \cdot n.$$

## Theorem

*Das  $i$ -te Element einer Folge von  $n$  Elementen kann im schlechtesten Fall in  $\Theta(n)$  Schritten gefunden werden.*

# Überblick

- |                                     |                                      |
|-------------------------------------|--------------------------------------|
| 1. Wiederholt Minimum finden        | $\mathcal{O}(n^2)$                   |
| 2. Sortieren und $A[i]$ ausgeben    | $\mathcal{O}(n \log n)$              |
| 3. Quickselect mit zufälligem Pivot | $\mathcal{O}(n)$ im Mittel           |
| 4. Median of Medians (Blum)         | $\mathcal{O}(n)$ im schlimmsten Fall |



# 5.1 Anhang

Herleitung einiger mathematischen Formeln

# [Erwartungswert der geometrischen Verteilung]

Zufallsvariable  $X \in \mathbb{N}^+$  mit  $\mathbb{P}(X = k) = (1 - p)^{k-1} \cdot p$ .

Erwartungswert

$$\begin{aligned}\mathbb{E}(X) &= \sum_{k=1}^{\infty} k \cdot (1 - p)^{k-1} \cdot p = \sum_{k=1}^{\infty} k \cdot q^{k-1} \cdot (1 - q) \\ &= \sum_{k=1}^{\infty} k \cdot q^{k-1} - k \cdot q^k = \sum_{k=0}^{\infty} (k + 1) \cdot q^k - k \cdot q^k \\ &= \sum_{k=0}^{\infty} q^k = \frac{1}{1 - q} = \frac{1}{p}.\end{aligned}$$

## 6. C++ vertieft (I)

Kurzwiederholung: Vektoren, Zeiger und Iteratoren

Bereichsbasiertes for, Schlüsselwort auto, eine Klasse für Vektoren, Indexoperator, Move-Konstruktion, Iterator.



# Was lernen wir heute?

- Schlüsselwort `auto`
- Bereichsbasiertes `for`
- Kurzwiederholung der Dreierregel
- Indexoperator
- Move Semantik, X-Werte und Fünferregel
- Eigene Iteratoren

# Wir erinnern uns...

```
#include <iostream>
#include <vector>
using iterator = std::vector<int>::iterator;
```

```
int main(){
    // Vector of length 10
    std::vector<int> v(10);
    // Input
    for (int i = 0; i < v.size(); ++i)
        std::cin >> v[i];
    // Output
    for (iterator it = v.begin(); it != v.end(); ++it)
        std::cout << *it << " ";
}
```

Das wollen wir genau verstehen!

Das sieht umständlich aus!

# Nützliche Tools (1): `auto` (C++11)

Das Schlüsselwort `auto`:

Der Typ einer Variablen wird inferiert vom Initialisierer.

## Beispiele

```
int x = 10;  
auto y = x; // int  
auto z = 3; // int  
std::vector<double> v(5);  
auto i = v[3]; // double
```

# Schon etwas besser...

```
#include <iostream>
#include <vector>

int main(){
    std::vector<int> v(10); // Vector of length 10

    for (int i = 0; i < v.size(); ++i)
        std::cin >> v[i];

    for (auto it = v.begin(); it != v.end(); ++it){
        std::cout << *it << " ";
    }
}
```

## Nützliche Tools (2): Bereichsbasiertes `for` (C++11)

```
for (range-declaration : range-expression)
    statement;
```

*range-declaration*: benannte Variable vom Elementtyp der durch range-expression spezifizierten Folge.

*range-expression*: Ausdruck, der eine Folge von Elementen repräsentiert via Iterator-Paar `begin()`, `end()` oder in Form einer Initialisierungsliste.

### Beispiele

```
std::vector<double> v(5);
for (double x: v) std::cout << x; // 00000
for (int x: {1,2,5}) std::cout << x; // 125
for (double& x: v) x=5;
```

# Ok, das ist cool!

```
#include <iostream>
#include <vector>

int main(){
    std::vector<int> v(10); // Vector of length 10

    for (auto& x: v)
        std::cin >> x;

    for (const auto x: v)
        std::cout << x << " ";
}
```

# Für unser genaues Verständnis

*Wir bauen selbst eine Vektorklasse, die so etwas kann!*

Auf dem Weg lernen wir etwas über

- RAI (Resource Acquisition is Initialization) und Move-Konstruktion
- Index-Operatoren und andere Nützlichkeiten
- Templates
- Exception Handling
- Funktoren und Lambda-Ausdrücke

# Eine Klasse für (double) Vektoren

```
class Vector{
public:
    // constructors
    Vector(): sz{0}, elem{nullptr} {};
    Vector(std::size_t s): sz{s}, elem{new double[s]} {}
    // destructor
    ~Vector(){
        delete[] elem;
    }
    // (something is missing here)
private:
    std::size_t sz;
    double* elem;
}
```



# Elementzugriffe

```
class Vector{
    ...
    // getter. pre: 0 <= i < sz;
    double get(std::size_t i) const{
        return elem[i];
    }
    // setter. pre: 0 <= i < sz;
    void set(std::size_t i, double d){
        elem[i] = d;
    }
    // size property
    std::size_t size() const {
        return sz;
    }
}
```

```
class Vector{
public:
    Vector();
    Vector(std::size_t s);
    ~Vector();
    double get(std::size_t i) const;
    void set(std::size_t i, double d);
    std::size_t size() const;
}
```

(Vector Schnittstelle)

# Was läuft schief?

```
int main(){
    Vector v(32);
    for (std::size_t i = 0; i!=v.size(); ++i)
        v.set(i, i);
    Vector w = v;
    for (std::size_t i = 0; i!=w.size(); ++i)
        w.set(i, i*i);
    return 0;
}
```

```
class Vector{
public:
    Vector();
    Vector(std::size_t s);
    ~Vector();
    double get(std::size_t i) const;
    void set(std::size_t i, double d);
    std::size_t size() const;
}
```

(Vector Schnittstelle)

```
*** Error in 'vector1': double free or corruption
(!prev): 0x0000000000d23c20 ***
===== Backtrace: =====
/lib/x86_64-linux-gnu/libc.so.6(+0x777e5) [0x7fe5a5ac97e5]
```

# Rule of Three!

```
class Vector{
...
public:
// copy constructor
Vector(const Vector &v)
    : sz{v.sz}, elem{new double[v.sz]} {
    std::copy(v.elem, v.elem + v.sz, elem);
}
}
```

```
class Vector{
public:
    Vector();
    Vector(std::size_t s);
    ~Vector();
    Vector(const Vector &v);
    double get(std::size_t i) const;
    void set(std::size_t i, double d);
    std::size_t size() const;
}
```

(Vector Schnittstelle)

# Rule of Three!

```
class Vector{
...
// assignment operator
Vector& operator=(const Vector& v){
    if (v.elem == elem) return *this;
    if (elem != nullptr) delete[] elem;
    sz = v.sz;
    elem = new double[sz];
    std::copy(v.elem, v.elem+v.sz, elem);
    return *this;
}
}
```

```
class Vector{
public:
    Vector();
    Vector(std::size_t s);
    ~Vector();
    Vector(const Vector& v);
    Vector& operator=(const Vector& v);
    double get(std::size_t i) const;
    void set(std::size_t i, double d);
    std::size_t size() const;
}
```

(Vector Schnittstelle)

Jetzt ist es zumindest korrekt. Aber umständlich.

## Eleganter geht so (Teil 1):

```
public:  
// copy constructor  
// (with constructor delegation)  
Vector(const Vector &v): Vector(v.sz)  
{  
    std::copy(v.elem, v.elem + v.sz, elem);  
}
```

# Eleganter geht so (Teil 2):

```
class Vector{
...
    // Assignment operator
    Vector& operator= (const Vector&v){
        Vector cpy(v);
        swap(cpy);
        return *this;
    }
private:
    // helper function
    void swap(Vector& v){
        std::swap(sz, v.sz);
        std::swap(elem, v.elem);
    }
}
```

copy-and-swap idiom: alle Felder von `*this` tauschen mit den Daten von `cpy`. Beim Verlassen von `operator=` wird `cpy` aufgeräumt (dekonstruiert), während die Kopie der Daten von `v` in `*this` verbleiben.

# Arbeit an der Fassade.

Getter und Setter unschön. Wir wollen einen Indexoperator.

Überladen! So?

```
class Vector{  
...  
    double operator[] (std::size_t pos) const{  
        return elem[pos];  
    }  
    void operator[] (std::size_t pos, double value){  
        elem[pos] = value;  
    }  
}
```

Nein!

# Referenztypen!

```
class Vector{
...
    // for non-const objects
    double& operator[] (std::size_t pos){
        return elem[pos]; // return by reference!
    }
    // for const objects
    const double& operator[] (std::size_t pos) const{
        return elem[pos];
    }
}
```



# Soweit, so gut.

```
int main(){
    Vector v(32); // constructor
    for (int i = 0; i<v.size(); ++i)
        v[i] = i; // subscript operator

    Vector w = v; // copy constructor
    for (int i = 0; i<w.size(); ++i)
        w[i] = i*i;

    const auto u = w;
    for (int i = 0; i<u.size(); ++i)
        std::cout << v[i] << ":" << u[i] << " "; // 0:0 1:1 2:4 ...
    return 0;
}
```

```
class Vector{
public:
    Vector();
    Vector(std::size_t s);
    ~Vector();
    Vector(const Vector &v);
    Vector& operator=(const Vector&v);
    const double& operator[] (std::size_t pos) const;
    double& operator[] (std::size_t pos);
    std::size_t size() const;
}
```

# Anzahl Kopien

Wie oft wird `v` kopiert?

```
Vector operator+ (const Vector& l, double r){
    Vector result (l); // Kopie von l nach result
    for (std::size_t i = 0; i < l.size(); ++i)
        result[i] = l[i] + r;
    return result; // Dekonstruktion von result nach Zuweisung
}

int main(){
    Vector v(16); // Allokation von elems[16]
    v = v + 1; // Kopie bei Zuweisung!
    return 0; // Dekonstruktion von v
}
```

`v` wird (mindestens) zwei Mal kopiert.

# Move-Konstruktor und Move-Zuweisung

```
class Vector{
...
    // move constructor
    Vector (Vector&& v): Vector() {
        swap(v);
    };
    // move assignment
    Vector& operator=(Vector&& v){
        swap(v);
        return *this;
    };
}
```

```
class Vector{
public:
    Vector();
    Vector(std::size_t s);
    ~Vector();
    Vector(const Vector &v);
    Vector& operator=(const Vector&v);
    Vector (Vector&& v);
    Vector& operator=(Vector&& v);
    const double& operator[] (std::size_t pos) const;
    double& operator[] (std::size_t pos);
    std::size_t size() const;
}
```

# Erklärung

Wenn das Quellobjekt einer Zuweisung direkt nach der Zuweisung nicht weiter existiert, dann kann der Compiler den Move-Zuweisungsoperator anstelle des Zuweisungsoperators einsetzen.<sup>6</sup> Damit wird eine potentiell teure Kopie vermieden. Anzahl der Kopien im vorigen Beispiel reduziert sich zu 1.

---

<sup>6</sup>Analoges gilt für den Kopier-Konstruktor und den Move-Konstruktor.

# Illustration zur Move-Semantik

```
// nonsense implementation of a "vector" for demonstration purposes
class Vec{
public:
    Vec () {
        std::cout << "default constructor\n";}
    Vec (const Vec&) {
        std::cout << "copy constructor\n";}
    Vec& operator = (const Vec&) {
        std::cout << "copy assignment\n"; return *this;}
    ~Vec() {}
};
```

# Wie viele Kopien?

```
Vec operator + (const Vec& a, const Vec& b){  
    Vec tmp = a;  
    // add b to tmp  
    return tmp;  
}  
  
int main (){  
    Vec f;  
    f = f + f + f + f;  
}
```

Ausgabe  
default constructor  
copy constructor  
copy constructor  
copy constructor  
copy assignment

4 Kopien des Vektors

# Illustration der Move-Semantik

```
// nonsense implementation of a "vector" for demonstration purposes
class Vec{
public:
    Vec () { std::cout << "default constructor\n";}
    Vec (const Vec&) { std::cout << "copy constructor\n";}
    Vec& operator = (const Vec&) {
        std::cout << "copy assignment\n"; return *this;}
    ~Vec() {}
    // new: move constructor and assignment
    Vec (Vec&&) {
        std::cout << "move constructor\n";}
    Vec& operator = (Vec&&) {
        std::cout << "move assignment\n"; return *this;}
};
```

# Wie viele Kopien?

```
Vec operator + (const Vec& a, const Vec& b){  
    Vec tmp = a;  
    // add b to tmp  
    return tmp;  
}  
  
int main (){  
    Vec f;  
    f = f + f + f + f;  
}
```

## Ausgabe

default constructor

copy constructor

copy constructor

copy constructor

move assignment

3 Kopien des Vektors



# Wie viele Kopien?

```
Vec operator + (Vec a, const Vec& b){  
    // add b to a  
    return a;  
}  
  
int main (){  
    Vec f;  
    f = f + f + f + f;  
}
```

## Ausgabe

default constructor

copy constructor

move constructor

move constructor

move constructor

move assignment

1 Kopie des Vektors

**Erklärung:** Move-Semantik kommt zum Einsatz, wenn ein x-wert (expired) zugewiesen wird. R-Wert-Rückgaben von Funktionen sind x-Werte.

# Wie viele Kopien

```
void swap(Vec& a, Vec& b){  
    Vec tmp = a;  
    a=b;  
    b=tmp;  
}
```

```
int main (){  
    Vec f;  
    Vec g;  
    swap(f,g);  
}
```

## Ausgabe

default constructor  
default constructor  
copy constructor  
copy assignment  
copy assignment

3 Kopien des Vektors

# X-Werte erzwingen

```
void swap(Vec& a, Vec& b){  
    Vec tmp = std::move(a);  
    a=std::move(b);  
    b=std::move(tmp);  
}  
  
int main (){  
    Vec f;  
    Vec g;  
    swap(f,g);  
}
```

## Ausgabe

default constructor  
default constructor  
move constructor  
move assignment  
move assignment

0 Kopien des Vektors

**Erklärung:** Mit `std::move` kann man einen L-Wert Ausdruck zu einem X-Wert machen. Dann kommt wieder Move-Semantik zum Einsatz.

<http://en.cppreference.com/w/cpp/utility/move>

# `std::swap` & `std::move`

`std::swap` ist (mit Templates) genau wie oben gesehen implementiert

`std::move` kann verwendet werden, um die Elemente eines Containers in einen anderen zu verschieben

```
std::move(va.begin(), va.end(), vb.begin())
```

# Bereichsbasiertes `for`

Wir wollten doch das:

```
Vector v = ...;  
for (auto x: v)  
    std::cout << x << " ";
```

Dafür müssen wir einen Iterator über `begin` und `end` bereitstellen.

# Iterator für den Vektor

```
class Vector{  
...  
    // Iterator  
    double* begin(){  
        return elem;  
    }  
    double* end(){  
        return elem+sz;  
    }  
}
```

(Zeiger unterstützen Iteration)

```
class Vector{  
public:  
    Vector();  
    Vector(std::size_t s);  
    ~Vector();  
    Vector(const Vector &v);  
    Vector& operator=(const Vector&v);  
    Vector (Vector&& v);  
    Vector& operator=(Vector&& v);  
    const double& operator[] (std::size_t pos) const;  
    double& operator[] (std::size_t pos);  
    std::size_t size() const;  
    double* begin();  
    double* end();  
}
```

# Const Iterator für den Vektor

```
class Vector{  
...  
    // Const-Iterator  
    const double* begin() const{  
        return elem;  
    }  
    const double* end() const{  
        return elem+sz;  
    }  
}
```

```
class Vector{  
public:  
    Vector();  
    Vector(std::size_t s);  
    ~Vector();  
    Vector(const Vector &v);  
    Vector& operator=(const Vector&v);  
    Vector (Vector&& v);  
    Vector& operator=(Vector&& v);  
    const double& operator[] (std::size_t pos) const;  
    double& operator[] (std::size_t pos);  
    std::size_t size() const;  
    double* begin();  
    double* end();  
    const double* begin() const;  
    const double* end() const;  
}
```

# Zwischenstand

```
Vector Natural(int from, int to){
    Vector v(to-from+1);
    for (auto& x: v) x = from++;
    return v;
}

int main(){
    auto v = Natural(5,12);
    for (auto x: v)
        std::cout << x << " "; // 5 6 7 8 9 10 11 12
    std::cout << std::endl;
        << "sum = "
        << std::accumulate(v.begin(), v.end(),0); // sum = 68
    return 0;
}
```



# Heutige Zusammenfassung

- Benutze `auto` um Typen vom Initialisierer zu inferieren.
- X-Werte sind solche, bei denen der Compiler weiss, dass Sie ihre Gültigkeit verlieren.
- Benutze Move-Konstruktion, um X-Werte zu verschieben statt zu kopieren.
- Wenn man genau weiss, was man tut, kann man X-Werte auch erzwingen.
- Indexoperatoren können überladen werden. Zum Schreiben benutzt man Referenzen.
- Hinter bereichsbasiertem `for` wirkt ein Iterator.
- Iteration wird unterstützt, indem man einen Iterator nach Konvention der Standardbibliothek implementiert.

# 7. Sortieren I

Einfache Sortierverfahren

# 7.1 Einfaches Sortieren

Sortieren durch Auswahl, Sortieren durch Einfügen, Bubblesort

[Ottman/Widmayer, Kap. 2.1, Cormen et al, Kap. 2.1, 2.2, Exercise 2.2-2, Problem 2-2

# Problemstellung

**Eingabe:** Ein Array  $A = (A[1], \dots, A[n])$  der Länge  $n$ .

**Ausgabe:** Eine Permutation  $A'$  von  $A$ , die sortiert ist:  $A'[i] \leq A'[j]$  für alle  $1 \leq i \leq j \leq n$ .

# Algorithmus: IsSorted( $A$ )

**Input:** Array  $A = (A[1], \dots, A[n])$  der Länge  $n$ .

**Output:** Boolesche Entscheidung "sortiert" oder "nicht sortiert"

```
for  $i \leftarrow 1$  to  $n - 1$  do  
  if  $A[i] > A[i + 1]$  then  
    return "nicht sortiert";  
return "sortiert";
```

# Beobachtung

IsSorted( $A$ ):“nicht sortiert”, wenn  $A[i] > A[i + 1]$  für ein  $i$ .

⇒ Idee:

```
for  $j \leftarrow 1$  to  $n - 1$  do
  if  $A[j] > A[j + 1]$  then
    swap( $A[j], A[j + 1]$ );
```

# Ausprobieren

5 ↔ 6   2   8   4   1   ( $j = 1$ )

5   6 ↔ 2   8   4   1   ( $j = 2$ )

5   2   6 ↔ 8   4   1   ( $j = 3$ )

5   2   6   8 ↔ 4   1   ( $j = 4$ )

5   2   6   4   8 ↔ 1   ( $j = 5$ )

5   2   6   4   1   8

- Nicht sortiert! 😞.
- Aber das grösste Element wandert ganz nach rechts. ⇒ Neue Idee! 😊

# Ausprobieren

5	6	2	8	4	1	$(j = 1, i = 1)$
5	6	2	8	4	1	$(j = 2)$
5	2	6	8	4	1	$(j = 3)$
5	2	6	8	4	1	$(j = 4)$
5	2	6	4	8	1	$(j = 5)$
5	2	6	4	1	8	$(j = 1, i = 2)$
2	5	6	4	1	8	$(j = 2)$
2	5	6	4	1	8	$(j = 3)$
2	5	4	6	1	8	$(j = 4)$
2	5	4	1	6	8	$(j = 1, i = 3)$
2	5	4	1	6	8	$(j = 2)$
2	4	5	1	6	8	$(j = 3)$
2	4	1	5	6	8	$(j = 1, i = 4)$
2	4	1	5	6	8	$(j = 2)$
2	1	4	5	6	8	$(i = 1, j = 5)$
1	2	4	5	6	8	

- Wende das Verfahren iterativ an.
- Für  $A[1, \dots, n]$ ,  
dann  $A[1, \dots, n - 1]$ ,  
dann  $A[1, \dots, n - 2]$ ,  
etc.



# Algorithmus: Bubblesort

**Input:** Array  $A = (A[1], \dots, A[n])$ ,  $n \geq 0$ .

**Output:** Sortiertes Array  $A$

```
for  $i \leftarrow 1$  to  $n - 1$  do
  for  $j \leftarrow 1$  to  $n - i$  do
    if  $A[j] > A[j + 1]$  then
      swap( $A[j]$ ,  $A[j + 1]$ );
```

# Analyse

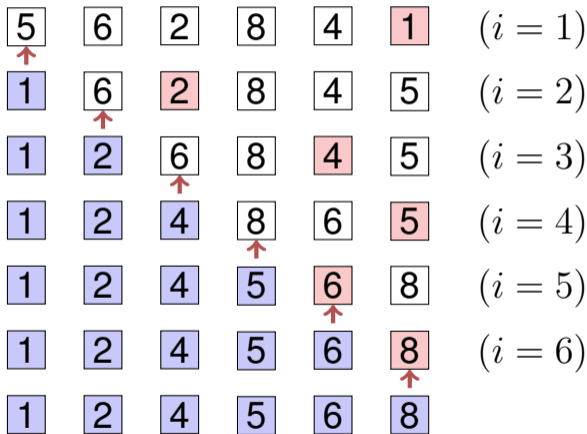
Anzahl Schlüsselvergleiche  $\sum_{i=1}^{n-1} (n - i) = \frac{n(n-1)}{2} = \Theta(n^2)$ .

Anzahl Vertauschungen im schlechtesten Fall:  $\Theta(n^2)$

❓ Was ist der schlechteste Fall?

⚠️ Wenn A absteigend sortiert ist.

# Sortieren durch Auswahl



- Auswahl des kleinsten Elementes durch Suche im unsortierten Teil  $A[i..n]$  des Arrays.
- Tausche kleinstes Element an das erste Element des unsortierten Teiles.
- Unsortierter Teil wird ein Element kleiner ( $i \rightarrow i + 1$ ). Wiederhole bis alles sortiert. ( $i = n$ )

# Algorithmus: Sortieren durch Auswahl

**Input:** Array  $A = (A[1], \dots, A[n])$ ,  $n \geq 0$ .

**Output:** Sortiertes Array  $A$

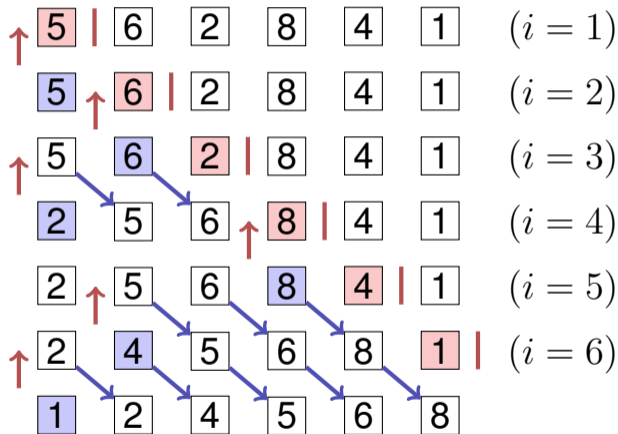
```
for  $i \leftarrow 1$  to  $n - 1$  do
   $p \leftarrow i$ 
  for  $j \leftarrow i + 1$  to  $n$  do
    if  $A[j] < A[p]$  then
       $p \leftarrow j$ 
  swap( $A[i]$ ,  $A[p]$ )
```

# Analyse

Anzahl Vergleiche im schlechtesten Fall:  $\Theta(n^2)$ .

Anzahl Vertauschungen im schlechtesten Fall:  $n - 1 = \Theta(n)$

# Sortieren durch Einfügen



- Iteratives Vorgehen:  
 $i = 1 \dots n$
- Einfügeposition für Element  $i$  bestimmen.
- Element  $i$  einfügen, ggfs. Verschiebung nötig.

# Sortieren durch Einfügen

❓ Welchen Nachteil hat der Algorithmus im Vergleich zum Sortieren durch Auswahl?

❗ Im schlechtesten Fall viele Elementverschiebungen.

❓ Welchen Vorteil hat der Algorithmus im Vergleich zum Sortieren durch Auswahl?

❗ Der Suchbereich (Einfügebereich) ist bereits sortiert.  
Konsequenz: binäre Suche möglich.

# Algorithmus: Sortieren durch Einfügen

**Input:** Array  $A = (A[1], \dots, A[n])$ ,  $n \geq 0$ .

**Output:** Sortiertes Array  $A$

**for**  $i \leftarrow 2$  **to**  $n$  **do**

$x \leftarrow A[i]$

$p \leftarrow \text{BinarySearch}(A[1..i-1], x)$ ; // Kleinstes  $p \in [1, i]$  mit  $A[p] \geq x$

**for**  $j \leftarrow i - 1$  **downto**  $p$  **do**

$A[j+1] \leftarrow A[j]$

$A[p] \leftarrow x$



# Analyse

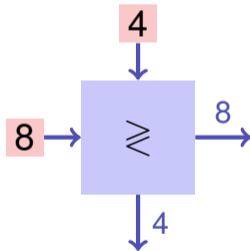
Anzahl Vergleiche im schlechtesten Fall:

$$\sum_{k=1}^{n-1} a \cdot \log k = a \log((n-1)!) \in \mathcal{O}(n \log n).$$

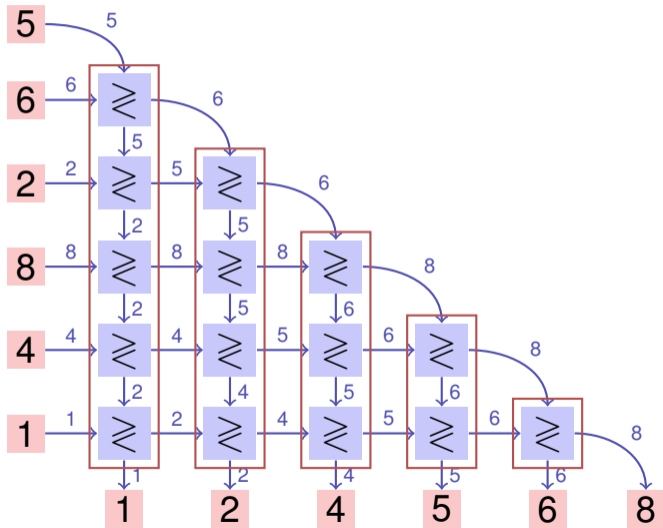
Anzahl Vertauschungen im schlechtesten Fall:  $\sum_{k=2}^n (k-1) \in \Theta(n^2)$

# Anderer Blickwinkel

Sortierknoten:

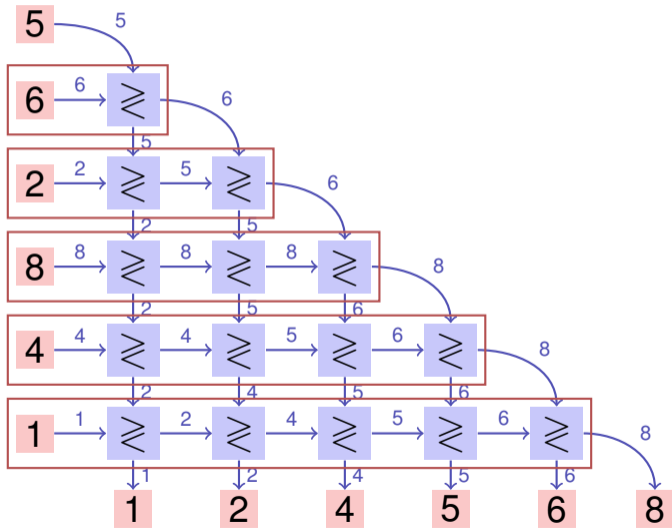


# Anderer Blickwinkel



- Wie Selection Sort [und wie Bubble Sort]

# Anderer Blickwinkel



■ Wie Insertion Sort

# Schlussfolgerung

Selection Sort, Bubble Sort und Insertion Sort sind in gewissem Sinne dieselben Sortieralgorithmen. Wird später präzisiert.<sup>7</sup>

---

<sup>7</sup>Im Teil über parallele Sortiernetzwerke. Für sequentiellen Code gelten natürlich weiterhin die zuvor gemachten Feststellungen.

# Shellsort (Donald Shell 1959)

Insertion Sort auf Teilfolgen der Form  $(A_{k \cdot i})$  ( $i \in \mathbb{N}$ ) mit absteigenden Abständen  $k$ . Letzte Länge ist zwingend  $k = 1$ .

Worst-case Performance hängt kritisch von den gewählten Teilfolgen ab.

Beispiele:

- Ursprünglich mit Folge  $1, 2, 4, 8, \dots, 2^k$  konzipiert. Laufzeit:  $\mathcal{O}(n^2)$
- Folge  $1, 3, 7, 15, \dots, 2^{k-1}$  (Hibbard 1963).  $\mathcal{O}(n^{3/2})$
- Folge  $1, 2, 3, 4, 6, 8, \dots, 2^p 3^q$  (Pratt 1971).  $\mathcal{O}(n \log^2 n)$

# Shellsort

9 8 7 6 5 4 3 2 1 0

1 8 7 6 5 4 3 2 9 0 insertion sort,  $k = 4$

1 0 7 6 5 4 3 2 9 8

1 0 3 6 5 4 7 2 9 8

1 0 3 2 5 4 7 6 9 8

1 0 3 2 5 4 7 6 9 8 insertion sort,  $k = 2$

1 0 3 2 5 4 7 6 9 8

0 1 2 3 4 5 6 7 8 9 insertion sort,  $k = 1$

# 8. Sortieren II

Heapsort, Quicksort, Mergesort



# 8.1 Heapsort

[Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

# Heapsort

Inspiration von Selectsort: Schnelles Einfügen

Inspiration von Insertionsort: Schnelles Finden der Position

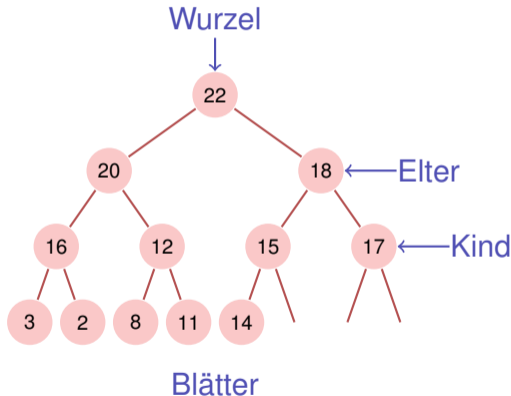
② Können wir das beste der beiden Welten haben?

⚠ Ja, aber nicht ganz so einfach...

# [Max-]Heap<sup>8</sup>

Binärer Baum mit folgenden Eigenschaften

- 1 vollständig, bis auf die letzte Ebene
- 2 Lücken des Baumes in der letzten Ebene höchstens rechts.
- 3 *Heap-Bedingung:*  
Max-(Min-)Heap: Schlüssel eines Kindes kleiner (größer) als der des Elternknotens

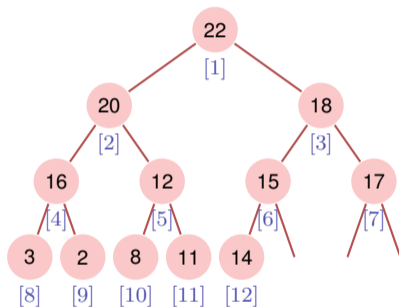
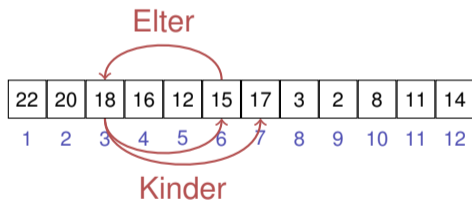


<sup>8</sup>Heap (Datenstruktur), nicht: wie in "Heap und Stack" (Speicherallokation)

# Heap als Array

Baum  $\rightarrow$  Array:

- $\text{Kinder}(i) = \{2i, 2i + 1\}$
- $\text{Elter}(i) = \lfloor i/2 \rfloor$

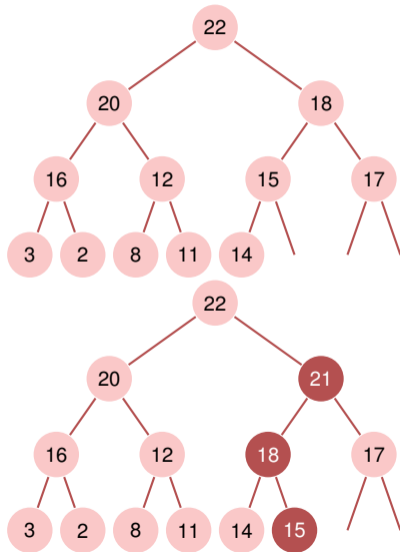


Abhängig von Startindex!<sup>9</sup>

<sup>9</sup>Für Arrays, die bei 0 beginnen:  $\{2i, 2i + 1\} \rightarrow \{2i + 1, 2i + 2\}$ ,  $\lfloor i/2 \rfloor \rightarrow \lfloor (i - 1)/2 \rfloor$

# Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.
- Anzahl Operationen im schlechtesten Fall:  $\mathcal{O}(\log n)$



# Algorithmus Aufsteigen( $A, m$ )

**Input:** Array  $A$  mit mindestens  $m + 1$  Elementen und  
Max-Heap-Struktur auf  $A[0, \dots, m - 1]$

**Output:** Array  $A$  mit Max-Heap-Struktur auf  $A[0, \dots, m]$ .

$v \leftarrow A[m]$  // Wert

$c \leftarrow m$  // derzeitiger Knoten

$p \leftarrow \lfloor (c - 1) / 2 \rfloor$  // Elternknoten

**while**  $c > 0$  and  $v > A[p]$  **do**

$A[c] \leftarrow A[p]$  // Wert Elternknoten  $\rightarrow$  derzeitiger Knoten

$c \leftarrow p$  // Elternknoten  $\rightarrow$  derzeitiger Knoten

$p \leftarrow \lfloor (c - 1) / 2 \rfloor$

$A[c] \leftarrow v$  // Wert  $\rightarrow$  derzeitiger Knoten

# Höhe eines Heaps

Vollständiger binärer Baum der Höhe<sup>10</sup>  $h$  hat

$$1 + 2 + 4 + 8 + \dots + 2^{h-1} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Knoten. Somit gilt für einen Heap der Höhe  $h$ :

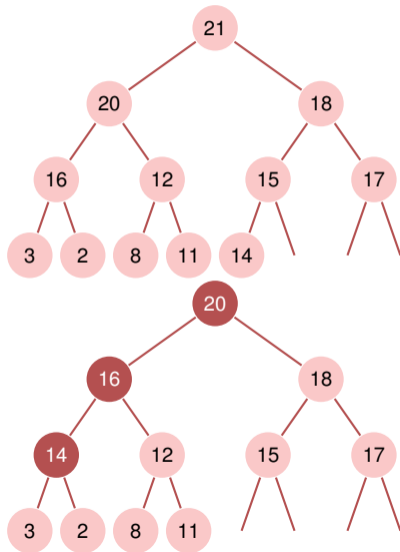
$$\begin{aligned} 2^{h-1} - 1 < n \leq 2^h - 1 \\ \Leftrightarrow 2^{h-1} < n + 1 \leq 2^h \end{aligned}$$

Also insbesondere  $h(n) = \lceil \log_2(n + 1) \rceil$  und  $h(n) \in \Theta(\log n)$ .

<sup>10</sup>Hier: Anzahl Kanten von der Wurzel zu einem Blatt

# Maximum entfernen

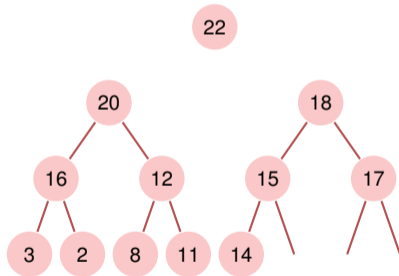
- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).
- Anzahl Operationen im schlechtesten Fall:  $\mathcal{O}(\log n)$





# Warum das korrekt ist: Rekursive Heap-Struktur

Ein Heap besteht aus zwei Teilheaps:



# Algorithmus Versickern( $A, i, m$ )

**Input:** Array  $A$  mit Heapstruktur für die Kinder von  $i$ . Letztes Element  $m$ .

**Output:** Array  $A$  mit Heapstruktur für  $i$  mit letztem Element  $m$ .

**while**  $2i \leq m$  **do**

$j \leftarrow 2i$ ; //  $j$  linkes Kind

**if**  $j < m$  and  $A[j] < A[j + 1]$  **then**

$j \leftarrow j + 1$ ; //  $j$  rechtes Kind mit grösserem Schlüssel

**if**  $A[i] < A[j]$  **then**

        swap( $A[i], A[j]$ )

$i \leftarrow j$ ; // weiter versickern

**else**

$i \leftarrow m$ ; // versickern beendet

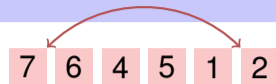
# Heap Sortieren

$A[1, \dots, n]$  ist Heap.

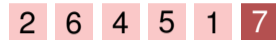
Solange  $n > 1$

- $\text{swap}(A[1], A[n])$
- $\text{Versickere}(A, 1, n - 1);$
- $n \leftarrow n - 1$

Tauschen  $\Rightarrow$



Versickern  $\Rightarrow$



Tauschen  $\Rightarrow$



Versickern  $\Rightarrow$



Tauschen  $\Rightarrow$



Versickern  $\Rightarrow$



Tauschen  $\Rightarrow$



Versickern  $\Rightarrow$



Tauschen  $\Rightarrow$



# Heap erstellen

**Beobachtung:** Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap.

**Folgerung:** Induktion von unten!

# Algorithmus HeapSort( $A, n$ )

**Input:** Array  $A$  der Länge  $n$ .

**Output:**  $A$  sortiert.

// Heap Bauen.

**for**  $i \leftarrow n/2$  **downto** 1 **do**

└ Versickere( $A, i, n$ );

// Nun ist  $A$  ein Heap.

**for**  $i \leftarrow n$  **downto** 2 **do**

└ swap( $A[1], A[i]$ )

└ Versickere( $A, 1, i - 1$ )

// Nun ist  $A$  sortiert.

# Analyse: Sortieren eines Heaps

Versickere durchläuft maximal  $\log n$  Knoten. An jedem Knoten 2 Schlüsselvergleiche.  $\Rightarrow$  Heap Sortieren kostet im schlechtesten Fall  $2n \log n$  Vergleiche.

Anzahl der Bewegungen vom Heap Sortieren auch  $\mathcal{O}(n \log n)$ .

# Analyse: Heap bauen

Aufrufe an Versickern:  $n/2$ . Also Anzahl Vergleiche und Bewegungen  $v(n) \in \mathcal{O}(n \log n)$ .

Versickerpfade sind aber im Mittel viel kürzer:

$$\begin{aligned} v(n) &= \sum_{l=0}^{\lfloor \log n \rfloor} \underbrace{2^l}_{\text{Anzahl Heaps auf Level } l} \cdot \underbrace{(\lfloor \log n \rfloor - l)}_{\text{Höhe Heaps auf Level } l} = \sum_{k=0}^{\lfloor \log n \rfloor} 2^{\lfloor \log n \rfloor - k} \cdot k \\ &\leq \sum_{k=0}^{\lfloor \log n \rfloor} \frac{n}{2^k} \cdot k = n \cdot \sum_{k=0}^{\lfloor \log n \rfloor} \frac{k}{2^k} \in \mathcal{O}(n) \end{aligned}$$

mit  $s(x) := \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$  ( $0 < x < 1$ )<sup>11</sup> und  $s(\frac{1}{2}) = 2$

---

<sup>11</sup>  $f(x) = \frac{1}{1-x} = 1 + x + x^2 \dots \Rightarrow f'(x) = \frac{1}{(1-x)^2} = 1 + 2x + \dots$

# Zwischenstand

Heapsort:  $\mathcal{O}(n \log n)$  Vergleiche und Bewegungen.

## ❓ Nachteile von Heapsort?

- ❗ Wenig Lokalität: per Definition springt Heapsort im sortierten Array umher (Negativer Cache Effekt).
- ❗ Zwei Vergleiche vor jeder benötigten Bewegung.



## 8.2 Mergesort

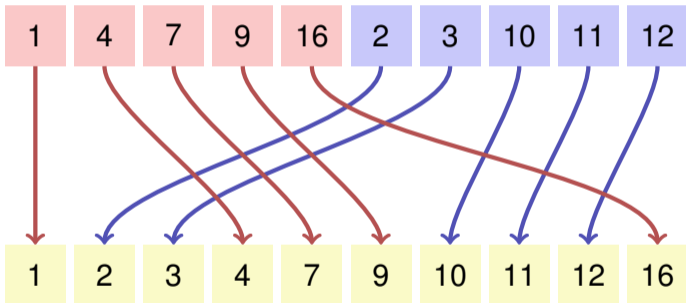
[Ottman/Widmayer, Kap. 2.4, Cormen et al, Kap. 2.3],

# Mergesort (Sortieren durch Verschmelzen)

Divide and Conquer!

- Annahme: Zwei Hälften eines Arrays  $A$  bereits sortiert.
- Folgerung: Minimum von  $A$  kann mit 2 Vergleichen ermittelt werden.
- Iterativ: Füge die beiden vorsortierten Hälften von  $A$  zusammen in  $\mathcal{O}(n)$ .

# Merge



# Algorithmus Merge( $A, l, m, r$ )

**Input:** Array  $A$  der Länge  $n$ , Indizes  $1 \leq l \leq m \leq r \leq n$ .

$A[l, \dots, m]$ ,  $A[m + 1, \dots, r]$  sortiert

**Output:**  $A[l, \dots, r]$  sortiert

1  $B \leftarrow$  new Array( $r - l + 1$ )

2  $i \leftarrow l$ ;  $j \leftarrow m + 1$ ;  $k \leftarrow 1$

3 **while**  $i \leq m$  and  $j \leq r$  **do**

4     **if**  $A[i] \leq A[j]$  **then**  $B[k] \leftarrow A[i]$ ;  $i \leftarrow i + 1$

5     **else**  $B[k] \leftarrow A[j]$ ;  $j \leftarrow j + 1$

6      $k \leftarrow k + 1$ ;

7 **while**  $i \leq m$  **do**  $B[k] \leftarrow A[i]$ ;  $i \leftarrow i + 1$ ;  $k \leftarrow k + 1$

8 **while**  $j \leq r$  **do**  $B[k] \leftarrow A[j]$ ;  $j \leftarrow j + 1$ ;  $k \leftarrow k + 1$

9 **for**  $k \leftarrow l$  **to**  $r$  **do**  $A[k] \leftarrow B[k - l + 1]$

# Korrektheit

Hypothese: Nach  $k$  Durchläufen der Schleife von Zeile 3 ist  $B[1, \dots, k]$  sortiert und  $B[k] \leq A[i]$ , falls  $i \leq m$  und  $B[k] \leq A[j]$  falls  $j \leq r$ .

Beweis per Induktion:

*Induktionsanfang:* Das leere Array  $B[1, \dots, 0]$  ist trivialerweise sortiert.

*Induktionsschluss* ( $k \rightarrow k + 1$ ):

- oBdA  $A[i] \leq A[j]$ ,  $i \leq m$ ,  $j \leq r$ .
- $B[1, \dots, k]$  ist nach Hypothese sortiert und  $B[k] \leq A[i]$ .
- Nach  $B[k + 1] \leftarrow A[i]$  ist  $B[1, \dots, k + 1]$  sortiert.
- $B[k + 1] = A[i] \leq A[i + 1]$  (falls  $i + 1 \leq m$ ) und  $B[k + 1] \leq A[j]$  falls  $j \leq r$ .
- $k \leftarrow k + 1$ ,  $i \leftarrow i + 1$ : Aussage gilt erneut.

# Analyse (Merge)

## Lemma

*Wenn: Array  $A$  der Länge  $n$ , Indizes  $1 \leq l < r \leq n$ .  $m = \lfloor (l + r)/2 \rfloor$  und  $A[l, \dots, m]$ ,  $A[m + 1, \dots, r]$  sortiert.*

*Dann: im Aufruf  $\text{Merge}(A, l, m, r)$  werden  $\Theta(r - l)$  viele Schlüsselbewegungen und Vergleiche durchgeführt.*

Beweis: (Inspektion des Algorithmus und Zählen der Operationen).

# Mergesort

5 2 6 1 8 4 3 9

5 2 6 1 8 4 3 9

5 2 6 1 8 4 3 9

5 2 6 1 8 4 3 9

2 5 1 6 4 8 3 9

1 2 5 6 3 4 8 9

1 2 3 4 5 6 8 9

Split

Split

Split

Merge

Merge

Merge

# Algorithmus (Rekursives 2-Wege) Mergesort( $A, l, r$ )

**Input:** Array  $A$  der Länge  $n$ .  $1 \leq l \leq r \leq n$

**Output:** Array  $A[l, \dots, r]$  sortiert.

**if**  $l < r$  **then**

```
 $m \leftarrow \lfloor (l + r) / 2 \rfloor$  // Mittlere Position  
Mergesort( $A, l, m$ ) // Sortiere vordere Hälfte  
Mergesort( $A, m + 1, r$ ) // Sortiere hintere Hälfte  
Merge( $A, l, m, r$ ) // Verschmelzen der Teilfolgen
```



# Analyse

Rekursionsgleichung für die Anzahl Vergleiche und Schlüsselbewegungen:

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n) \in \Theta(n \log n)$$

# Algorithmus StraightMergesort( $A$ )

*Rekursion vermeiden*: Verschmelze Folgen der Länge 1, 2, 4... direkt

**Input:** Array  $A$  der Länge  $n$

**Output:** Array  $A$  sortiert

$length \leftarrow 1$

**while**  $length < n$  **do** // Iteriere über die Längen  $n$

$r \leftarrow 0$

**while**  $r + length < n$  **do** // Iteriere über die Teilfolgen

$l \leftarrow r + 1$

$m \leftarrow l + length - 1$

$r \leftarrow \min(m + length, n)$

        Merge( $A, l, m, r$ )

$length \leftarrow length \cdot 2$

Wie rekursives Mergesort führt reines 2-Wege-Mergesort immer  $\Theta(n \log n)$  viele Schlüsselvergleiche und -bewegungen aus.

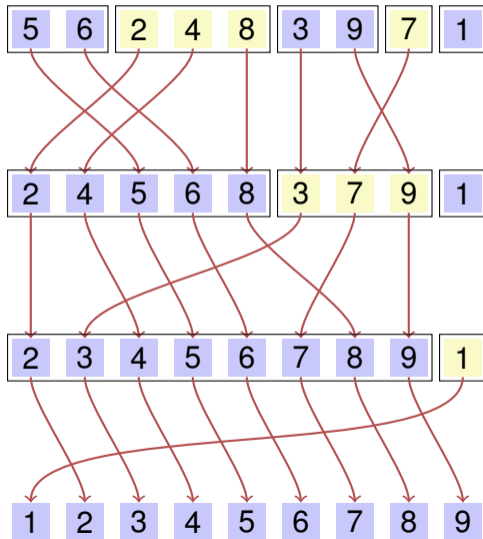
# Natürliches 2-Wege Mergesort

Beobachtung: Obige Varianten nutzen nicht aus, wenn vorsortiert ist und führen immer  $\Theta(n \log n)$  viele Bewegungen aus.

❓ Wie kann man teilweise vorsortierte Folgen besser sortieren?

❗ Rekursives Verschmelzen von bereits vorsortierten Teilen (*Runs*) von  $A$ .

# Natürliches 2-Wege Mergesort



# Algorithmus NaturalMergesort( $A$ )

**Input:** Array  $A$  der Länge  $n > 0$

**Output:** Array  $A$  sortiert

**repeat**

$r \leftarrow 0$

**while**  $r < n$  **do**

$l \leftarrow r + 1$

$m \leftarrow l$ ; **while**  $m < n$  **and**  $A[m + 1] \geq A[m]$  **do**  $m \leftarrow m + 1$

**if**  $m < n$  **then**

$r \leftarrow m + 1$ ; **while**  $r < n$  **and**  $A[r + 1] \geq A[r]$  **do**  $r \leftarrow r + 1$

            Merge( $A, l, m, r$ );

**else**

$r \leftarrow n$

**until**  $l = 1$

# Analyse

❓ Ist es auch im Mittel asymptotisch besser als StraightMergesort?

❗ Nein. Unter Annahme der Gleichverteilung der paarweise unterschiedlichen Schlüssel haben wir im Mittel  $n/2$  Stellen  $i$  mit  $k_i > k_{i+1}$ , also  $n/2$  Runs und sparen uns lediglich einen Durchlauf, also  $n$  Vergleiche.

Natürliches Mergesort führt im schlechtesten und durchschnittlichen Fall  $\Theta(n \log n)$  viele Vergleiche und Bewegungen aus.

## 8.3 Quicksort

[Ottman/Widmayer, Kap. 2.2, Cormen et al, Kap. 7]



# Quicksort

❓ Was ist der Nachteil von Mergesort?

❗ Benötigt zusätzlich  $\Theta(n)$  Speicherplatz für das Verschmelzen.

❓ Wie könnte man das Verschmelzen einsparen?

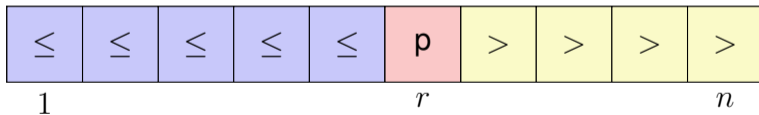
❗ Sorge dafür, dass jedes Element im linken Teil kleiner ist als im rechten Teil.

❓ Wie?

❗ Pivotieren und Aufteilen!

# Pivotieren

- 1 Wähle ein (beliebiges) Element  $p$  als Pivotelement
- 2 Teile  $A$  in zwei Teile auf: einen Teil  $L$  der Elemente mit  $A[i] \leq p$  und einen Teil  $R$  der Elemente mit  $A[i] > p$ .
- 3 Quicksort: Rekursion auf Teilen  $L$  und  $R$



# Algorithmus Partition( $A[l..r], p$ )

**Input:** Array  $A$ , welches den Pivot  $p$  im Intervall  $[l, r]$  mindestens einmal enthält.

**Output:** Array  $A$  partitioniert in  $[l..r]$  um  $p$ . Rückgabe der Position von  $p$ .

**while**  $l \leq r$  **do**

**while**  $A[l] < p$  **do**

$l \leftarrow l + 1$

**while**  $A[r] > p$  **do**

$r \leftarrow r - 1$

    swap( $A[l], A[r]$ )

**if**  $A[l] = A[r]$  **then**

$l \leftarrow l + 1$

**return**  $l-1$

# Algorithmus Quicksort( $A[l, \dots, r]$ )

**Input:** Array  $A$  der Länge  $n$ .  $1 \leq l \leq r \leq n$ .

**Output:** Array  $A$ , sortiert zwischen  $l$  und  $r$ .

**if**  $l < r$  **then**

    Wähle Pivot  $p \in A[l, \dots, r]$

$k \leftarrow \text{Partition}(A[l, \dots, r], p)$

    Quicksort( $A[l, \dots, k - 1]$ )

    Quicksort( $A[k + 1, \dots, r]$ )

# Quicksort (willkürlicher Pivot)

2 4 5 6 8 3 7 9 1

2 1 3 6 8 5 7 9 4

1 2 3 4 5 8 7 9 6

1 2 3 4 5 6 7 9 8

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

# Analyse: Anzahl Vergleiche

*Schlechtester Fall.* Pivotelement = Minimum oder Maximum; Anzahl Vergleiche:

$$T(n) = T(n - 1) + c \cdot n, T(1) = 0 \quad \Rightarrow \quad T(n) \in \Theta(n^2)$$

# Analyse: Anzahl Vertauschungen

Resultat eines Aufrufes an Partition (Pivot 3):

2 1 3 6 8 5 7 9 4

- ② Wie viele Vertauschungen haben hier maximal stattgefunden?
- ① 2. Die maximale Anzahl an Vertauschungen ist gegeben durch die Anzahl Schlüssel im kleineren Bereich.

# Analyse: Anzahl Vertauschungen

## *Gedankenspiel*

- Jeder Schlüssel aus dem kleineren Bereich zahlt bei einer Vertauschung eine Münze.
- Wenn ein Schlüssel eine Münze gezahlt hat, ist der Bereich, in dem er sich befindet maximal halb so gross wie zuvor.
- Jeder Schlüssel muss also maximal  $\log n$  Münzen zahlen. Es gibt aber nur  $n$  Schlüssel.

*Folgerung:* Es ergeben sich  $\mathcal{O}(n \log n)$  viele Schlüsselvertauschungen im schlechtesten Fall!



# Randomisiertes Quicksort

Quicksort wird trotz  $\Theta(n^2)$  Laufzeit im schlechtesten Fall oft eingesetzt.

Grund: Quadratische Laufzeit unwahrscheinlich, sofern die Wahl des Pivots und die Vorsortierung nicht eine ungünstige Konstellation aufweisen.

Vermeidung: Zufälliges Ziehen eines Pivots. Mit gleicher Wahrscheinlichkeit aus  $[l, r]$ .

# Analyse (Randomisiertes Quicksort)

Erwartete Anzahl verglichener Schlüssel bei Eingabe der Länge  $n$ :

$$T(n) = (n - 1) + \frac{1}{n} \sum_{k=1}^n (T(k - 1) + T(n - k)), \quad T(0) = T(1) = 0$$

Behauptung  $T(n) \leq 4n \log n$ .

Beweis per Induktion:

*Induktionsanfang:* klar für  $n = 0$  (mit  $0 \log 0 := 0$ ) und für  $n = 1$ .

*Hypothese:*  $T(n) \leq 4n \log n$  für ein  $n$ .

*Induktionsschritt:*  $(n - 1 \rightarrow n)$

# Analyse (Randomisiertes Quicksort)

$$\begin{aligned}T(n) &= n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} T(k) \stackrel{H}{\leq} n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} 4k \log k \\&= n - 1 + \sum_{k=1}^{n/2} 4k \underbrace{\log k}_{\leq \log n - 1} + \sum_{k=n/2+1}^{n-1} 4k \underbrace{\log k}_{\leq \log n} \\&\leq n - 1 + \frac{8}{n} \left( (\log n - 1) \sum_{k=1}^{n/2} k + \log n \sum_{k=n/2+1}^{n-1} k \right) \\&= n - 1 + \frac{8}{n} \left( (\log n) \cdot \frac{n(n-1)}{2} - \frac{n}{4} \left( \frac{n}{2} + 1 \right) \right) \\&= 4n \log n - 4 \log n - 3 \leq 4n \log n\end{aligned}$$

# Analyse (Randomisiertes Quicksort)

## Theorem

*Im Mittel benötigt randomisiertes Quicksort  $\mathcal{O}(n \cdot \log n)$  Vergleiche.*

# Praktische Anmerkungen

Rekursionstiefe im schlechtesten Fall:  $n - 1$ <sup>12</sup>. Dann auch Speicherplatzbedarf  $\mathcal{O}(n)$ .

Kann vermieden werden: Rekursion nur auf dem kleineren Teil. Dann garantiert  $\mathcal{O}(\log n)$  Rekursionstiefe und Speicherplatzbedarf.

---

<sup>12</sup>Stack-Overflow möglich!

# Quicksort mit logarithmischem Speicherplatz

**Input:** Array  $A$  der Länge  $n$ .  $1 \leq l \leq r \leq n$ .

**Output:** Array  $A$ , sortiert zwischen  $l$  und  $r$ .

**while**  $l < r$  **do**

    Wähle Pivot  $p \in A[l, \dots, r]$

$k \leftarrow \text{Partition}(A[l, \dots, r], p)$

**if**  $k - l < r - k$  **then**

        Quicksort( $A[l, \dots, k - 1]$ )

$l \leftarrow k + 1$

**else**

        Quicksort( $A[k + 1, \dots, r]$ )

$r \leftarrow k - 1$

Der im ursprünglichen Algorithmus verbleibende Aufruf an Quicksort( $A[l, \dots, r]$ ) geschieht iterativ (Tail Recursion ausgenutzt!): die If-Anweisung wurde zur While Anweisung.

# Praktische Anmerkungen

- Für den Pivot wird in der Praxis oft der Median von drei Elementen genommen. Beispiel:  $\text{Median3}(A[l], A[r], A[\lfloor l + r/2 \rfloor])$ .
- Es existiert eine Variante von Quicksort mit konstanten Speicherplatzbedarf. Idee: Zwischenspeichern des alten Pivots am Ort des neuen Pivots.
- Komplizierte Divide-And-Conquer-Algorithmen verwenden oft als Basisfall einen trivialen ( $\Theta(n^2)$ ) Algorithmus für kleine Problemgrößen.

## 8.4 Anhang

Herleitung einiger mathematischen Formeln



$$\log n! \in \Theta(n \log n)$$

$$\begin{aligned}\log n! &= \sum_{i=1}^n \log i \leq \sum_{i=1}^n \log n = n \log n \\ \sum_{i=1}^n \log i &= \sum_{i=1}^{\lfloor n/2 \rfloor} \log i + \sum_{\lfloor n/2 \rfloor + 1}^n \log i \\ &\geq \sum_{i=2}^{\lfloor n/2 \rfloor} \log 2 + \sum_{\lfloor n/2 \rfloor + 1}^n \log \frac{n}{2} \\ &= \underbrace{(\lfloor n/2 \rfloor - 2 + 1)}_{> n/2 - 1} + \underbrace{(n - \lfloor n/2 \rfloor)}_{\geq n/2} (\log n - 1) \\ &> \frac{n}{2} \log n - 2.\end{aligned}$$

$$[n! \in o(n^n)]$$

$$\begin{aligned} n \log n &\geq \sum_{i=1}^{\lfloor n/2 \rfloor} \log 2i + \sum_{i=\lfloor n/2 \rfloor+1}^n \log i \\ &= \sum_{i=1}^n \log i + \left\lfloor \frac{n}{2} \right\rfloor \log 2 \\ &> \sum_{i=1}^n \log i + n/2 - 1 = \log n! + n/2 - 1 \end{aligned}$$

$$\begin{aligned} n^n &= 2^{n \log_2 n} \geq 2^{\log_2 n!} \cdot 2^{n/2} \cdot 2^{-1} = n! \cdot 2^{n/2-1} \\ \Rightarrow \frac{n!}{n^n} &\leq 2^{-n/2+1} \xrightarrow{n \rightarrow \infty} 0 \Rightarrow n! \in o(n^n) = \mathcal{O}(n^n) \setminus \Omega(n^n) \end{aligned}$$

**[Sogar  $n! \in o((n/c)^n) \forall 0 < c < e$  ]**

Konvergenz oder Divergenz von  $f_n = \frac{n!}{(n/c)^n}$ .

Quotientenkriterium

$$\frac{f_{n+1}}{f_n} = \frac{(n+1)!}{\left(\frac{n+1}{c}\right)^{n+1}} \cdot \frac{\left(\frac{n}{c}\right)^n}{n!} = c \cdot \left(\frac{n}{n+1}\right)^n \longrightarrow c \cdot \frac{1}{e} \lesseqgtr 1 \text{ wenn } c \lesseqgtr e$$

denn  $\left(1 + \frac{1}{n}\right)^n \rightarrow e$ . Sogar die Reihe  $\sum_{i=1}^n f_n$  konvergiert / divergiert für  $c \lesseqgtr e$ .

$f_n$  divergiert für  $c = e$ , denn (Stirling):  $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ .

# [ Quotientenkriterium]

Quotientenkriterium für eine Folge  $(f_n)_{n \in \mathbb{N}}$ : Wenn  $\frac{f_{n+1}}{f_n} \xrightarrow{n \rightarrow \infty} \lambda$ , dann sind die Folge  $f_n$  und auch die Reihe  $\sum_{i=1}^n f_i$

- konvergent, falls  $\lambda < 1$  und
- divergent, falls  $\lambda > 1$ .

# [ Quotientenkriterium Herleitung ]

Quotientenkriterium ergibt sich aus: Geometrische Reihe

$$S_n(r) := \sum_{i=0}^n r^i = \frac{1 - r^{n+1}}{1 - r}.$$

konvergiert für  $n \rightarrow \infty$  genau dann wenn  $-1 < r < 1$ .

Sei nämlich  $0 \leq \lambda < 1$ :

$$\begin{aligned} & \forall \varepsilon > 0 \exists n_0 : f_{n+1}/f_n < \lambda + \varepsilon \forall n \geq n_0 \\ \Rightarrow & \exists \varepsilon > 0, \exists n_0 : f_{n+1}/f_n \leq \mu < 1 \forall n \geq n_0 \end{aligned}$$

Somit

$$\sum_{n=n_0}^{\infty} f_n \leq f_{n_0} \cdot \sum_{n=n_0}^{\infty} \mu^{n-n_0} \quad \text{konvergiert.}$$

(Analog für Divergenz)

## **9. C++ vertieft (II): Templates**

# Was lernen wir heute?

- Templates von Klassen
- Funktionentemplates
- Spezialisierung
- Templates mit Werten

# Motivation

Ziel: generische Vektor-Klasse und Funktionalität.

## Beispiele

```
Vector<double> vd(10);
```

```
Vector<int> vi(10);
```

```
Vector<char> vi(20);
```

```
auto nd = vd * vd; // norm (vector of double)
```

```
auto ni = vi * vi; // norm (vector of int)
```



# Typen als Template Parameter

- 1 Ersetze in der konkreten Implementation einer Klasse den Typ, der generisch werden soll (beim Vektor: `double`) durch einen Stellvertreter, z.B. `T`.
- 2 Stelle der Klasse das Konstrukt `template<typename T>`<sup>13</sup> voran (ersetze `T` ggfs. durch den Stellvertreter)..

Das Konstrukt `template<typename T>` kann gelesen werden als “für alle Typen `T`”.

---

<sup>13</sup>gleichbedeutend: `template<class T>`

# Typen als Template Parameter

```
template <typename ElementType>
class Vector{
    std::size_t size;
    ElementType* elem;
public:
    ...
    Vector(std::size_t s):
        size{s},
        elem{new ElementType[s]}{}
    ...
    ElementType& operator [] (std::size_t pos){
        return elem[pos];
    }
    ...
}
```

# Template Instanziierung

`Vector<typeName>` erzeugt Typinstanz von `Vector` mit `ElementType=typeName`.

Bezeichnung: **Instanziierung**.

## Beispiele

```
Vector<double> x;           // vector of double
Vector<int> y;             // vector of int
Vector<Vector<double>> x;  // vector of vector of double
```

# Type-checking

Templates sind weitgehend Ersetzungsregeln zur Instanzierungszeit und während der Kompilation. Es wird immer so wenig geprüft wie nötig und so viel wie möglich.

# Beispiel

```
template <typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r):left{l}, right{r}{}
    T min(){
        return left < right ? left : right;
    }
};
```

```
Pair<int> a(10,20); // ok
auto m = a.min(); // ok
Pair<Pair<int>> b(a,Pair<int>(20,30)); // ok
auto n = b.min(); no match for operator< !
```

# Generische Programmierung

Generische Komponenten sollten eher als **Generalisierung eines oder mehrerer Beispiele** entwickelt werden als durch Ableitung von Grundprinzipien.

```
template <typename T>
class Vector{
public:
    Vector ();
    Vector(std :: size_t);
    ~Vector();
    Vector(const Vector&);
    Vector& operator=(const Vector&);
    Vector (Vector&&);
    Vector& operator=(Vector&&);
    const T& operator[] (std :: size_t) const;
    T& operator[] (std :: size_t);
    std :: size_t size() const;
    T* begin();
    T* end();
    const T* begin() const;
    const T* end() const;
}
```

# Funktionentemplates

- 1 Ersetze in der konkreten Implementation einer Funktion den Typ, der generisch werden soll durch einen Namen, z.B. **T**,
- 2 Stelle der Funktion das Konstrukt `template<typename T>`<sup>14</sup> voran (ersetze **T** ggfs. durch den gewählten Namen).

---

<sup>14</sup>gleichbedeutend: `template<class T>`

# Funktionentemplates

```
template <typename T>
void swap(T& x, T&y){
    T temp = x;
    x = y;
    y = temp;
}
```

Typen der Aufrufparameter determinieren die Version der Funktion, welche (kompiliert und) verwendet wird:

```
int x=5;
int y=6;
swap(x,y); // calls swap with T=int
```



# Grenzen der Magie

```
template <typename T>
void swap(T& x, T&y){
    T temp = x;
    x = y;
    y = temp;
}
```

Eine unverträgliche Version der Funktion wird nicht erzeugt:

```
int x=5;
double y=6;
swap(x,y); // error: no matching function for ...
```

## .. auch mit Operatoren

```
template <typename T>
```

```
class Pair{  
    T left; T right;
```

```
public:
```

```
    Pair(T l, T r):left{l}, right{r}{}  
    T min(){ return left < right? left: right; }  
    std::ostream& print (std::ostream& os) const{  
        return os << "(" << left << ", " << right << ")";  
    }  
};
```

```
Pair<int> a(10,20); // ok  
std::cout << a; // ok
```

```
template <typename T>
```

```
std::ostream& operator<< (std::ostream& os, const Pair<T>& pair){  
    return pair.print(os);  
}
```

# Praktisch!

```
// Output of an arbitrary container
template <typename T>
void output(const T& t){
    for (auto x: t)
        std::cout << x << " ";
    std::cout << "\n";
}

int main(){
    std::vector<int> v={1,2,3};
    output(v); // 1 2 3
}
```

# Explizite Typangabe

```
// input of an arbitrary pair
template <typename T>
Pair<T> read(){
    T left;
    T right;
    std::cin << left << right;
    return Pair<T>(left,right);
}
...
```

```
auto p = read<double>();
```

Wenn der Typ bei der Instanzierung nicht inferiert werden kann, muss er explizit angegeben werden.

# Mächtig!

```
template <typename T> // square number
T sq(T x){
    return x*x;
}
template <typename Container, typename F>
void apply(Container& c, F f){ // x ← f(x) forall x in c
    for(auto& x: c)
        x = f(x);
}
int main(){
    std::vector<int> v={1,2,3};
    apply(v,sq<int>);
    output(v); // 1 4 9
}
```

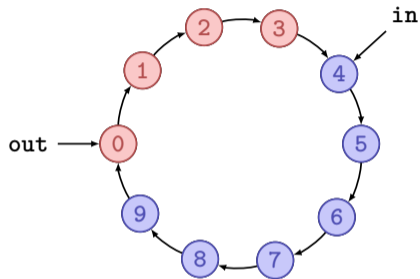
# Spezialisierung

```
template <>
class Pair<bool>{
    short both;
public:
    Pair(bool l, bool r):both{(l?1:0) + (r?2:0)} {};
    std::ostream& print (std::ostream& os) const{
        return os << "(" << both % 2 << ", " << both /2 << ")";
    }
};

Pair<int> i(10,20); // ok -- generic template
std::cout << i << std::endl; // (10,20);
Pair<bool> b(true, false); // ok -- special bool version
std::cout << b << std::endl; // (1,0)
```

# Templateparametrisierung mit Werten

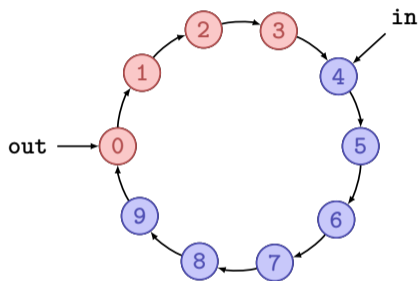
```
template <typename T, int size>
class CircularBuffer{
    T buf[size] ;
    int in; int out;
public:
    CircularBuffer():in{0},out{0}{};
    bool empty(){
        return in == out;
    }
    bool full(){
        return (in + 1) % size == out;
    }
    void put(T x); // declaration
    T get();      // declaration
};
```



# Templateparametrisierung mit Werten

```
template <typename T, int size>
void CircularBuffer<T,size>::put(T x){
    assert(!full());
    buf[in] = x;
    in = (in + 1) % size;
}
```

```
template <typename T, int size>
T CircularBuffer<T,size>::get(){
    assert(!empty());
    T x = buf[out];
    out = (out + 1) % size; ← Optimierungspotential, wenn  $size = 2^k$ .
    return x;
}
```





# Speichermanagement

## Richtlinie “Dynamischer Speicher”

Zu jedem `new` gibt es ein passendes `delete`!

Vermeide:

- *Speicherlecks*: “alte” Objekte, die den Speicher blockieren
- Zeiger auf freigegebene Objekte: hängende Zeiger (*dangling pointers*)
- Mehrfache Freigabe eines Objektes mit `delete`.

Wie?

# Smart Pointers

- Können sicherstellen, dass ein Objekt gelöscht wird genau dann, wenn es nicht mehr genutzt wird
- Basieren auf dem RAII (Resource Acquisition is Initialization) Paradigma.
- Können an die Stelle jedes gewöhnlichen Pointers treten: sind als Klassentemplates implementiert.
- Es gibt `std::unique_ptr<>`, `std::shared_ptr<>` (und `std::weak_ptr<>`)

```
std::unique_ptr<Node> nodeU(new Node()); // unique pointer
std::shared_ptr<Node> nodeS(new Node()); // shared pointer
```

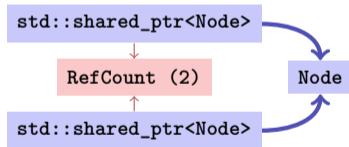
# Unique Pointer

- Der Dekonstruktor von `std::unique_ptr<T>` löscht den enthaltenen Zeiger.
- `std::unique_ptr<T>` hat exklusiv Zugriff auf den enthaltenen Zeiger auf T.
- Kopierkonstruktor und Assignment Operator sind gelöscht. Ein Unique Pointer kann nicht als Wert kopiert werden. Movekonstruktor ist vorhanden: der Zeiger kann verschoben werden.
- Kein Zusatzaufwand zur Laufzeit im Vergleich zu einem normalen Zeiger.

```
std::unique_ptr<Node> nodeU(new Node()); // unique pointer
std::unique_ptr<Node> node2 = std::move(nodeU); // ok
std::unique_ptr<Node> node3 = nodeU; // error
```

# Shared Pointer

- `std::shared_ptr<T>` zählt die Anzahl von Besitzern eines Zeigers (Referenzzähler). Wenn der Referenzzähler auf 0 fällt, wird der Zeiger gelöscht.
- Shared Pointers können kopiert werden.
- Shared Pointers haben zusätzlichen Speicher- und Laufzeitbedarf: sie verwalten den Referenzzähler zur Laufzeit und enthalten jeweils einen Zeiger auf den Referenzzähler.



```
std::shared_ptr<Node> nodeS(new Node()); // shared pointer, rc = 1
std::shared_ptr<Node> node2 = std::move(nodeS); // ok, rc unchanged
std::shared_ptr<Node> node3 = node2; // ok, rc = 2
```

# Smart Pointers

## Einige Regeln

- Niemals `delete` auf einen Zeiger im Smart Pointer aufrufen.
- `new` vermeiden, stattdessen:

```
std::unique_ptr<Node> nodeU = std::make_unique<Node>()  
std::shared_ptr<Node> nodeS = std::make_shared<Node>()
```

- Wo möglich, `std::unique_ptr` verwenden.
- Bei der Verwendung von `std::shared_ptr` sicherstellen, dass es keine Zyklen im Zeigergraphen gibt.

# 10. Sortieren III

Untere Schranken für das vergleichsbasierte Sortieren, Radix- und Bucketsort

# 10.1 Untere Grenzen für Vergleichbasiertes Sortieren

[Ottman/Widmayer, Kap. 2.8, Cormen et al, Kap. 8.1]

# Untere Schranke für das Sortieren

Bis hierher: Sortieren im schlechtesten Fall benötigt  $\Omega(n \log n)$  Schritte.

Geht es besser? Nein:

## Theorem

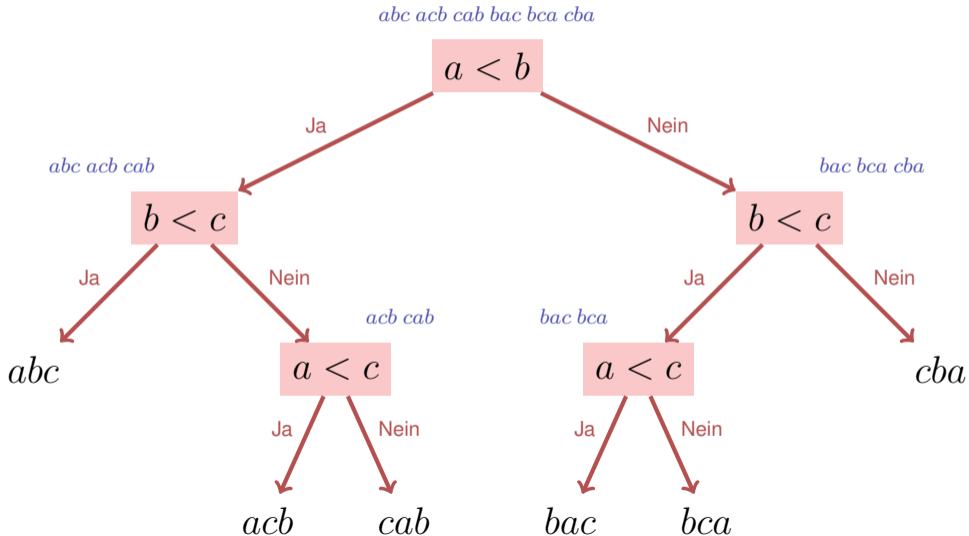
*Vergleichsbasierte Sortierverfahren benötigen im schlechtesten Fall und im Mittel mindestens  $\Omega(n \log n)$  Schlüsselvergleiche.*



# Vergleichsbasiertes Sortieren

- Algorithmus muss unter  $n!$  vielen Anordnungsmöglichkeiten einer Folge  $(A_i)_{i=1,\dots,n}$  die richtige identifizieren.
- Zu Beginn weiss der Algorithmus nichts.
- Betrachten den “Wissensgewinn” des Algorithmus als Entscheidungsbaum:
  - Knoten enthalten verbleibende Möglichkeiten
  - Kanten enthalten Entscheidungen

# Entscheidungsbaum



# Entscheidungsbaum

Ein binärer Baum mit  $L$  Blättern hat  $K = L - 1$  innere Knoten.<sup>15</sup>

Die Höhe eines binären Baumes mit  $L$  Blättern ist mindestens  $\log_2 L$ .  $\Rightarrow$  Höhe des Entscheidungsbaumes  $h \geq \log n! \in \Omega(n \log n)$ .

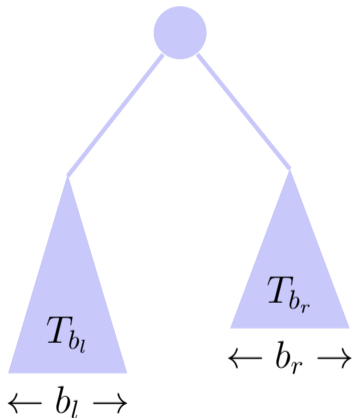
Somit auch die Länge des längsten Pfades im Entscheidungsbaum  $\in \Omega(n \log n)$ .

Bleibt zu zeigen: mittlere Länge  $M(n)$  eines Pfades  
 $M(n) \in \Omega(n \log n)$ .

---

<sup>15</sup>Beweis: starte mit leerem Baum,  $K = 0$ ,  $L = 1$ . Jeder hinzugefügte Knoten ersetzt ein Blatt durch 2 Blätter. Also.

# Untere Schranke im Mittel



- Entscheidungsbaum  $T_n$  mit  $n$  Blättern, mittlere Tiefe eines Blatts  $m(T_n)$
- Annahme:  $m(T_n) \geq \log n$  nicht für alle  $n$ .
- Wähle kleinstes  $b$  mit  $m(T_b) < \log b \Rightarrow b \geq 2$
- $b_l + b_r = b$  with  $b_l > 0$  und  $b_r > 0 \Rightarrow b_l < b, b_r < b \Rightarrow m(T_{b_l}) \geq \log b_l$  und  $m(T_{b_r}) \geq \log b_r$

# Untere Schranke im Mittel

Mittlere Tiefe eines Blatts:

$$\begin{aligned}m(T_b) &= \frac{b_l}{b}(m(T_{b_l}) + 1) + \frac{b_r}{b}(m(T_{b_r}) + 1) \\ &\geq \frac{1}{b}(b_l(\log b_l + 1) + b_r(\log b_r + 1)) = \frac{1}{b}(b_l \log 2b_l + b_r \log 2b_r) \\ &\geq \frac{1}{b}(b \log b) = \log b.\end{aligned}$$

Widerspruch. ■

Die letzte Ungleichung gilt, da  $f(x) = x \log x$  konvex ist ( $f''(x) = 1/x > 0$ ) und für eine konvexe Funktion gilt  $f((x+y)/2) \leq 1/2f(x) + 1/2f(y)$  ( $x = 2b_l, y = 2b_r$  einsetzen).<sup>16</sup> Einsetzen von  $x = 2b_l, y = 2b_r$ , und  $b_l + b_r = b$ .

---

<sup>16</sup>allgemein  $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$  für  $0 \leq \lambda \leq 1$ .

## 10.2 Radixsort und Bucketsort

Radixsort, Bucketsort [Ottman/Widmayer, Kap. 2.5, Cormen et al, Kap. 8.3]

# Radix Sort

*Vergleichsbasierte Sortierverfahren:* Schlüssel vergleichbar ( $<$  oder  $>$ ,  $=$ ). Ansonsten keine Voraussetzung.

*Andere Idee:* nutze mehr Information über die Zusammensetzung der Schlüssel.

# Annahmen

Annahme: Schlüssel darstellbar als Wörter aus einem Alphabet mit  $m$  Elementen.

## Beispiele

$m = 10$	Dezimalzahlen	$183 = 183_{10}$
$m = 2$	Dualzahlen	$101_2$
$m = 16$	Hexadezimalzahlen	$A0_{16}$
$m = 26$	Wörter	“INFORMATIK”

$m$  heisst die Wurzel (lateinisch *Radix*) der Darstellung.



# Annahmen

- Schlüssel =  $m$ -adische Zahlen mit gleicher Länge.
- Verfahren  $z$  zur Extraktion der  $k$ -ten Ziffer eines Schlüssels in  $\mathcal{O}(1)$  Schritten.

## Beispiel

$$z_{10}(0, 85) = 5$$

$$z_{10}(1, 85) = 8$$

$$z_{10}(2, 85) = 0$$

# Radix-Exchange-Sort

Schlüssel mit Radix 2.

Beobachtung: Wenn für ein  $k \geq 0$ :

$$z_2(i, x) = z_2(i, y) \text{ für alle } i > k$$

und

$$z_2(k, x) < z_2(k, y),$$

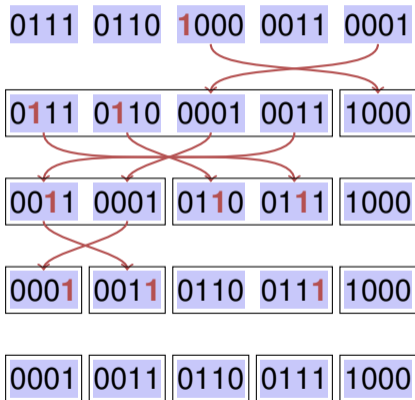
dann ist  $x < y$ .

# Radix-Exchange-Sort

Idee:

- Starte mit maximalem  $k$ .
- Binäres Aufteilen der Datensätze mit  $z_2(k, \cdot) = 0$  vs.  $z_2(k, \cdot) = 1$  wie bei Quicksort.
- $k \leftarrow k - 1$ .

# Radix-Exchange-Sort



# Algorithmus RadixExchangeSort( $A, l, r, b$ )

**Input:** Array  $A$  der Länge  $n$ , linke und rechte Grenze  $1 \leq l \leq r \leq n$ ,  
Bitposition  $b$

**Output:** Array  $A$ , im Bereich  $[l, r]$  nach Bits  $[0, \dots, b]$  sortiert.

**if**  $l < r$  **and**  $b \geq 0$  **then**

$i \leftarrow l - 1$

$j \leftarrow r + 1$

**repeat**

**repeat**  $i \leftarrow i + 1$  **until**  $z_2(b, A[i]) = 1$  **or**  $i \geq j$

**repeat**  $j \leftarrow j - 1$  **until**  $z_2(b, A[j]) = 0$  **or**  $i \geq j$

**if**  $i < j$  **then** swap( $A[i], A[j]$ )

**until**  $i \geq j$

    RadixExchangeSort( $A, l, i - 1, b - 1$ )

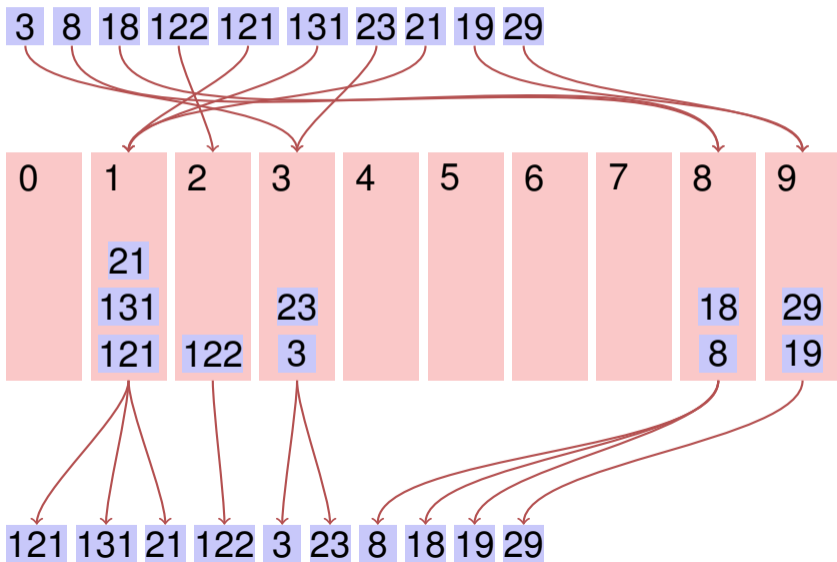
    RadixExchangeSort( $A, i, r, b - 1$ )

# Analyse

RadixExchangeSort ist rekursiv mit maximaler Rekursionstiefe = maximaler Anzahl Ziffern  $p$ .

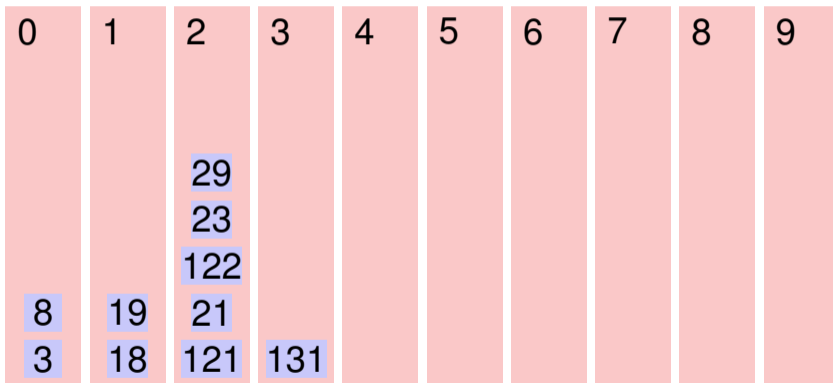
Laufzeit im schlechtesten Fall  $\mathcal{O}(p \cdot n)$ .

# Bucket Sort (Sortieren durch Fachverteilen)



# Bucket Sort (Sortieren durch Fachverteilen)

121 131 21 122 3 23 8 18 19 29

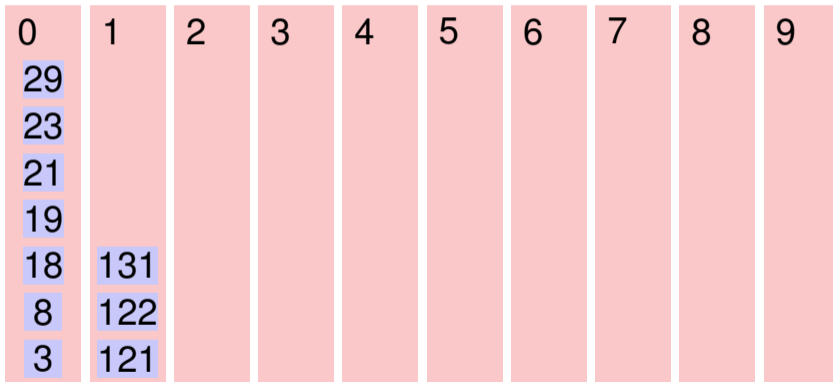


3 8 18 19 121 21 122 23 29



# Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 19 121 21 122 23 29



3 8 18 19 21 23 29 121 122 131 😊

# Implementationsdetails

Bucketgrösse sehr unterschiedlich. Möglichkeiten

- Verkettete Liste oder dynamisches Array für jede Ziffer.
- Ein Array der Länge  $n$ , Offsets für jede Ziffer in erstem Durchlauf bestimmen.

Annahmen: Eingabelänge  $n$ , Anzahl Bits / Ganzzahl:  $k$ , Anzahl Buckets:  $2^b$

Asymptotische Laufzeit  $\mathcal{O}\left(\frac{k}{b} \cdot (n + 2^b)\right)$ .

Zum Beispiel:  $k = 32$ ,  $2^b = 256$  :  $\frac{k}{b} \cdot (n + 2^b) = 4n + 1024$ .

# 11. Elementare Datenstrukturen

Abstrakte Datentypen Stapel, Warteschlange,  
Implementationsvarianten der verketteten Liste [Ottman/Widmayer,  
Kap. 1.5.1-1.5.2, Cormen et al, Kap. 10.1.-10.2]

# Abstrakte Datentypen

Wir erinnern uns<sup>17</sup> (Vorlesung Informatik I)

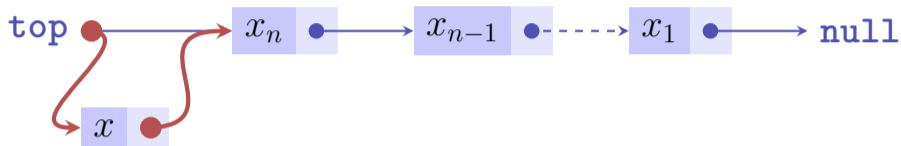
Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

- **push**( $x, S$ ): Legt Element  $x$  auf den Stapel  $S$ .
- **pop**( $S$ ): Entfernt und liefert oberstes Element von  $S$ , oder **null**.
- **top**( $S$ ): Liefert oberstes Element von  $S$ , oder **null**.
- **isEmpty**( $S$ ): Liefert **true** wenn Stack leer, sonst **false**.
- **emptyStack**(): Liefert einen leeren Stack.

---

<sup>17</sup>hoffentlich

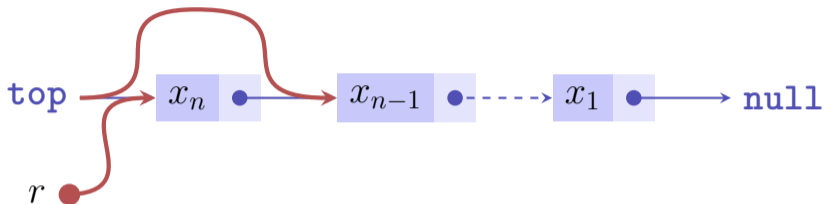
# Implementation Push



`push`( $x, S$ ):

- 1 Erzeuge neues Listenelement mit  $x$  und Zeiger auf den Wert von `top`.
- 2 Setze `top` auf den Knoten mit  $x$ .

# Implementation Pop



`pop(S)`:

- 1 Ist `top=null`, dann gib `null` zurück
- 2 Andernfalls merke Zeiger  $p$  von `top` in  $r$ .
- 3 Setze `top` auf  $p.next$  und gib  $r$  zurück

# Analyse

Jede der Operationen `push`, `pop`, `top` und `isEmpty` auf dem Stack ist in  $\mathcal{O}(1)$  Schritten ausführbar.

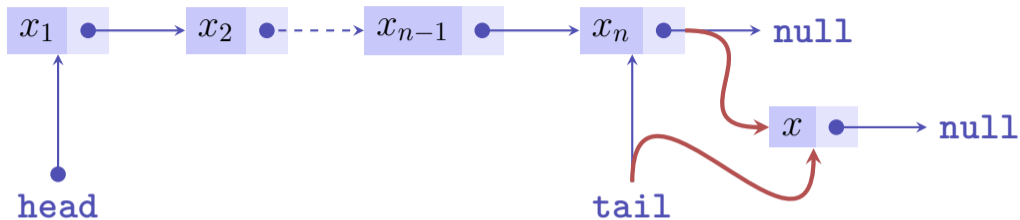
# Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

- **enqueue**( $x, Q$ ): fügt  $x$  am Ende der Schlange an.
- **dequeue**( $Q$ ): entfernt  $x$  vom Beginn der Schlange und gibt  $x$  zurück (**null** sonst.)
- **head**( $Q$ ): liefert das Objekt am Beginn der Schlange zurück (**null** sonst.)
- **isEmpty**( $Q$ ): liefert **true** wenn Queue leer, sonst **false**.
- **emptyQueue**( $Q$ ): liefert leere Queue zurück.



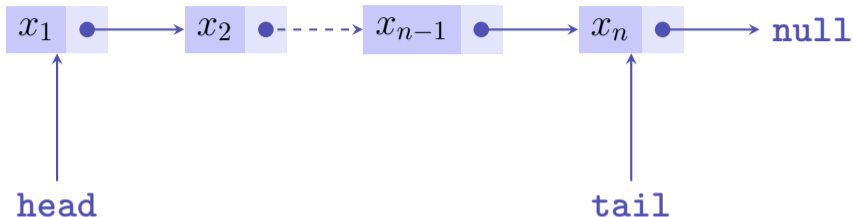
# Implementation Queue



`enqueue`( $x, S$ ):

- 1 Erzeuge neues Listenelement mit  $x$  und Zeiger auf `null`.
- 2 Wenn `tail`  $\neq$  `null`, setze `tail.next` auf den Knoten mit  $x$ .
- 3 Setze `tail` auf den Knoten mit  $x$ .
- 4 Ist `head` = `null`, dann setze `head` auf `tail`.

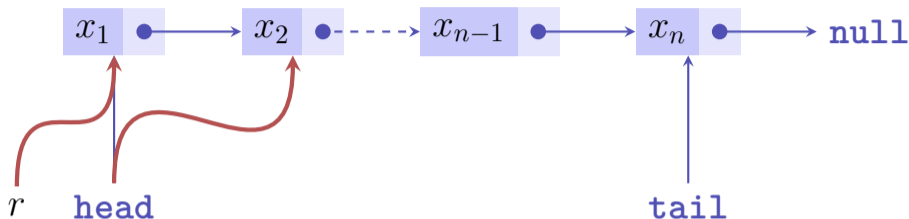
# Invarianten!



Mit dieser Implementation gilt

- entweder `head = tail = null`,
- oder `head = tail  $\neq$  null` und `head.next = null`
- oder `head  $\neq$  null` und `tail  $\neq$  null` und `head  $\neq$  tail` und `head.next  $\neq$  null`.

# Implementation Queue



`dequeue(S)`:

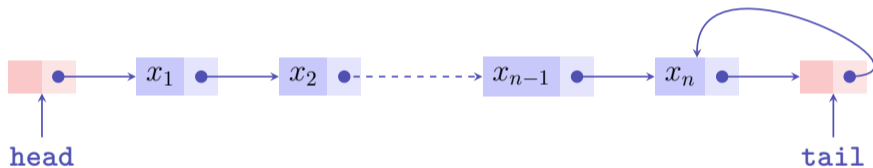
- 1 Merke Zeiger von `head` in  $r$ . Wenn  $r = \text{null}$ , gib  $r$  zurück.
- 2 Setze den Zeiger von `head` auf `head.next`.
- 3 Ist nun `head = null`, dann setze `tail` auf `null`.
- 4 Gib den Wert von  $r$  zurück.

# Analyse

Jede der Operationen `enqueue`, `dequeue`, `head` und `isEmpty` auf der Queue ist in  $\mathcal{O}(1)$  Schritten ausführbar.

# Implementationsvarianten verketteter Listen

Liste mit Dummy-Elementen (Sentinels).

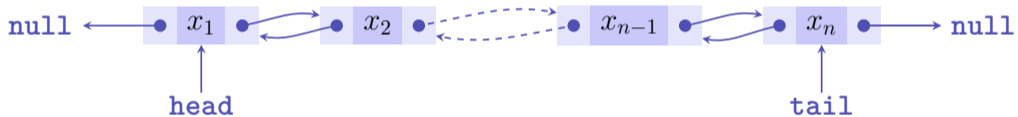


Vorteil: Weniger Spezialfälle!

Variante davon: genauso, dabei Zeiger auf ein Element immer einfach indirekt gespeichert. (Bsp: Zeiger auf  $x_3$  zeigt auf  $x_2$ .)

# Implementationsvarianten verketteter Listen

## Doppelt verkettete Liste



# Übersicht

	enqueue	delete	search	concat
(A)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
(B)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
(C)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
(D)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

(A) = Einfach verkettet

(B) = Einfach verkettet, mit Dummyelement am Anfang und Ende

(C) = Einfach verkettet, mit einfach indirekter Elementadressierung

(D) = Doppelt verkettet

# Prioritätswarteschlange (Priority Queue)

Priority Queue = Warteschlange mit Prioritäten.

Operationen

- `insert(x, p, Q)`: Füge Objekt  $x$  mit Priorität  $p$  ein.
- `extractMax(Q)`: Entferne Objekt  $x$  mit höchster Priorität und liefere es.



# Implementation Prioritätswarteschlange

Mit einem Max-Heap!

Also

- `insert` in Zeit  $\mathcal{O}(\log n)$  und
- `extractMax` in Zeit  $\mathcal{O}(\log n)$ .

# 12. Amortisierte Analyse

Amortisierte Analyse: Aggregat Analyse, Konto-Methode, Potentialmethode [Ottman/Widmayer, Kap. 3.3, Cormen et al, Kap. 17]

# Multistack

Multistack unterstützt neben `push` und `pop` noch

`multipop(s, S)`: Entferne die  $\min(\text{size}(S), k)$  zuletzt eingefügten Objekte und liefere diese zurück.

Implementation wie beim Stack. Laufzeit von `multipop` ist  $\mathcal{O}(k)$ .

# Akademische Frage

Führen wir auf einem Stack mit  $n$  Elementen  $n$  mal `multipop(k,S)` aus, kostet das dann  $\mathcal{O}(n^2)$ ?

Sicher richtig, denn jeder `multipop` kann Zeit  $\mathcal{O}(n)$  haben.

Wie bekommen wir eine schärfere Abschätzung?

# Amortisierte Analyse

- Obere Schranke: Abschätzung der *durchschnittlichen* Laufzeit jeder betrachteten Operation im *schlechtesten Fall*.

$$\frac{1}{n} \sum_{i=1}^n \text{Kosten}(\text{op}_i)$$

- Nutzt aus, dass wenige teure Operationen vielen billigen Operationen gegenüberstehen.
- In der amortisierten Analyse sucht man nach einer Kostenfunktion / einem Potential, um zu zeigen, wie die billigen Operationen für die teuren Operationen “aufkommen” können.

# Aggregierte Analyse

Direkte Argumentation: berechne eine Schranke für die Gesamtzahl der Elementaroperationen und teile durch die Anzahl der Operationen

# Aggregierte Analyse: (Stack)

- Bei  $n$  Operationen können insgesamt maximal  $n$  Elemente auf den Stack gelegt werden. Also können auch insgesamt nur maximal  $n$  Elemente vom Stack entfernt werden.
- Für die Gesamtkosten ergibt sich

$$\sum_{i=1}^n \text{Kosten}(\text{op}_i) \leq 2n$$

und somit

$$\textit{amortisierte Kosten}(\text{op}_i) \leq 2 \in \mathcal{O}(1)$$

# Kontomethode

## Modell

- Der Computer basiert auf Münzen: jede Elementaroperation der Maschine kostet eine Münze.
  - Für jede Operation  $op_k$  einer Datenstruktur wird eine bestimmte Anzahl Münzen  $a_k$  auf ein Konto  $A$  eingezahlt:  $A_k = A_{k-1} + a_k$
  - Die Münzen vom Konto  $A$  werden verwendet, um die anfallenden echten Kosten  $t_k$  zu bezahlen.
  - Das Konto  $A$  muss zu jeder Zeit genügend Münzen aufweisen, um die laufende Operation  $op_k$  zu bezahlen:  $A_k - t_k \geq 0 \forall k$ .
- $\Rightarrow a_k$  sind die amortisierten Kosten der Operation  $op_k$ .



# Kontomethode (Stack)

- Aufruf von **push**: kostet 1 CHF und zusätzlich kommt 1 CHF auf das Bankkonto ( $a_k = 2$ )
- Aufruf von **pop**: kostet 1 CHF, wird durch Rückzahlung vom Bankkonto beglichen. ( $a_k = 0$ )

Kontostand wird niemals negativ.

$a_k \leq 2 \forall k$ , also: konstante amortisierte Kosten.

# Potentialmethode

Leicht anderes Modell

- Definiere ein *Potential*  $\Phi_i$ , welches *zum Zustand der betrachteten Datenstruktur* zum Zeitpunkt  $i$  gehört.
- Das Potential soll zum Ausgleichen teurer Operationen verwendet werden und muss daher so gewählt sein, dass es bei (häufigen) günstigen Operationen erhöht wird, während es die (seltenen) teuren Operationen durch einen fallenden Wert bezahlt.

# Potentialmethode (formal)

Bezeichne  $t_i$  die realen Kosten der Operation  $op_i$ .

Potentialfunktion  $\Phi_i \geq 0$  zur Datenstruktur nach  $i$  Operationen.

Voraussetzung:  $\Phi_i \geq \Phi_0 \forall i$ .

*Amortisierte Kosten* der  $i$ -ten Operation:

$$a_i := t_i + \Phi_i - \Phi_{i-1}.$$

Es gilt nämlich

$$\sum_{i=1}^n a_i = \sum_{i=1}^n (t_i + \Phi_i - \Phi_{i-1}) = \left( \sum_{i=1}^n t_i \right) + \Phi_n - \Phi_0 \geq \sum_{i=1}^n t_i.$$

# Beispiel Stack

Potentialfunktion  $\Phi_i =$  Anzahl Elemente auf dem Stack.

- **push**( $x, S$ ): Reale Kosten  $t_i = 1$ .  $\Phi_i - \Phi_{i-1} = 1$ . Amortisierte Kosten  $a_i = 2$ .
- **pop**( $S$ ): Reale Kosten  $t_i = 1$ .  $\Phi_i - \Phi_{i-1} = -1$ . Amortisierte Kosten  $a_i = 0$ .
- **multipop**( $k, S$ ): Reale Kosten  $t_i = k$ .  $\Phi_i - \Phi_{i-1} = -k$ . Amortisierte Kosten  $a_i = 0$ .

Alle Operationen haben *konstante amortisierte Kosten*! Im Durchschnitt hat also Multipop konstanten Zeitbedarf. <sup>18</sup>

---

<sup>18</sup>Achtung: es geht nicht um den probabilistischen Mittelwert sondern den (worst-case) Durchschnitt der Kosten.

# Beispiel Binärer Zähler

Binärer Zähler mit  $k$  bits. Im schlimmsten Fall für jede Zähloperation maximal  $k$  Bitflips. Also  $\mathcal{O}(n \cdot k)$  Bitflips für Zählen von 1 bis  $n$ . Geht das besser?

Reale Kosten  $t_i =$  Anzahl Bitwechsel von 0 nach 1 plus Anzahl Bitwechsel von 1 nach 0.

$$\dots 0 \underbrace{1111111}_{l \text{ Einsen}} + 1 = \dots 1 \underbrace{0000000}_{l \text{ Nullen}}.$$

$$\Rightarrow t_i = l + 1$$

# Binärer Zähler: Aggregatmethode

Zähle die Anzahl Bitwechsel beim Zählen von 0 bis  $n - 1$ .

Beobachtung

- Bit 0 wechselt für jedes  $k - 1 \rightarrow k$
- Bit 1 wechselt für jedes  $2k - 1 \rightarrow 2k$
- Bit 2 wechselt für jedes  $4k - 1 \rightarrow 4k$

Gesamte Anzahl Bitwechsel  $\sum_{i=0}^{n-1} \frac{n}{2^i} \leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$

Amortisierte Kosten für jede Erhöhung:  $\mathcal{O}(1)$  Bitwechsel.

# Binärer Zähler: Kontomethode

Beobachtung: bei jedem Inkrementieren wird genau ein Bit auf 1 gesetzt, während viele Bits auf 0 gesetzt werden könnten. Nur ein vorgängig auf 1 gesetztes Bit kann wieder auf 0 zurückgesetzt werden.

$a_i = 2$ : 1 CHF reale Kosten für das Setzen  $0 \rightarrow 1$  plus 1 CHF für das Konto. Jedes Zurücksetzen  $1 \rightarrow 0$  kann vom Konto beglichen werden.

# Binärer Zähler: Potentialmethode

$$\dots 0 \underbrace{1111111}_l \text{ Einsen} + 1 = \dots 1 \underbrace{0000000}_l \text{ Nullen}$$

Potentialfunktion  $\Phi_i$ : Anzahl der 1-Bits von  $x_i$ .

$$\Rightarrow \Phi_0 = 0 \leq \Phi_i \forall i$$

$$\Rightarrow \Phi_i - \Phi_{i-1} = 1 - l,$$

$$\Rightarrow a_i = t_i + \Phi_i - \Phi_{i-1} = l + 1 + (1 - l) = 2.$$

Amortisiert konstante Kosten für eine Zähloperation. 😊



# 13. Wörterbücher

Wörterbuch, Selbstandornung, Implementation Wörterbuch mit Array / Liste / Skipliste. [Ottman/Widmayer, Kap. 3.3,1.7, Cormen et al, Kap. Problem 17-5]

# Wörterbuch (Dictionary)

ADT zur Verwaltung von Schlüsseln aus  $\mathcal{K}$  mit Operationen

- **insert**( $k, D$ ): Hinzufügen von  $k \in \mathcal{K}$  in Wörterbuch  $D$ . Bereits vorhanden  $\Rightarrow$  Fehlermeldung.
- **delete**( $k, D$ ): Löschen von  $k$  aus dem Wörterbuch  $D$ . Nicht vorhanden  $\Rightarrow$  Fehlermeldung.
- **search**( $k, D$ ): Liefert **true** wenn  $k \in D$ , sonst **false**.

# Idee

Implementiere Wörterbuch als sortiertes Array.

Anzahl Elementaroperationen im schlechtesten Fall

Suchen  $\mathcal{O}(\log n)$  😊




Einfügen  $\mathcal{O}(n)$  😞

Löschen  $\mathcal{O}(n)$  😞

# Andere Idee

Implementiere Wörterbuch als verkettete Liste

Anzahl Elementaroperationen im schlechtesten Fall

Suchen	$\mathcal{O}(n)$	
Einfügen	$\mathcal{O}(1)$ <sup>19</sup>	
Löschen	$\mathcal{O}(n)$	

---

<sup>19</sup>Unter der Voraussetzung, dass wir die Existenz nicht prüfen wollen.

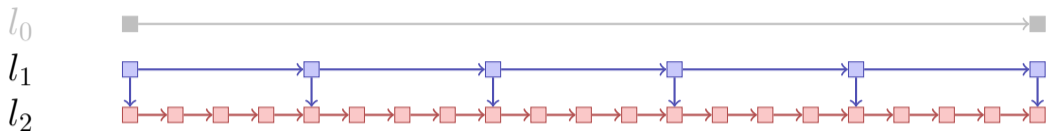
## 13.1 Skiplisten

# Sortierte Verkettete Liste



Element / Einfügeort suchen: *worst-case*  $n$  Schritte.

# Sortierte Verkettete Liste mit 2 Ebenen



■ Anzahl Elemente:  $n_0 := n$

■ Schrittweite Ebene 1:  $n_1$

■ Schrittweite Ebene 2:  $n_2 = 1$

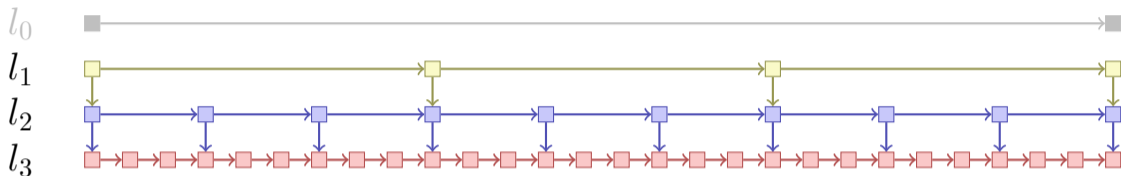
⇒ Element / Einfügeort suchen: worst-case  $\frac{n_0}{n_1} + \frac{n_1}{n_2}$ .

⇒ Beste Wahl für<sup>20</sup>  $n_1$ :  $n_1 = \frac{n_0}{n_1} = \sqrt{n_0}$ .

Element / Einfügeort suchen: **worst-case**  $2\sqrt{n}$  Schritte.

<sup>20</sup>Differenzieren und 0 setzen, siehe Anhang

# Sortierte Verkettete Liste mit 3 Ebenen



- Anzahl Elemente:  $n_0 := n$
- Schrittweiten Ebenen  $0 < i < 3$ :  $n_i$
- Schrittweite auf Ebene 3:  $n_3 = 1$

⇒ Beste Wahl für  $(n_1, n_2)$ :  $n_2 = \frac{n_0}{n_1} = \frac{n_1}{n_2} = \sqrt[3]{n_0}$ .

Element / Einfügeort suchen: *worst-case*  $3 \cdot \sqrt[3]{n}$  Schritte.



# Sortierte Verkettete Liste mit $k$ Ebenen (Skipliste)

- Anzahl Elemente:  $n_0 := n$
- Schrittweiten Ebenen  $0 < i < k$ :  $n_i$
- Schrittweite auf Ebene  $k$ :  $n_k = 1$

⇒ Beste Wahl für  $(n_1, \dots, n_k)$ :  $n_{k-1} = \frac{n_0}{n_1} = \frac{n_1}{n_2} = \dots = \sqrt[k]{n_0}$ .

Element / Einfügeort suchen: *worst-case*  $k \cdot \sqrt[k]{n}$

Schritte<sup>21</sup> (Derivation: Appendix).

Annahme:  $n = 2^k$

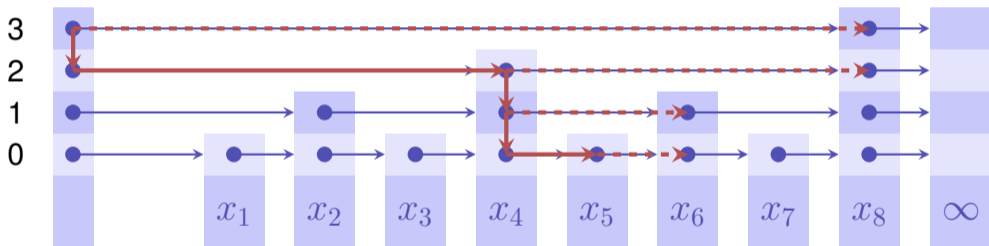
⇒ worst case  $\log_2 n \cdot 2$  Schritte und  $\frac{n_i}{n_{i+1}} = 2 \forall 0 \leq i < \log_2 n$ .

---

<sup>21</sup>(Herleitung: Anhang)

# Suche in Skipliste

## Perfekte Skipliste



$x_1 \leq x_2 \leq x_3 \leq \dots \leq x_9$ .

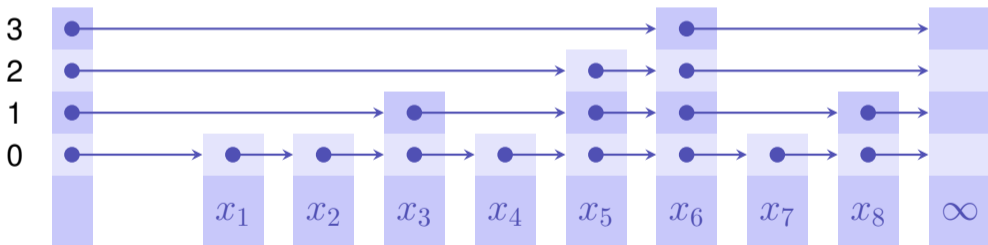
Beispiel: Suche nach einem Schlüssel  $x$  mit  $x_5 < x < x_6$ .

# Analyse Perfekte Skipliste (schlechtester Fall)

Suchen in  $\mathcal{O}(\log n)$ . Einfügen in  $\mathcal{O}(n)$ .

# Randomisierte Skipliste

Idee: Füge jeweils einen Knoten mit zufälliger Höhe  $H$  ein, wobei  $\mathbb{P}(H = i) = \frac{1}{2^{i+1}}$ .



# Analyse Randomisierte Skipliste

## Theorem

*Die Anzahl an erwarteten Elementaroperationen für Suchen, Einfügen und Löschen eines Elements in einer randomisierten Skipliste ist  $\mathcal{O}(\log n)$ .*

Der längliche Beweis, welcher im Rahmen dieser Vorlesung nicht geführt wird, betrachtet die Länge eines Weges von einem gesuchten Knoten zurück zum Startpunkt im höchsten Level.

## 13.2 [Selbstanordnung]

nicht im Unterricht präsentiert

# Selbstanordnung

Problematisch bei der Verwendung verketteter Listen: lineare Suchzeit

*Idee:* Versuche, die Listenelemente so anzuordnen, dass Zugriffe über die Zeit hinweg schneller möglich sind

Zum Beispiel

- Transpose: Bei jedem Zugriff auf einen Schlüssel wird dieser um eine Position nach vorne bewegt.
- Move-to-Front (MTF): Bei jedem Zugriff auf einen Schlüssel wird dieser ganz nach vorne bewegt.

# Transpose

Transpose:

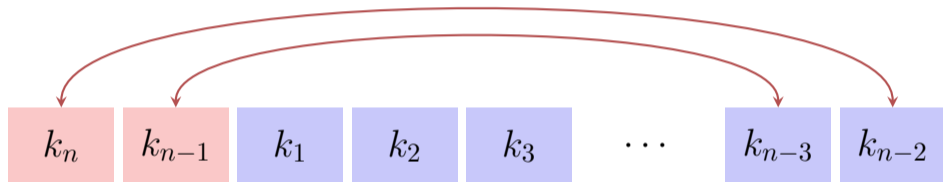


Worst case:  $n$  Wechselseitige Zugriffe auf  $k_{n-1}$  und  $k_n$ . Laufzeit:  $\Theta(n^2)$



# Move-to-Front

Move-to-Front:



$n$  Wechselnde Zugriffe auf  $k_{n-1}$  und  $k_n$ . Laufzeit:  $\Theta(n)$

Man kann auch hier Folge mit quadratischer Laufzeit angeben, z.B. immer das letzte Element. Aber dafür ist keine offensichtliche Strategie bekannt, die viel besser sein könnte als MTF.

# Analyse

Vergleichen MTF mit dem bestmöglichen Konkurrenten (Algorithmus) A. Wie viel besser kann A sein?

Annahmen:

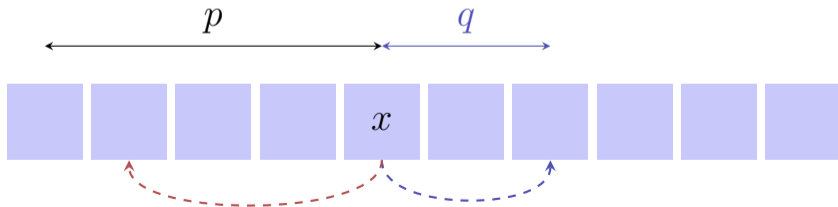
- MTF und A dürfen jeweils nur das zugegriffene Element  $x$  verschieben.
- MTF und A starten mit derselben Liste.

$M_k$  und  $A_k$  bezeichnen die Liste nach dem  $k$ -ten Schritt.  $M_0 = A_0$ .

# Analyse

Kosten:

- Zugriff auf  $x$ : Position  $p$  von  $x$  in der Liste.
- Keine weiteren Kosten, wenn  $x$  **vor**  $p$  verschoben wird.
- Weitere Kosten  $q$  für jedes Element, das  $x$  von  $p$  aus nach **hinten** verschoben wird.



# Amortisierte Analyse

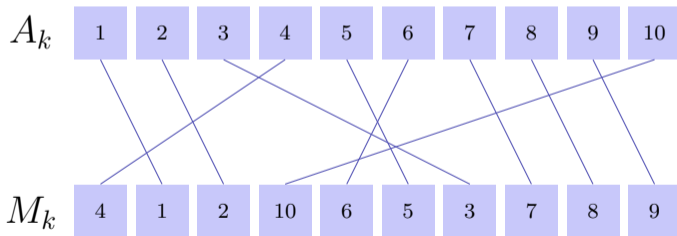
Sei eine beliebige Folge von Suchanfragen gegeben und seien  $G_k^{(M)}$  und  $G_k^{(A)}$  jeweils die Kosten im Schritt  $k$  für Move-to-Front und A. Suchen Abschätzung für  $\sum_k G_k^{(M)}$  im Vergleich zu  $\sum_k G_k^{(A)}$ .

⇒ Amortisierte Analyse mit Potentialfunktion  $\Phi$ .

# Potentialfunktion

Potentialfunktion  $\Phi =$  Anzahl der Inversionen von  $A$  gegen MTF.

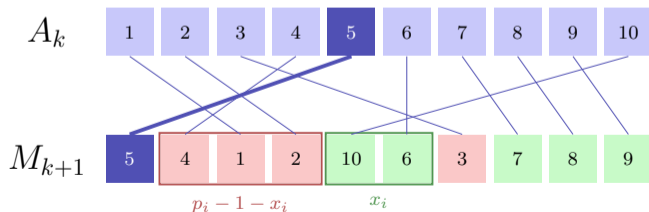
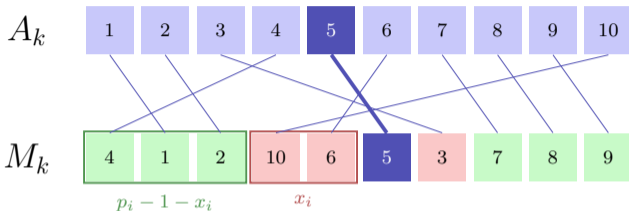
Inversion = Paar  $x, y$  so dass für die Positionen von  $x$  und  $y$   
 $(p^{(A)}(x) < p^{(A)}(y)) \neq (p^{(M)}(x) < p^{(M)}(y))$



#Inversionen = #Kreuzungen

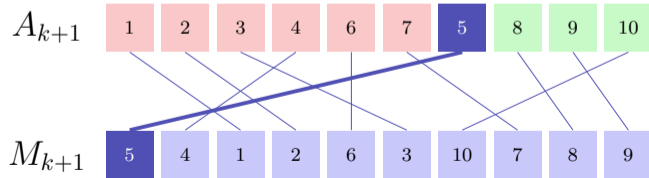
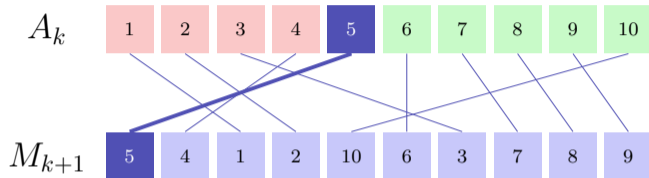
# Abschätzung der Potentialfunktion: MTF

- Element  $i$  an Position  $p_i := p^{(M)}(i)$ .
- Zugriffskosten  $C_k^{(M)} = p_i$ .
- $x_i$ : Anzahl Elemente, die in M vor  $p_i$  und in A nach  $i$  stehen.
- MTF löst  $x_i$  Inversionen auf.
- $p_i - x_i - 1$ : Anzahl Elemente, die in M vor  $p_i$  und in A vor  $i$  stehen.
- MTF erzeugt  $p_i - 1 - x_i$  Inversionen.



# Abschätzung der Potentialfunktion: A

- Element  $i$  an Position  $p^{(A)}(i)$ .
- $X_k^{(A)}$ : Anzahl Verschiebungen nach hinten (sonst 0).
- Zugriffskosten für  $i$ :  
 $C_k^{(A)} = p^{(A)}(i) \geq p^{(M)}(i) - x_i$ .
- A erhöht die Anzahl Inversionen höchstens um  $X_k^{(A)}$ .



# Abschätzung

$$\Phi_{k+1} - \Phi_k \leq -x_i + (p_i - 1 - x_i) + X_k^{(A)}$$

Amortisierte Kosten von MTF im  $k$ -ten Schritt:

$$\begin{aligned} a_k^{(M)} &= C_k^{(M)} + \Phi_{k+1} - \Phi_k \\ &\leq p_i - x_i + (p_i - 1 - x_i) + X_k^{(A)} \\ &= (p_i - x_i) + (p_i - x_i) - 1 + X_k^{(A)} \\ &\leq C_k^{(A)} + C_k^{(A)} - 1 + X_k^{(A)} \leq 2 \cdot C_k^{(A)} + X_k^{(A)}. \end{aligned}$$



# Abschätzung

Kosten Summiert

$$\begin{aligned}\sum_k G_k^{(M)} &= \sum_k C_k^{(M)} \leq \sum_k a_k^{(M)} \leq \sum_k 2 \cdot C_k^{(A)} + X_k^{(A)} \\ &\leq 2 \cdot \sum_k C_k^{(A)} + X_k^{(A)} \\ &= 2 \cdot \sum_k G_k^{(A)}\end{aligned}$$

MTF führt im schlechtesten Fall höchstens doppelt so viele Operationen aus wie eine optimale Strategie.

# 13.3 Anhang

Mathematik zur Skipliste

# [Mathematik zur $k$ -Level Skipliste]

Gegeben Anzahl Datenpunkte  $n_0$ , Anzahl Level  $k > 0$  und Anzahl Elemente  $n_l$  die pro Level  $l$  übersprungen werden,  $n_k = 1$ .  
Maximale Anzahl totaler Schritte in der Skipliste:

$$f(\vec{n}) = \frac{n_0}{n_1} + \frac{n_1}{n_2} + \dots + \frac{n_{k-1}}{n_k}$$

Minimiere  $f$  für  $(n_1, \dots, n_{k-1})$ :  $\frac{\partial f(\vec{n})}{\partial n_t} = 0$  für alle  $0 < t < k$ ,

$$\frac{\partial f(\vec{n})}{\partial n_t} = -\frac{n_{t-1}}{n_t^2} + \frac{1}{n_{t+1}} = 0 \Rightarrow n_{t+1} = \frac{n_t^2}{n_{t-1}} \text{ und } \frac{n_{t+1}}{n_t} = \frac{n_t}{n_{t-1}}.$$

# [Mathematik zur $k$ -Level Skipliste]

Vorige Folie  $\Rightarrow \frac{n_t}{n_0} = \frac{n_t}{n_{t-1}} \frac{n_{t-1}}{n_{t-2}} \cdots \frac{n_1}{n_0} = \left( \frac{n_1}{n_0} \right)^t$

Insbesondere  $1 = n_k = \frac{n_1^k}{n_0^{k-1}} \Rightarrow n_1 = \sqrt[k]{n_0^{k-1}}$

Also  $n_{k-1} = \frac{n_0}{n_1} = \sqrt[k]{\frac{n_0^k}{n_0^{k-1}}} = \sqrt[k]{n_0}$ .

Maximale Anzahl Schritte in der Skipliste  $f(\vec{n}) = k \cdot (\sqrt[k]{n_0})$

Angenommen  $n_0 = 2^k$ , dann  $\frac{n_l}{n_{l+1}} = 2$  für alle  $0 \leq l < k$  (Skipliste halbiert die Daten in jedem Level), und  $f(n) = k \cdot 2 = 2 \log_2 n \in \Theta(\log n)$ .

# 14. Hashing

Hashtabellen, Pre-Hashing, Hashing, Kollisionsauflösung durch Verketteten, Einfaches gleichmässiges Hashing, Gebräuchliche Hashfunktionen, Tabellenvergrösserung, offene Addressierung: Sondieren, Gleichmässiges Hashing, Universelles hashing, Perfektes Hashing [Ottman/Widmayer, Kap. 4.1-4.3.2, 4.3.4, Cormen et al, Kap. 11-11.4]

# Motivierendes Beispiel

*Ziel:* Effiziente Verwaltung einer Tabelle aller  $n$  ETH-Studenten

*Mögliche Anforderung:* Schneller Zugriff (Einfügen, Löschen, Finden) von Datensätzen nach Name.

# Wörterbuch (Dictionary)

Abstrakter Datentyp (ADT)  $D$  zur Verwaltung einer Menge von Einträgen<sup>22</sup>  $i$  mit Schlüsseln  $k \in \mathcal{K}$ . Operationen

- $D.\text{insert}(i)$ : Hinzufügen oder Überschreiben von  $i$  im Wörterbuch  $D$ .
- $D.\text{delete}(i)$ : Löschen von  $i$  aus dem Wörterbuch  $D$ . Nicht vorhanden  $\Rightarrow$  Fehlermeldung.
- $D.\text{search}(k)$ : Liefert Eintrag mit Schlüssel  $k$ , wenn er existiert.

---

<sup>22</sup>Schlüssel-Wert Paare  $(k, v)$ , im Folgenden betrachten wir hauptsächlich die Schlüssel.

# Wörterbuch in C++

*Assoziativer Container* `std::unordered_map<>`

```
// Create an unordered_map of strings that map to strings
std::unordered_map<std::string, std::string> u = {
    {"RED", "#FF0000"}, {"GREEN", "#00FF00"}
};

u["BLUE"] = "#0000FF"; // Add

std::cout << "The HEX of color RED is: " << u["RED"] << "\n";

for( const auto& n : u ) // iterate over key-value pairs
    std::cout << n.first << ":" << n.second << "\n";
```



# Motivation / Verwendung

Wahrscheinlich *die* gängigste Datenstruktur

- Unterstützt in vielen Programmiersprachen (C++, Java, Python, Ruby, Javascript, C# ...)
- Offensichtliche Verwendung
  - Datenbanken / Tabellenkalkulation
  - Symboltabellen in Compilern und Interpretern
- Weniger offensichtlich
  - Substring Suche (Google, grep)
  - Ähnlichkeit von Texten (Dokumentenvergleich, DNA)
  - Dateisynchronisation
  - Kryptographie: Filetransfer / Identifikation

# 1. Idee: Direkter Zugriff (Array)

Index	Eintrag
0	-
1	-
2	-
3	[3,wert(3)]
4	-
5	-
⋮	⋮
k	[k,wert(k)]
⋮	⋮

## *Probleme*

- 1 Schlüssel müssen nichtnegative ganze Zahlen sein
- 2 Grosser Schlüsselbereich  $\Rightarrow$  grosses Array

# Lösung zum ersten Problem: Pre-hashing

Prehashing: Bilde Schlüssel ab auf positive Ganzzahlen mit einer Funktion  $ph : \mathcal{K} \rightarrow \mathbb{N}$

- Theoretisch immer möglich, denn jeder Schlüssel ist als Bitsequenz im Computer gespeichert
- Theoretisch auch:  $x = y \Leftrightarrow ph(x) = ph(y)$
- In der Praxis: APIs bieten Funktionen zum pre-hashing an. (Java: `object.hashCode()`, C++: `std::hash<>`, Python: `hash(object)`)
- APIs bilden einen Schlüssel aus der Schlüsselmenge ab auf eine Ganzzahl mit beschränkter Grösse.<sup>23</sup>

---

<sup>23</sup>Somit gilt die Implikation  $ph(x) = ph(y) \Rightarrow x = y$  nicht mehr für alle  $x, y$ .

# Prehashing Beispiel: String

Zuordnung Name  $s = s_1 s_2 \dots s_{l_s}$  zu Schlüssel

$$ph(s) = \left( \sum_{i=1}^{l_s} s_{l_s-i+1} \cdot b^i \right) \bmod 2^w$$

$b$  so, dass verschiedene Namen möglichst verschiedene Schlüssel erhalten.

$w$  Wortgröße des Systems (z.B. 32 oder 64).

**Beispiel (Java), mit  $b = 31$ ,  $w = 32$  Ascii-Werte  $s_i$ .**

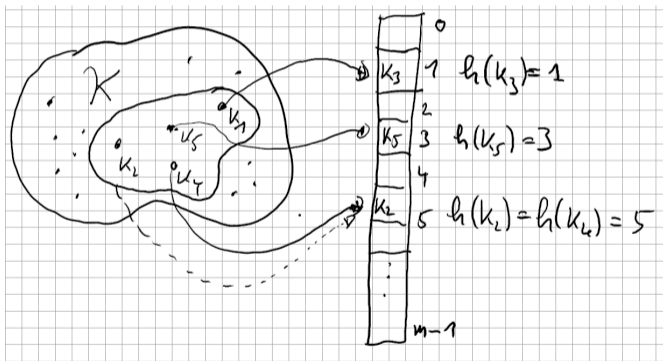
Anna  $\mapsto$  2045632

Jacqueline  $\mapsto$  2042089953442505  $\bmod 2^{32} = 507919049$

# Lösung zum zweiten Problem: Hashing

Reduziere des Schlüsseluniversum: Abbildung (Hash-Funktion)

$h : \mathcal{K} \rightarrow \{0, \dots, m - 1\}$  ( $m \approx n =$  Anzahl Einträge in der Tabelle)



Kollision:  $h(k_i) = h(k_j)$ .

# Nomenklatur

*Hashfunktion*  $h$ : Abbildung aus der Menge der Schlüssel  $\mathcal{K}$  auf die Indexmenge  $\{0, 1, \dots, m - 1\}$  eines Arrays (*Hashtabelle*).

$$h : \mathcal{K} \rightarrow \{0, 1, \dots, m - 1\}.$$

Meist  $|\mathcal{K}| \gg m$ . Es gibt also  $k_1, k_2 \in \mathcal{K}$  mit  $h(k_1) = h(k_2)$  (*Kollision*).

Eine Hashfunktion sollte die Menge der Schlüssel möglichst gleichmässig auf die Positionen der Hashtabelle verteilen.

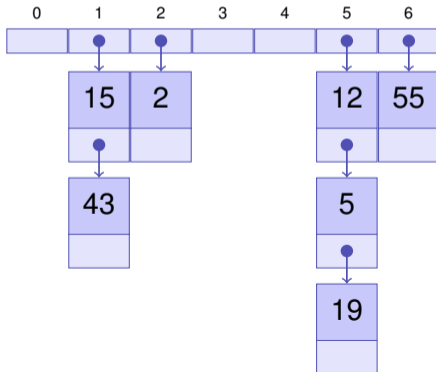
# Behandlung von Kollisionen: Verkettung

Beispiel  $m = 7$ ,  $\mathcal{K} = \{0, \dots, 500\}$ ,  $h(k) = k \bmod m$ .

Schlüssel 12, 55, 5, 15, 2, 19, 43

Direkte Verkettung der Überläufer

Hashtabelle



Überläufer

# Algorithmen zum Hashing mit Verkettung

- **insert**( $i$ ) Prüfe ob Schlüssel  $k$  vom Eintrag  $i$  in Liste an Position  $h(k)$ . Falls nein, füge  $i$  am Ende der Liste ein; andernfalls ersetze das Element durch  $i$ .
- **find**( $k$ ) Prüfe ob Schlüssel  $k$  in Liste an Position  $h(k)$ . Falls ja, gib die Daten zum Schlüssel  $k$  zurück. Andernfalls Rückgabe eines leeren Elements **null**.
- **delete**( $k$ ) Durchsuche die Liste an Position  $h(k)$  nach  $k$ . Wenn Suche erfolgreich, entferne das entsprechende Listenelement.



# Worst-case Analyse

Schlechtester Fall: alle Schlüssel werden auf den gleichen Index abgebildet.

⇒  $\Theta(n)$  pro Operation im schlechtesten Fall. 😞

# Einfaches Gleichmässiges Hashing

*Starke Annahmen:* Jeder beliebige Schlüssel wird

- mit gleicher Wahrscheinlichkeit (Uniformität)
- und unabhängig von den anderen Schlüsseln (Unabhängigkeit)

auf einen der  $m$  verfügbaren Slots abgebildet.

# Einfaches Gleichmässiges Hashing

Unter der Voraussetzung von einfachem gleichmässigen Hashing:  
*Erwartete Länge* einer Kette, wenn  $n$  Elemente in eine Hashtabelle mit  $m$  Elementen eingefügt werden

$$\begin{aligned}\mathbb{E}(\text{Länge Kette } j) &= \mathbb{E} \left( \sum_{i=0}^{n-1} \mathbb{1}(k_i = j) \right) = \sum_{i=0}^{n-1} \mathbb{P}(k_i = j) \\ &= \sum_{i=1}^n \frac{1}{m} = \frac{n}{m}\end{aligned}$$

$\alpha = n/m$  heisst *Belegungsfaktor* oder *Füllgrad* der Hashtabelle.

# Einfaches Gleichmässiges Hashing

## Theorem

*Sei eine Hashtabelle Verkettung gefüllt mit Füllgrad  $\alpha = \frac{n}{m} < 1$ .  
Unter der Annahme vom einfachen gleichmässigen Hashing hat die  
nächste Operation erwartete Laufzeitkosten von  $\leq 1 + \alpha$ .*

Folgerung: ist die Anzahl der Slots  $m$  der Hashtabelle immer mindestens proportional zur Anzahl Elemente  $n$  in der Hashtabelle,  $n \in \mathcal{O}(m) \Rightarrow$  Erwartete Laufzeit der Operationen Suchen, Einfügen und Löschen ist  $\mathcal{O}(1)$ .

# Weitere Analyse (direkt verkettete Liste)

- 1 Erfolglose Suche. Durchschnittliche Listenlänge ist  $\alpha = \frac{n}{m}$ . Liste muss ganz durchlaufen werden.

⇒ Durchschnittliche Anzahl betrachteter Einträge

$$C'_n = \alpha.$$

- 2 Erfolgreiche Suche. Betrachten die Einfügeschichtorie: Schlüssel  $j$  sieht durchschnittliche Listenlänge  $(j - 1)/m$ .

⇒ Durchschnittliche Anzahl betrachteter Einträge

$$C_n = \frac{1}{n} \sum_{j=1}^n (1 + (j - 1)/m) = 1 + \frac{1}{n} \frac{n(n - 1)}{2m} \approx 1 + \frac{\alpha}{2}.$$

# Vor und Nachteile der Verkettung

Vorteile der Strategie:

- Belegungsfaktoren  $\alpha > 1$  möglich
- Entfernen von Schlüsseln einfach

Nachteile

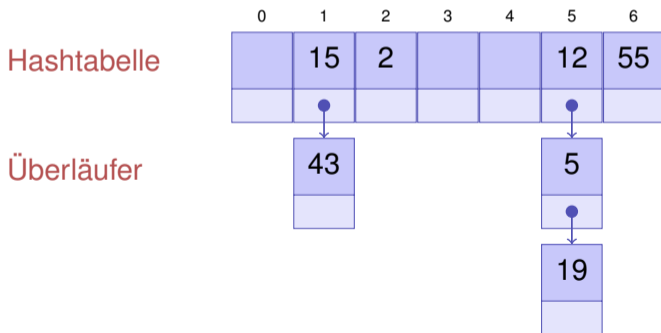
- Speicherverbrauch der Verkettung

# [Variante:Indirekte Verkettung]

Beispiel  $m = 7$ ,  $\mathcal{K} = \{0, \dots, 500\}$ ,  $h(k) = k \bmod m$ .

Schlüssel 12, 55, 5, 15, 2, 19, 43

Indirekte Verkettung der Überläufer



# Beispiele gebräuchlicher Hashfunktionen

$$h(k) = k \bmod m$$

Ideal:  $m$  Primzahl, nicht zu nahe bei Potenzen von 2 oder 10

Aber oft:  $m = 2^k - 1$  ( $k \in \mathbb{N}$ )



# Beispiele gebräuchlicher Hashfunktionen

## Multiplikationsmethode

$$h(k) = \lfloor (a \cdot k \bmod 2^w) / 2^{w-r} \rfloor \bmod m$$

- $m = 2^r$ ,  $w =$  Grösse des Maschinenworts in Bits.
- Multiplikation addiert  $k$  entlang aller Bits von  $a$ , Ganzzahldivision mit  $2^{w-r}$  und  $\bmod m$  extrahiert die oberen  $r$  Bits.
- Als Code geschrieben:  $a * k \gg (w-r)$
- Guter Wert für  $a$ :  $\lfloor \frac{\sqrt{5}-1}{2} \cdot 2^w \rfloor$ : Integer, der die ersten  $w$  Bits des gebrochenen Teils der irrationalen Zahl darstellt.

# Illustration

$$\begin{array}{r} \leftarrow w \text{ bits} \rightarrow \\ \begin{array}{|c|} \hline k \\ \hline \end{array} \begin{array}{l} k \\ a \end{array} \\ \times \begin{array}{|c|c|} \hline 11 & 1 \\ \hline \end{array} \\ \hline \begin{array}{|c|} \hline k \\ \hline \end{array} \\ + \begin{array}{|c|} \hline k \\ \hline \end{array} \\ + \begin{array}{|c|} \hline k \\ \hline \end{array} \\ = \begin{array}{|c|c|} \hline & \leftarrow r \text{ bits} \rightarrow \\ \hline \end{array} \\ \hline \gg (w - r) \begin{array}{|c|c|} \hline 0 & \leftarrow r \text{ bits} \rightarrow \\ \hline \end{array} \end{array}$$

# Tabellenvergrößerung

- Wissen nicht a priori, wie gross  $n$  sein wird.
- Benötigen  $m = \Theta(n)$  zu jeder Zeit.

Grösse der Tabelle muss angepasst werden. Hash-Funktion ändert sich  $\Rightarrow$  *Rehashing*

- Alloziere Array  $A'$  mit Grösse  $m' > m$
- Füge jeden Eintrag von  $A$  erneut in  $A'$  ein (mit erneutem Hashing)
- Setze  $A \leftarrow A'$ .
- Kosten:  $\mathcal{O}(n + m + m')$ .

Wie wählt man  $m'$ ?

# Tabellenvergrößerung

- 1.Idee  $n = m \Rightarrow m' \leftarrow m + 1$

Bei jedem Einfügen vergrössern. Kosten

$$\Theta(1 + 2 + 3 + \dots + n) = \Theta(n^2) \text{ 😞}$$

- 2.Idee  $n = m \Rightarrow m' \leftarrow 2m$  Vergrössern nur wenn  $m = 2^i$ :

$$\Theta(1 + 2 + 4 + 8 + \dots + n) = \Theta(n)$$

Einige Einfügeoperationen kosten lineare Zeit, aber im Durchschnitt kosten sie  $\Theta(1)$  😊

Jede Operation vom Hashing mit Verketteten hat erwartet amortisierte Kosten  $\Theta(1)$ .

( $\Rightarrow$  Amortisierte Analyse)

# Offene Adressierung<sup>24</sup>

Speichere die Überläufer direkt in der Hashtabelle mit einer *Sondierungsfunktion*  $s : \mathcal{K} \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$   
Tabellenposition des Schlüssels entlang der *Sondierungsfolge*

$$S(k) := (s(k, 0), s(k, 1), \dots, s(k, m - 1)) \quad \text{mod } m$$

Sondierungsfolge muss für jedes  $k \in \mathcal{K}$  eine Permutation sein von  $\{0, 1, \dots, m - 1\}$

---

<sup>24</sup>Begriffsklärung: Dieses Verfahren nutzt *offene Adressierung* (Positionen in der Hashtabelle nicht fixiert), ist aber *ein geschlossenes Hashverfahren* (Einträge bleiben in der Hashtabelle)

# Algorithmen zur offenen Adressierung

- **insert**( $i$ ) Suche Schlüssel  $k$  von  $i$  in der Tabelle gemäss Sondierungssequenz  $S(k)$ . Ist  $k$  nicht vorhanden, füge  $k$  an die erste freie Position in der Sondierungsfolge ein. Andernfalls Fehlermeldung.
- **find**( $k$ ) Durchlaufe Tabelleneinträge gemäss  $S(k)$ . Wird  $k$  gefunden, gib die zu  $k$  gehörenden Daten zurück. Andernfalls Rückgabe eines leeres Elements **null**.
- **delete**( $k$ ) Suche  $k$  in der Tabelle gemäss  $S(k)$ . Wenn  $k$  gefunden, ersetze  $k$  durch den speziellen Schlüssel **removed**.

# Lineares Sondieren

$$s(k, j) = h(k) + j \Rightarrow S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1) \text{ mod } m$$

Beispiel  $m = 7$ ,  $\mathcal{K} = \{0, \dots, 500\}$ ,  $h(k) = k \text{ mod } m$ .  
Schlüssel 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
5	15	2	19		12	55

# [Analyse Lineares Sondieren (ohne Herleitung)]

- 1 Erfolglose Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C'_n \approx \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right)$$

- 2 Erfolgreiche Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C_n \approx \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right).$$



# Diskussion

Beispiel  $\alpha = 0.95$

Erfolgreiche Suche betrachtet im Durchschnitt 200 Tabelleneinträge!  
(hier ohne Herleitung).

❓ Grund für die schlechte Performance?

❗ *Primäre Häufung*: Ähnliche Hashadressen haben ähnliche Sondierungsfolgen  $\Rightarrow$  lange zusammenhängende belegte Bereiche.

# Quadratisches Sondieren

$$s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$

$$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots) \pmod m$$

Beispiel  $m = 7$ ,  $\mathcal{K} = \{0, \dots, 500\}$ ,  $h(k) = k \pmod m$ .

Schlüssel 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
19	15	2		5	12	55

# [Analyse Quadratisches Sondieren (ohne Herleitung)]

- 1 Erfolglose Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C'_n \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$$

- 2 Erfolgreiche Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C_n \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}.$$

# Diskussion

Beispiel  $\alpha = 0.95$

Erfolgreiche Suche betrachtet im Durchschnitt 22 Tabelleneinträge  
(hier ohne Herleitung)

❓ Grund für die schlechte Performance?

❗ *Sekundäre Häufung*: Synonyme  $k$  und  $k'$  (mit  $h(k) = h(k')$ )  
durchlaufen dieselbe Sondierungsfolge.

# Double Hashing

Zwei Hashfunktionen  $h(k)$  und  $h'(k)$ .  $s(k, j) = h(k) + j \cdot h'(k)$ .

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m - 1)h'(k)) \pmod m$

Beispiel:

$m = 7$ ,  $\mathcal{K} = \{0, \dots, 500\}$ ,  $h(k) = k \pmod 7$ ,  $h'(k) = 1 + k \pmod 5$ .

Schlüssel 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
5	15	2	19		12	55

# Double Hashing

- Sondierungsreihenfolge muss Permutation aller Hashadressen bilden. Also  $h'(k) \neq 0$  und  $h'(k)$  darf  $m$  nicht teilen, z.B. garantiert mit  $m$  prim.
- $h'$  sollte möglichst unabhängig von  $h$  sein (Vermeidung sekundärer Häufung).

Unabhängigkeit:

$$\mathbb{P}((h(k) = h(k')) \wedge (h'(k) = h'(k'))) = \mathbb{P}(h(k) = h(k')) \cdot \mathbb{P}(h'(k) = h'(k')).$$

Unabhängigkeit weitgehend erfüllt von  $h(k) = k \bmod m$  und  $h'(k) = 1 + k \bmod (m - 2)$  ( $m$  prim).

# [Analyse Double Hashing]

Sind  $h$  und  $h'$  unabhängig, dann:

- 1 Erfolglose Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C'_n \approx \frac{1}{1 - \alpha}$$

- 2 Erfolgreiche Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C_n \approx \frac{1}{\alpha} \ln \left( \frac{1}{1 - \alpha} \right)$$

# Gleichmässiges Hashing

Starke Annahme: Die Sondierungssequenz  $S(k)$  eines Schlüssels  $k$  ist mit gleicher Wahrscheinlichkeit eine der  $m!$  vielen Permutationssequenzen von  $\{0, 1, \dots, m - 1\}$ .

(Double Hashing kommt dem am ehesten nahe)



# Analyse gleichmässiges Hashing mit offener Addressierung

## Theorem

*Sei eine Hashtabelle mit offener Addressierung gefüllt mit Füllgrad  $\alpha = \frac{n}{m} < 1$ . Unter der Annahme vom gleichmässigen Hashing hat die nächste Operation erwartete Laufzeitkosten von  $\leq \frac{1}{1-\alpha}$ .*

# Analyse gleichmässiges Hashing mit offener Addressierung

Beweis des Theorems: Zufallsvariable  $X$ : Anzahl Sondierungen bei einer erfolglosen Suche.

$$\begin{aligned}\mathbb{P}(X \geq i) &\stackrel{*}{=} \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\stackrel{**}{\leq} \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}. \quad (1 \leq i \leq m)\end{aligned}$$

\*:  $A_j$ : Slot beim  $j$ -ten Schritt belegt.

$$\mathbb{P}(A_1 \cap \cdots \cap A_{i-1}) = \mathbb{P}(A_1) \cdot \mathbb{P}(A_2|A_1) \cdot \dots \cdot \mathbb{P}(A_{i-1}|A_1 \cap \cdots \cap A_{i-2}),$$

\*\* :  $\frac{n-1}{m-1} < \frac{n}{m}$  da<sup>25</sup>  $n < m$ .

Ausserdem  $\mathbb{P}(x \geq i) = 0$  für  $i \geq m$ . Also

$$\mathbb{E}(X) \stackrel{\text{Anhang}}{=} \sum_{i=1}^{\infty} \mathbb{P}(X \geq i) \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}.$$

---

<sup>25</sup>  $\frac{n-1}{m-1} < \frac{n}{m} \Leftrightarrow \frac{n-1}{n} < \frac{m-1}{m} \Leftrightarrow 1 - \frac{1}{n} < 1 - \frac{1}{m} \Leftrightarrow n < m$  ( $n > 0, m > 0$ )

## [Erfolgreiche Suche beim gleichmässigen offenen Hashing]

### Theorem

*Sei eine Hashtabelle mit offener Addressierung gefüllt mit Füllgrad  $\alpha = \frac{n}{m} < 1$ . Unter der Annahme vom gleichmässigen Hashing hat die erfolgreiche Suche erwartete Laufzeitkosten von  $\leq \frac{1}{\alpha} \cdot \log \frac{1}{1-\alpha}$ .*

Beweis: Cormen et al, Kap. 11.4

# Übersicht

	$\alpha = 0.50$		$\alpha = 0.90$		$\alpha = 0.95$	
	$C_n$	$C'_n$	$C_n$	$C'_n$	$C_n$	$C'_n$
(Direkte) Verkettung	1.25	0.50	1.45	0.90	1.48	0.95
Lineares Sondieren	1.50	2.50	5.50	50.50	10.50	200.50
Quadratisches Sondieren	1.44	2.19	2.85	11.40	3.52	22.05
Gleichmässiges Hashing	1.39	2.00	2.56	10.00	3.15	20.00

:  $C_n$ : Anzahl Schritte erfolgreiche Suche,  $C'_n$ : Anzahl Schritte erfolglose Suche, Belegungsgrad  $\alpha$ .

# Universelles Hashing

- $|\mathcal{K}| > m \Rightarrow$  Menge “ähnlicher Schlüssel” kann immer so gewählt sein, so dass überdurchschnittlich viele Kollisionen entstehen.
- Unmöglich, einzelne für alle Fälle “beste” Hashfunktion auszuwählen.
- Jedoch möglich<sup>26</sup>: randomisieren!

*Universelle Hashklasse*  $\mathcal{H} \subseteq \{h : \mathcal{K} \rightarrow \{0, 1, \dots, m - 1\}\}$  ist eine Familie von Hashfunktionen, so dass

$$\forall k_1 \neq k_2 \in \mathcal{K} \text{ gilt } |\{h \in \mathcal{H} \text{ mit } h(k_1) = h(k_2)\}| \leq \frac{|\mathcal{H}|}{m}.$$

---

<sup>26</sup>Ähnlich wie beim Quicksort

# Universelles Hashing

## Theorem

*Eine aus einer universellen Klasse  $\mathcal{H}$  von Hashfunktionen zufällig gewählte Funktion  $h \in \mathcal{H}$  verteilt im Erwartungswert eine beliebige Folge von Schlüsseln aus  $\mathcal{K}$  so gleichmässig wie nur möglich auf die verfügbaren Plätze.*

*Beim Hashing mit Verketteten ist die erwartete Kettenlänge für ein nicht enthaltenes Element  $\leq \alpha = n/m$ . Die erwartete Kettenlänge für ein enthaltenes Element ist  $\leq 1 + \alpha$ .*

# Universelles Hashing

Vorbemerkung zum Beweis des Theorems.

Definiere mit  $x, y \in \mathcal{K}$ ,  $h \in \mathcal{H}$ ,  $Y \subseteq \mathcal{K}$ :

$$\delta(h, x, y) = \begin{cases} 1, & \text{falls } h(x) = h(y) \\ 0, & \text{sonst,} \end{cases} \quad \text{ist } h(x) = h(y) \text{ (0 oder 1)?}$$

$$\delta(h, x, Y) = \sum_{y \in Y} \delta(x, y, h), \quad \text{für viele } y \in Y \text{ ist } h(x) = h(y)?$$

$$\delta(\mathcal{H}, x, y) = \sum_{h \in \mathcal{H}} \delta(x, y, h) \quad \text{für wie viele } h \in \mathcal{H} \text{ ist } h(x) = h(y)?.$$

$\mathcal{H}$  ist universell, wenn für alle  $x, y \in \mathcal{K}$ ,  $x \neq y$  :  $\delta(\mathcal{H}, x, y) \leq |\mathcal{H}|/m$ .

# Universelles Hashing

## Beweis des Theorems

$S \subseteq \mathcal{K}$ : bereits gespeicherte Schlüssel.  $x$  wird hinzugefügt: ( $x \notin S$ )

Erwartete Anzahl Kollisionen von  $x$  mit  $S$

$$\begin{aligned}\mathbb{E}_{\mathcal{H}}(\delta(h, x, S)) &= \sum_{h \in \mathcal{H}} \delta(h, x, S) / |\mathcal{H}| \\ &= \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \sum_{y \in S} \delta(h, x, y) = \frac{1}{|\mathcal{H}|} \sum_{y \in S} \sum_{h \in \mathcal{H}} \delta(h, x, y) \\ &= \frac{1}{|\mathcal{H}|} \sum_{y \in S} \delta(\mathcal{H}, x, y) \\ &\leq \frac{1}{|\mathcal{H}|} \sum_{y \in S} \frac{|\mathcal{H}|}{m} = \frac{|S|}{m} = \alpha.\end{aligned}$$





# Universelles Hashing

$S \subseteq \mathcal{K}$ : bereits gespeicherte Schlüssel, nun  $x \in S$ .

Erwartete Anzahl Kollisionen von  $x$  mit  $S$

$$\begin{aligned}\mathbb{E}_{\mathcal{H}}(\delta(x, S, h)) &= \sum_{h \in \mathcal{H}} \delta(x, S, h) / |\mathcal{H}| \\ &= \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \sum_{y \in S} \delta(h, x, y) = \frac{1}{|\mathcal{H}|} \sum_{y \in S} \sum_{h \in \mathcal{H}} \delta(h, x, y) \\ &= \frac{1}{|\mathcal{H}|} \left( \delta(\mathcal{H}, x, x) + \sum_{y \in S - \{x\}} \delta(\mathcal{H}, x, y) \right) \\ &\leq \frac{1}{|\mathcal{H}|} \left( |\mathcal{H}| + \sum_{y \in S - \{x\}} |\mathcal{H}|/m \right) = 1 + \frac{|S| - 1}{m} = 1 + \frac{n - 1}{m} \leq 1 + \alpha.\end{aligned}$$



# Konstruktion Universelle Hashklasse

Sei Schlüsselmenge  $\mathcal{K} = \{0, \dots, u - 1\}$  und  $p \geq u$  Primzahl.. Mit  $a \in \mathcal{K} \setminus \{0\}$ ,  $b \in \mathcal{K}$  definiere

$$h_{ab} : \mathcal{K} \rightarrow \{0, \dots, m - 1\}, h_{ab}(x) = ((ax + b) \bmod p) \bmod m.$$

Dann gilt

## Theorem

*Die Klasse  $\mathcal{H} = \{h_{ab} \mid a, b \in \mathcal{K}, a \neq 0\}$  ist eine universelle Klasse von Hashfunktionen.*

(Hier ohne Beweis. Siehe z.B. Cormen et al, Kap. 11.3.3)

# Perfektes Hashing

Ist im Vorhinein die Menge der verwendeten Schlüssel bekannt?  
Dann kann die Hashfunktion perfekt, also kollisionsfrei, gewählt werden.

Beispiel: Tabelle der Schlüsselwörter in einem Compiler.

# Beobachtung (Geburtstagsparadoxon umgekehrt)

- $h$  zufällig gewählt aus universeller Hashfamilie  $\mathcal{H}$ .
- $n$  Schlüssel  $S \subset \mathcal{K}$
- Zufallsvariable  $X$  : Anzahl Kollisionen der  $n$  Schlüssel aus  $S$

⇒

$$\begin{aligned}\mathbb{E}(X) &= \mathbb{E} \left( \sum_{i \neq j} \mathbb{1}(h(k_i) = h(k_j)) \right) = \sum_{i \neq j} \mathbb{E} (\mathbb{1}(h(k_i) = h(k_j))) \\ &\stackrel{*}{=} \binom{n}{2} \frac{1}{m} \leq \frac{n^2}{2m}\end{aligned}$$

\* # Ungeordnete Paare  $\sum_{i \neq j} 1 = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} (n-1-i) = n(n-1) - n(n-1)/2 = n(n-1)/2$

# Perfektes Hashing mit $\Theta(n^2)$ Speicherbedarf

Wenn  $m = n^2 \Rightarrow \mathbb{E}(X) \leq \frac{1}{2}$ .

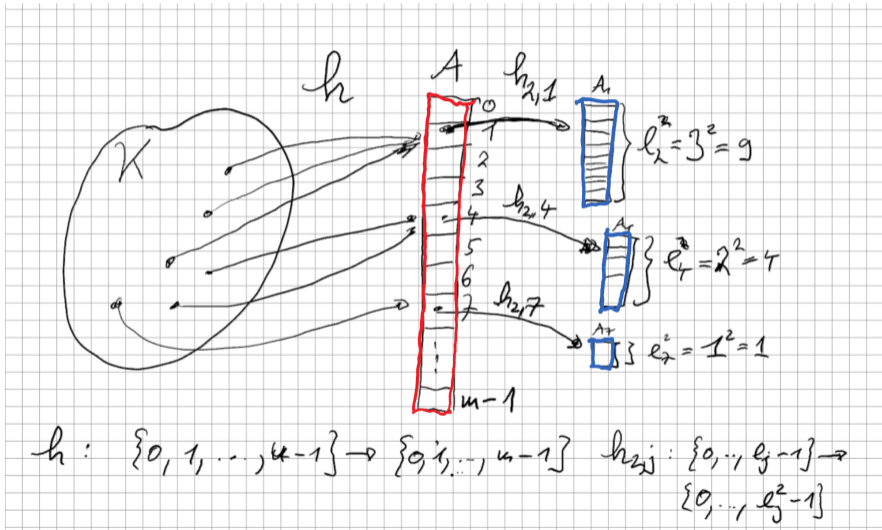
Markov-Ungleichung<sup>27</sup>  $\mathbb{P}(X \geq 1) \leq \frac{\mathbb{E}(X)}{1} \leq \frac{1}{2}$

Also

$$\mathbb{E}(X < 1) = \mathbb{E}(\text{keine Kollision}) \geq \frac{1}{2}.$$

Folgerung: in erwartet  $2 \cdot n$  Schritten kann man zu  $n$  Schlüsseln eine kollisionsfreie Hashtabelle der Grösse  $m = n^2$  durch zufällige Wahl aus einer universellen Hashfamilie konstruieren.

# Perfect Hashing Idea



# Perfektes Hashing mit $\Theta(n)$ Speicherbedarf

## 2-Stufiges Verfahren

- 1 Wähle  $m = n$  und  $h : \{0, 1, \dots, u - 1\} \rightarrow \{0, 1, \dots, m - 1\}$  aus einer universellen Hashfamilie. Füge alle  $n$  Schlüssel in die Hashtabelle mit Verketteten ein. Sei  $l_i$  die Länge der Kette am Index  $i$ .

Wenn  $\sum_{i=0}^{m-1} l_i^2 > 4n$ , dann wiederhole diesen Schritt 1.

- 2 Für jeden Index  $i = 1, \dots, m - 1$  mit  $l_i > 0$  erzeuge so lange Hashtabellen für die enthaltenen  $l_i$  Schlüssel der Länge  $l_i^2$  mit universellem Hashing (Hashfunktion  $h_{2,i}$ ), bis keine Kollisionen auftreten.

Speicherbedarf  $\Theta(n)$ .

# Erwartete Laufzeiten

- Für Schritt 1: Hashtabelle der Grösse  $m = n$ .  
Wir zeigen auf der nächsten Seite, dass  $\mathbb{E} \left( \sum_{j=0}^{m-1} l_j^2 \right) \leq 2n$ .  
Dann folgt (Markov):  $\mathbb{P} \left( \sum_{j=0}^{m-1} l_j^2 \geq 4n \right) \leq \frac{2n}{4n} = \frac{1}{2}$ .  
 $\Rightarrow$  Erwartete zwei Wiederholungen vom Schritt 1.
- Für Schritt 2:  $\sum l_i^2 \leq 4n$ . Für jedes  $i$  erwartet zwei Versuche mit Laufzeit  $l_i^2$ . Insgesamt  $\mathcal{O}(n)$   
 $\Rightarrow$  Die perfekte Hashtabelle kann in erwarteter  $\mathcal{O}(n)$  Schritten erstellt werden.



# Erwarteter Speicherverbrauch Hashtabellen 2.Stufe

$$\begin{aligned}\mathbb{E} \left( \sum_{j=0}^{m-1} l_j^2 \right) &= \mathbb{E} \left( \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} \sum_{i'=0}^{n-1} \mathbb{1}(h(k_i) = h(k_{i'}) = j) \right) \\ &= \mathbb{E} \left( \sum_{i=0}^{n-1} \sum_{i'=0}^{n-1} \mathbb{1}(h(k_i) = h(k_{i'})) \right) \\ &= \mathbb{E} \left( \sum_{i=i'} \mathbb{1}(h(k_i) = h(k_{i'})) + 2 \cdot \sum_{i \neq i'} \mathbb{1}(h(k_i) = h(k_{i'})) \right) \\ &= n + 2 \cdot \sum_{i \neq i'} \mathbb{E} (\mathbb{1}(h(k_i) = h(k_{i'}))) \\ &= n + 2 \binom{n}{2} \frac{1}{m} \stackrel{m=n}{=} 2n - 1 \leq 2n.\end{aligned}$$

# 14.9 Anhang

Mathematische Formeln

# [Geburtstagsparadoxon]

Annahme:  $m$  Urnen,  $n$  Kugeln (oBdA  $n \leq m$ ).

$n$  Kugeln werden gleichverteilt in Urnen gelegt.



Wie gross ist die Kollisionswahrscheinlichkeit?

**Geburtstagsparadoxon:** Bei wie vielen Personen ( $n$ ) ist die Wahrscheinlichkeit, dass zwei am selben Tag ( $m = 365$ ) Geburtstag haben grösser als 50%?

# [Geburtstagsparadoxon]

$$\mathbb{P}(\text{keine Kollision}) = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \frac{m!}{(m-n)! \cdot m^n}.$$

Sei  $a \ll m$ . Mit  $e^x = 1 + x + \frac{x^2}{2!} + \dots$  approximiere  $1 - \frac{a}{m} \approx e^{-\frac{a}{m}}$ .

Damit:

$$1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \dots \cdot \left(1 - \frac{n-1}{m}\right) \approx e^{-\frac{1+\dots+n-1}{m}} = e^{-\frac{n(n-1)}{2m}}.$$

Es ergibt sich

$$\mathbb{P}(\text{Kollision}) = 1 - e^{-\frac{n(n-1)}{2m}}.$$

Auflösung zum Geburtstagsparadoxon: Bei 23 Leuten ist die Wahrscheinlichkeit für Geburtstagskollision 50.7%. Zahl

stammt von der leicht besseren Approximation via Stirling Formel.  $n! \approx \sqrt{2\pi n} \cdot n^n \cdot e^{-n}$

# [Erwartungswertformel]

$X \geq 0$  diskrete Zufallsvariable mit  $\mathbb{E}(X) < \infty$

$$\mathbb{E}(X) \stackrel{(def)}{=} \sum_{x=0}^{\infty} x \mathbb{P}(X = x)$$

$$\stackrel{\text{Aufzählen}}{=} \sum_{x=1}^{\infty} \sum_{y=x}^{\infty} \mathbb{P}(X = y)$$

$$= \sum_{x=0}^{\infty} \mathbb{P}(X \geq x)$$

# [Markov Ungleichung]

diskrete Version

$$\begin{aligned}\mathbb{E}(X) &= \sum_{x=-\infty}^{\infty} x\mathbb{P}(X = x) \\ &\geq \sum_{x=a}^{\infty} x\mathbb{P}(X = x) \\ &\geq a \sum_{x=a}^{\infty} \mathbb{P}(X = x) \\ &= a \cdot \mathbb{P}(X \geq a)\end{aligned}$$

⇒

$$\mathbb{P}(X \geq a) \leq \frac{\mathbb{E}(X)}{a}$$

# 15. C++ vertieft (III): Funktoren und Lambda

# Was lernen wir heute?

- Funktoren: Objekte mit überladenem Funktionsoperator ().
- Closures
- Lambda-Ausdrücke: Syntaktischer Zucker
- Captures
- Variablen vom Funktionstyp



# Funktoren: Motivierung

Ein simpler Ausgabefilter

```
template <typename T, typename Function>
void filter(const T& collection, Function f){
    for (const auto& x: collection)
        if (f(x)) std::cout << x << " ";
    std::cout << "\n";
}
```

( `filter` funktioniert wenn das erste Argument einen Iterator anbietet und das zweite auf Elemente angewendet werden kann und das Resultat zu `bool` konvertierbar ist. )

# Funktoren: Motivierung

```
template <typename T, typename Function>  
void filter(const T& collection, Function f);
```

```
template <typename T>  
bool even(T x){  
    return x % 2 == 0;  
}
```

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};  
filter(a,even<int>); // output: 2,4,6,16
```

# Funktor: Objekt mit überladenem Operator ()

```
class GreaterThan{
    int value; // state
public:
    GreaterThan(int x):value{x}{}

    bool operator() (int par) const {
        return par > value;
    }
};
```

Ein Funktor ist ein aufrufbares Objekt. Kann verstanden werden als Funktion mit Zustand.

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;
filter(a, GreaterThan(value)); // 9,11,16,19
```

# Funktor: Objekt mit überladenem Operator ()

```
template <typename T>
class GreaterThan{
    T value;
public:
    GreaterThan(T x):value{x}{}

    bool operator() (T par) const{
        return par > value;
    }
};
```

(geht natürlich auch mit  
Template)

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;
filter(a, GreaterThan<int>(value)); // 9,11,16,19
```

# Dasselbe mit Lambda-Expression

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
```

```
int value=8;
```

```
filter(a, [value](int x) {return x > value;} );
```

# Summe aller Elemente - klassisch

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};  
int sum = 0;  
for (auto x: a)  
    sum += x;  
std::cout << sum << std::endl; // 83
```

# Summe aller Elemente - mit Funktor

```
template <typename T>
struct Sum{
    T value = 0;

    void operator() (T par){ value += par; }
};

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
Sum<int> sum;
// for_each copies sum: we need to copy the result back
sum = std::for_each(a.begin(), a.end(), sum);
std::cout << sum.value << std::endl; // 83
```

# Summe aller Elemente - mit Referenzen<sup>28</sup>

```
template <typename T>
struct SumR{
    T& value;
    SumR (T& v):value{v} {}

    void operator() (T par){ value += par; }
};

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int s=0;
SumR<int> sum{s};
// cannot (and do not need to) assign to sum here
std::for_each(a.begin(), a.end(), sum);
std::cout << s << std::endl; // 83
```

<sup>28</sup>Geht natürlich sehr ähnlich auch mit Zeigern



# Summe aller Elemente - mit $\Lambda$

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
```

```
int s=0;
```

```
std::for_each(a.begin(), a.end(), [&s] (int x) {s += x;} );
```

```
std::cout << s << std::endl;
```

# Sortieren, mal anders

```
// pre: i >= 0
// post: returns sum of digits of i
int q(int i){
    int res =0;
    for(;i>0;i/=10)
        res += i % 10;
    return res;
}
```

```
std::vector<int> v {10,12,9,7,28,22,14};
std::sort (v.begin(), v.end(),
    [] (int i, int j) { return q(i) < q(j);}
);
```

Jetzt  $v = 10, 12, 22, 14, 7, 9, 28$  (sortiert nach Quersumme)

# Lambda-Expressions im Detail

`[value]` `(int x)` `->bool` `{return x > value;}`

Capture      Parameter      Rückgabe-  
typ      Anweisung

# Closure

```
[value] (int x) ->bool {return x > value;}
```

- Lambda-Expressions evaluieren zu einem temporären Objekt – einer closure
- Die closure erhält den Ausführungskontext der Funktion, also die captured Objekte.
- Lambda-Expressions können als Funktoren implementiert werden.

# Simple Lambda-Expression

```
[] () -> void {std::cout << "Hello World";}
```

Aufruf:

```
[] () -> void {std::cout << "Hello World";}();
```

Zuweisung:

```
auto f = [] () -> void {std::cout << "Hello World";};
```

# Minimale Lambda-Expression

`[] {}`

- Rückgabetypp kann inferiert werden, wenn kein oder nur ein `return`:<sup>29</sup>

```
[] () {std::cout << "Hello World";}
```

- Keine Parameter und kein expliziter Rückgabetypp  $\Rightarrow$  `()` kann weggelassen werden

```
[] {std::cout << "Hello World";}
```

- `[...]` kann nie weggelassen werden.

---

<sup>29</sup>Seit C++14 auch mehrere `returns`, sofern derselbe Rückgabetypp deduziert wird

# Beispiele

```
[] (int x, int y) {std::cout << x * y;} (4,5);
```

Output: 20

# Beispiele

```
int k = 8;  
auto f = [](int& v) {v += v;};  
f(k);  
std::cout << k;
```

Output: 16



# Beispiele

```
int k = 8;  
auto f = [] (int v) {v += v;};  
f(k);  
std::cout << k;
```

Output: 8

# Capture – Lambdas

Für Lambda-Expressions bestimmt die capture-Liste über den zugreifbaren Teil des Kontextes

Syntax:

- `[x]`: Zugriff auf kopierten Wert von x (nur lesend)
- `[&x]`: Zugriff zur Referenz von x
- `[&x, y]`: Zugriff zur Referenz von x und zum kopierten Wert von y
- `[&]`: Default-Referenz-Zugriff auf alle Objekte im Kontext der Lambda-Expression
- `[=]`: Default-Werte-Zugriff auf alle Objekte im Kontext der Lambda-Expression

# Capture – Lambdas

```
int elements=0;
int sum=0;
std::for_each(v.begin(), v.end(),
    [&] (int k) {sum += k; elements++;} // capture all by reference
)
```

# Capture – Lambdas

```
template <typename T>
void sequence(vector<int> & v, T done){
    int i=0;
    while (!done()) v.push_back(i++);
}
```

```
vector<int> s;
sequence(s, [&] {return s.size() >= 5;} )
```

jetzt v = 0 1 2 3 4

Die capture liste bezieht sich auf den Kontext der Lambda Expression

# Capture – Lambdas

Wann wird der Wert gelesen?

```
int v = 42;  
auto func = [=] {std::cout << v << "\n"};  
v = 7;  
func();
```

Ausgabe: 42

Werte werden bei der Definition der (temporären) Lambda-Expression zugewiesen.

# Capture – Lambdas

(Warum) funktioniert das?

```
class Limited{
    int limit = 10;
public:
    // count entries smaller than limit
    int count(const std::vector<int>& a){
        int c = 0;
        std::for_each(a.begin(), a.end(),
            [=,&c] (int x) {if (x < limit) c++;}
        );
        return c;
    }
};
```

Der `this` pointer wird per default implizit kopiert

# Capture – Lambdas

```
struct mutant{  
    int i = 0;  
    void do(){ [=] {i=42;}();}  
};
```

```
mutant m;  
m.do();  
std::cout << m.i;
```

Ausgabe: 42

Der `this pointer` wird per default implizit kopiert

# Lambda Ausdrücke sind Funktoren

```
[x, &y] () {y = x;}
```

kann implementiert werden als

```
unnamed {x,y};
```

mit

```
class unnamed {  
    int x; int& y;  
    unnamed (int x_, int& y_) : x (x_), y (y_) {}  
    void operator () () {y = x;}  
};
```



# Lambda Ausdrücke sind Funktoren

```
[=] () {return x + y;}
```

kann implementiert werden als

```
unnamed {x,y};
```

mit

```
class unnamed {  
    int x; int y;  
    unnamed (int x_, int y_) : x (x_), y (y_) {}  
    int operator () () const {return x + y;}  
};
```

# Polymorphic Function Wrapper `std::function`

```
#include <functional>

int k= 8;
std::function<int(int)> f;
f = [k](int i){ return i+k; };
std::cout << f(8); // 16
```

Kann verwendet werden, um Lambda-Expressions zu speichern.

Andere Beispiele

```
std::function<int(int, int)>;
std::function<void(double)> ...
```

<http://en.cppreference.com/w/cpp/utility/functional/function>

# Beispiel

```
template <typename T>
auto toFunction(std::vector<T> v){
    return [v] (T x) -> double {
        int index = (int)(x+0.5);
        if (index < 0) index = 0;
        if (index >= v.size()) index = v.size()-1;
        return v[index];
    };
}
```

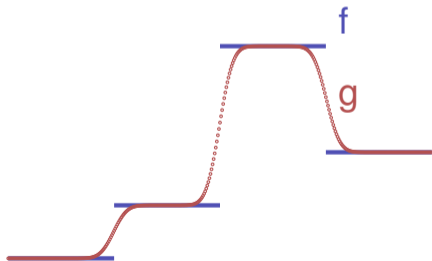
# Beispiel

```
auto Gaussian(double mu, double sigma){
    return [mu,sigma](double x) {
        const double a = ( x - mu ) / sigma;
        return std::exp( -0.5 * a * a );
    };
}
```

```
template <typename F, typename Kernel>
auto smooth(F f, Kernel kernel){
    return [kernel,f] (auto x) {
        // compute convolution ...
        // and return result
    };
}
```

# Beispiel

```
std::vector<double> v {1,2,5,3};  
auto f = toFunction(v);  
auto k = Gaussian(0,0.1);  
auto g = smooth(f,k);
```



# Zusammenfassung

- Funktoren erlauben die funktionale Programmierung mit C++. Lambdas sind syntaktischer Zucker, der das deutlich vereinfacht
- Mit Funktoren/Lambdas sind klassische Muster aus der funktionalen Programmierung (z.B. map / filter / reduce) auch in C++ möglich.
- In Kombination mit Templates und Typinferenz (`auto`) können sehr mächtige Funktionen in Variablen gespeichert werden, Funktionen können sogar Funktionen zurückgeben (sog. Funktionen höherer Ordnung).

# 16. Natürliche Suchbäume

[Ottman/Widmayer, Kap. 5.1, Cormen et al, Kap. 12.1 - 12.3]

# Wörterbuchimplementationen

Hashing: Implementierung von Wörterbüchern mit erwartet sehr schnellen Zugriffszeiten.

Nachteile von Hashing: im schlechtesten Fall lineare Zugriffszeit.

Manche Operationen gar nicht unterstützt:

- Aufzählen von Schlüssel in aufsteigender Anordnung
- Nächst kleinerer Schlüssel zu gegebenem Schlüssel
- Schlüssel  $k$  in vorgegebenem Intervall  $k \in [l, r]$



# Bäume

Bäume sind

- Verallgemeinerte Listen: Knoten können mehrere Nachfolger haben
- Spezielle Graphen: Graphen bestehen aus Knoten und Kanten. Ein Baum ist ein zusammenhängender, gerichteter, azyklischer Graph.

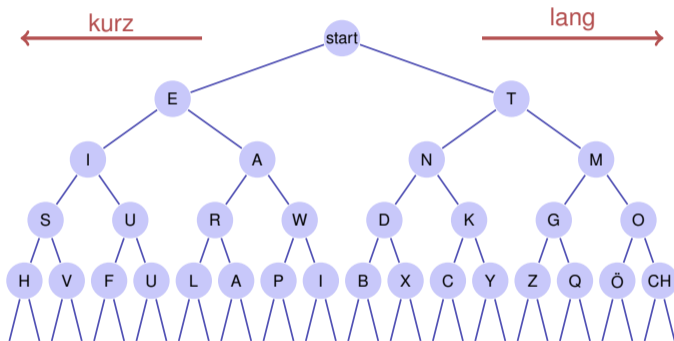
# Bäume

## Verwendung

- Entscheidungsbäume: Hierarchische Darstellung von Entscheidungsregeln
- Syntaxbäume: Parsen und Traversieren von Ausdrücken, z.B. in einem Compiler
- Codebäume: Darstellung eines Codes, z.B. Morsealphabet, Huffman Code
- Suchbäume: ermöglichen effizientes Suchen eines Elementes



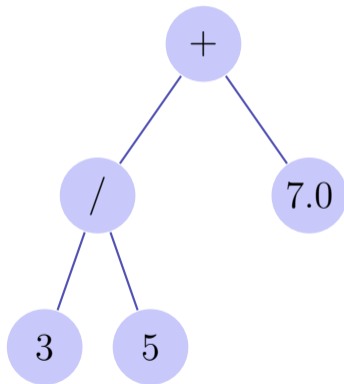
# Beispiele



Morsealphabet

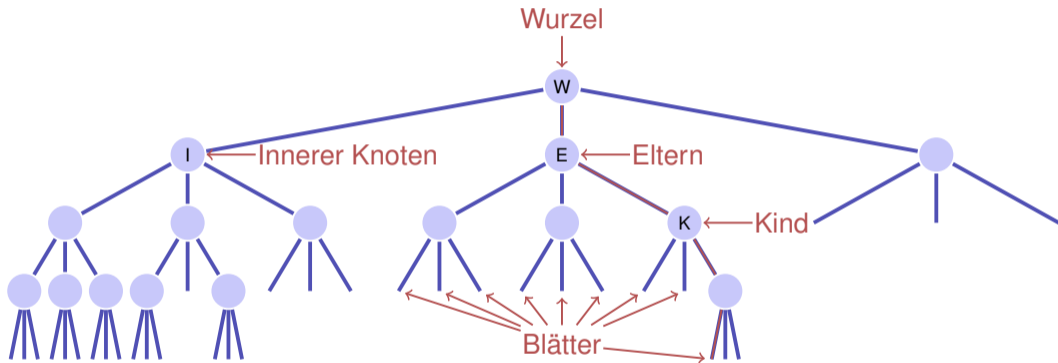
# Beispiele

$3/5 + 7.0$



Ausdrucksbaum

# Nomenklatur



- Ordnung des Baumes: Maximale Anzahl Kindknoten, hier: 3
- Höhe des Baumes: maximale Pfadlänge Wurzel – Blatt (hier: 4)

# Binäre Bäume

Ein binärer Baum ist

- entweder ein Blatt, d.h. ein leerer Baum,
- oder ein innerer Knoten mit zwei Bäumen  $T_l$  (linker Teilbaum) und  $T_r$  (rechter Teilbaum) als linken und rechten Nachfolger.

In jedem inneren Knoten  $v$  wird gespeichert

- ein Schlüssel  $v.key$  und
- zwei Zeiger  $v.left$  und  $v.right$  auf die Wurzeln der linken und rechten Teilbäume.

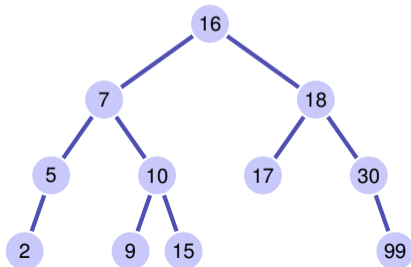
key	
left	right

Ein Blatt wird durch den **null**-Zeiger repräsentiert

# Binärer Suchbaum

Ein binärer Suchbaum ist ein binärer Baum, der die *Suchbaumeigenschaft* erfüllt:

- Jeder Knoten  $v$  speichert einen Schlüssel
- Schlüssel im linken Teilbaum  $v.left$  kleiner als  $v.key$
- Schlüssel im rechten Teilbaum  $v.right$  grösser als  $v.key$



# Suchen

**Input:** Binärer Suchbaum mit Wurzel  $r$ ,  
Schlüssel  $k$

**Output:** Knoten  $v$  mit  $v.key = k$  oder **null**

$v \leftarrow r$

**while**  $v \neq \text{null}$  **do**

**if**  $k = v.key$  **then**

        | **return**  $v$

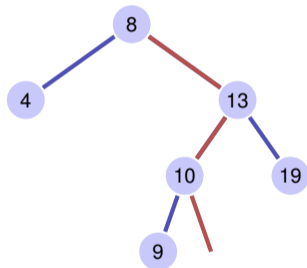
**else if**  $k < v.key$  **then**

        |  $v \leftarrow v.left$

**else**

        |  $v \leftarrow v.right$

**return null**



Search (12)  $\rightarrow$  **null**



# Höhe eines Baumes

Die Höhe  $h(T)$  eines binären Baumes  $T$  mit Wurzel  $r$  ist gegeben als

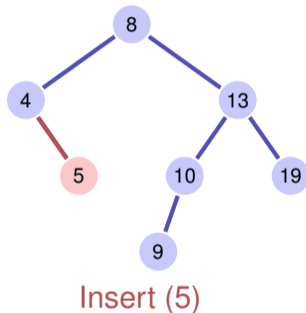
$$h(r) = \begin{cases} 0 & \text{falls } r = \mathbf{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{sonst.} \end{cases}$$

Die Laufzeit der Suche ist somit im schlechtesten Fall  $\mathcal{O}(h(T))$

# Einfügen eines Schlüssels

Einfügen des Schlüssels  $k$

- Suche nach  $k$ .
- Wenn erfolgreich:  
Fehlerausgabe
- Wenn erfolglos: Einfügen des  
Schlüssels am erreichten Blatt.

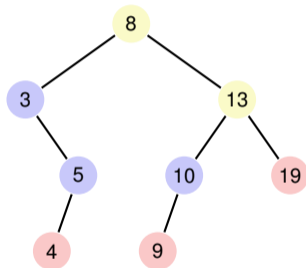


# Knoten entfernen

Drei Fälle möglich

- Knoten hat keine Kinder
- Knoten hat ein Kind
- Knoten hat zwei Kinder

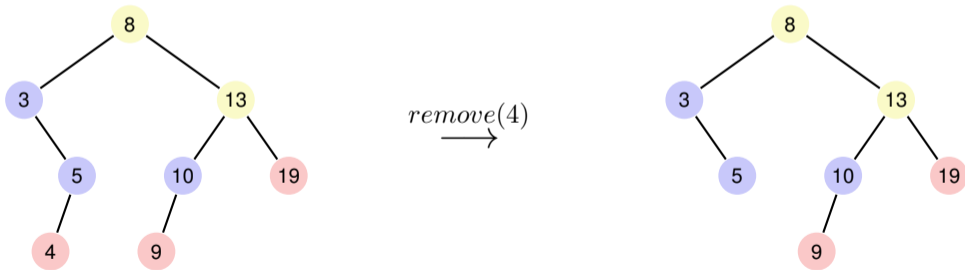
[Blätter zählen hier nicht]



# Knoten entfernen

Knoten hat keine Kinder

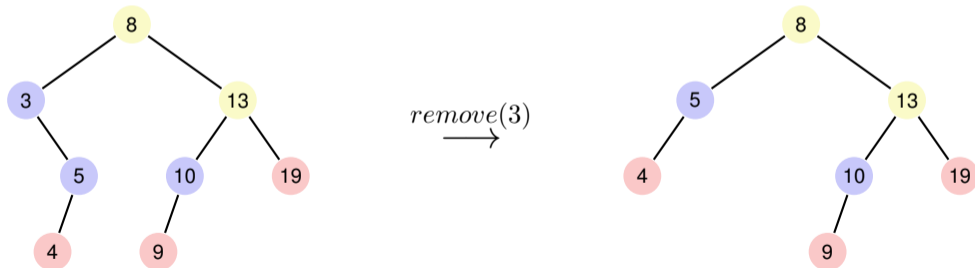
Einfacher Fall: Knoten durch Blatt ersetzen.



# Knoten entfernen

Knoten hat ein Kind

Auch einfach: Knoten durch das einzige Kind ersetzen.



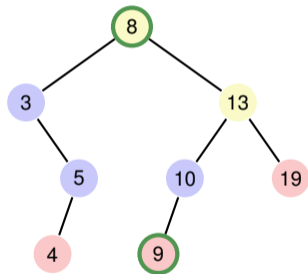
# Knoten entfernen

Knoten  $v$  hat zwei Kinder

Beobachtung: Der kleinste Schlüssel im rechten Teilbaum  $v.\text{right}$  (der *symmetrische Nachfolger* von  $v$ )

- ist kleiner als alle Schlüssel in  $v.\text{right}$
- ist grösser als alle Schlüssel in  $v.\text{left}$
- und hat kein linkes Kind.

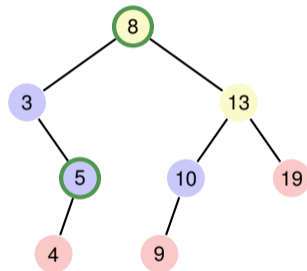
Lösung: ersetze  $v$  durch seinen symmetrischen Nachfolger



# Aus Symmetriegründen...

Knoten  $v$  hat zwei Kinder

Auch möglich: ersetze  $v$  durch seinen symmetrischen Vorgänger



# Algorithmus SymmetricSuccessor( $v$ )

**Input:** Knoten  $v$  eines binären Suchbaumes

**Output:** Symmetrischer Nachfolger von  $v$

$w \leftarrow v.\text{right}$

$x \leftarrow w.\text{left}$

**while**  $x \neq \text{null}$  **do**

$w \leftarrow x$   
     $x \leftarrow x.\text{left}$

**return**  $w$



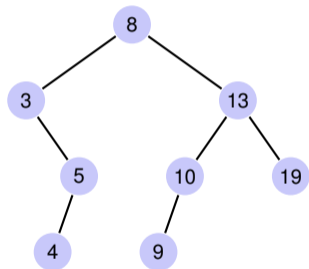
# Analyse

Löschen eines Elementes  $v$  aus einem Baum  $T$  benötigt  $\mathcal{O}(h(T))$   
Elementarschritte:

- Suchen von  $v$  hat Kosten  $\mathcal{O}(h(T))$
- Hat  $v$  maximal ein Kind ungleich **null**, dann benötigt das Entfernen  $\mathcal{O}(1)$
- Das Suchen des symmetrischen Nachfolgers  $n$  benötigt  $\mathcal{O}(h(T))$   
Schritte. Entfernen und Einfügen von  $n$  hat Kosten  $\mathcal{O}(1)$

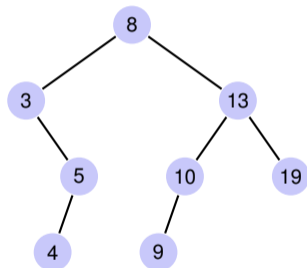
# Traversierungsarten

- Hauptreihenfolge (preorder):  $v$ , dann  $T_{\text{left}}(v)$ , dann  $T_{\text{right}}(v)$ .  
8, 3, 5, 4, 13, 10, 9, 19
- Nebenreihenfolge (postorder):  $T_{\text{left}}(v)$ , dann  $T_{\text{right}}(v)$ , dann  $v$ .  
4, 5, 3, 9, 10, 19, 13, 8
- Symmetrische Reihenfolge (inorder):  
 $T_{\text{left}}(v)$ , dann  $v$ , dann  $T_{\text{right}}(v)$ .  
3, 4, 5, 8, 9, 10, 13, 19

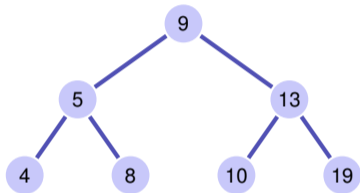


# Weitere unterstützte Operationen

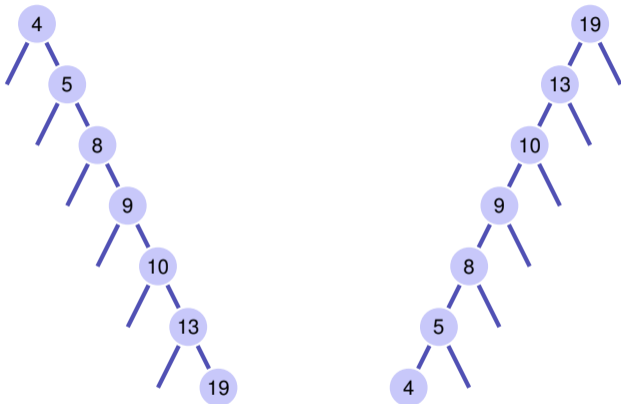
- $\text{Min}(T)$ : Auslesen des Minimums in  $\mathcal{O}(h)$
- $\text{ExtractMin}(T)$ : Auslesen und Entfernen des Minimums in  $\mathcal{O}(h)$
- $\text{List}(T)$ : Ausgeben einer sortierten Liste der Elemente von  $T$
- $\text{Join}(T_1, T_2)$ : Zusammenfügen zweier Bäume mit  $\max(T_1) < \min(T_2)$  in  $\mathcal{O}(n)$ .



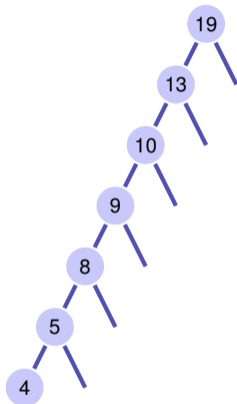
# Degenerierte Suchbäume



Insert 9,5,13,4,8,10,19  
bestmöglich  
balanciert



Insert 4,5,8,9,10,13,19  
Lineare Liste



Insert 19,13,10,9,8,5,4  
Lineare Liste

# Probabilistisch

Ein Suchbaum, welcher aus einer zufälligen Sequenz von Zahlen erstellt wird hat erwartete Pfadlänge von  $\mathcal{O}(\log n)$ .

Achtung: das gilt nur für Einfügeoperation. Wird der Baum zufällig durch Einfügen und Entfernen gebildet, ist die erwartete Pfadlänge  $\mathcal{O}(\sqrt{n})$ .

*Balancierte* Bäume stellen beim Einfügen und Entfernen (z.B. durch *Rotationen*) sicher, dass der Baum balanciert bleibt und liefern eine  $\mathcal{O}(\log n)$  Worst-Case-Garantie.

# 17. AVL Bäume

Balancierte Bäume [Ottman/Widmayer, Kap. 5.2-5.2.1, Cormen et al, Kap. Problem 13-3]

# Ziel

Suchen, Einfügen und Entfernen eines Schlüssels in Baum mit  $n$  Schlüsseln, welche in zufälliger Reihenfolge eingefügt wurden im Mittel in  $\mathcal{O}(\log_2 n)$  Schritten.

Schlechtester Fall jedoch:  $\Theta(n)$  (degenerierter Baum).

**Ziel:** Verhinderung der Degenerierung. Künstliches, bei jeder Update-Operation erfolgtes Balancieren eines Baumes

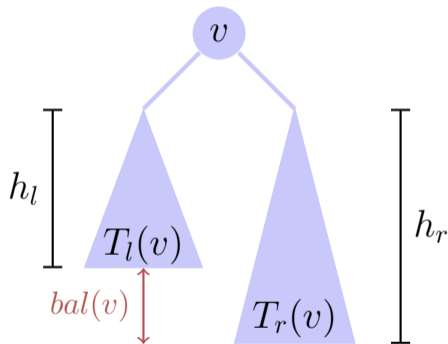
Balancierung: garantiere, dass ein Baum mit  $n$  Knoten stets eine Höhe von  $\mathcal{O}(\log n)$  hat.

**Adelson-Venskii und Landis (1962): AVL-Bäume**

# Balance eines Knotens

Die *Balance* eines Knotens  $v$  ist definiert als die Höhendifferenz seiner beiden Teilbäume  $T_l(v)$  und  $T_r(v)$

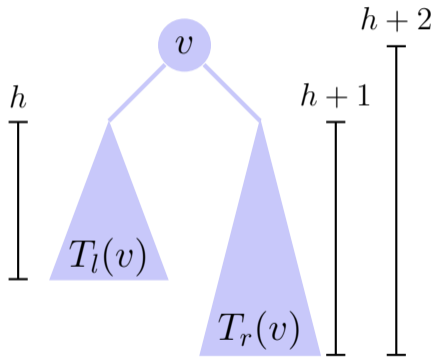
$$\text{bal}(v) := h(T_r(v)) - h(T_l(v))$$



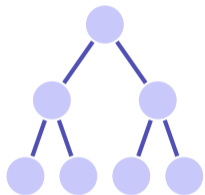


# AVL Bedingung

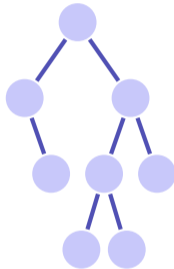
*AVL Bedingung: für jeden Knoten  $v$  eines Baumes gilt  $\text{bal}(v) \in \{-1, 0, 1\}$*



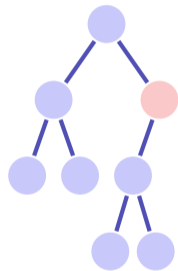
# (Gegen-)Beispiele



AVL Baum der Höhe  
2



AVL Baum der Höhe  
3

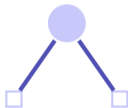


Kein AVL Baum

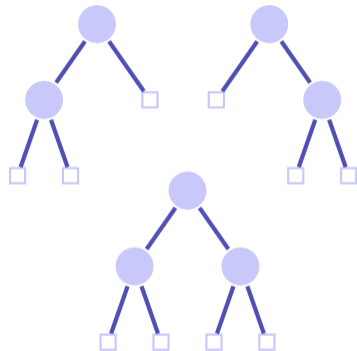
# Anzahl Blätter

- 1. Beobachtung: Ein Suchbaum mit  $n$  Schlüsseln hat genau  $n + 1$  Blätter. Einfaches Induktionsargument.
  - Der Suchbaum mit  $n = 0$  Schlüsseln hat  $m = 1$  Blätter
  - Wird ein Schlüssel (Knoten) hinzugefügt ( $n \rightarrow n + 1$ ), so ersetzt er ein Blatt und fügt zwei Blätter hinzu ( $m \rightarrow m - 1 + 2 = m + 1$ ).
- 2. Beobachtung: untere Grenze für Anzahl Blätter eines Suchbaums zu gegebener Höhe erlaubt Abschätzung der maximalen Höhe eines Suchbaums zu gegebener Anzahl Schlüssel.

# Untere Grenze Blätter



AVL Baum der Höhe 1 hat  
 $N(1) := 2$  Blätter



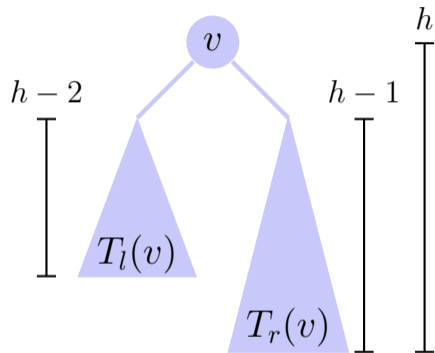
AVL Baum der Höhe 2 hat  
mindestens  $N(2) := 3$   
Blätter

# Untere Grenze Blätter für $h > 2$

- Höhe eines Teilbaums  $\geq h - 1$ .
- Höhe des anderen Teilbaums  $\geq h - 2$ .

Minimale Anzahl Blätter  $N(h)$  ist

$$N(h) = N(h - 1) + N(h - 2)$$



Insgesamt gilt  $N(h) = F_{h+2}$  mit **Fibonacci-Zahlen**  $F_0 := 0$ ,  $F_1 := 1$ ,  
 $F_n := F_{n-1} + F_{n-2}$  für  $n > 1$ .

# Fibonacci Zahlen, geschlossene Form

Es gilt<sup>30</sup>

$$F_i = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)$$

mit den Wurzeln  $\phi, \hat{\phi}$  der Gleichung vom goldenen Schnitt  
 $x^2 - x - 1 = 0$ :

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.618$$

$$\hat{\phi} = \frac{1 - \sqrt{5}}{2} \approx -0.618$$

---

<sup>30</sup>Herleitung mit Erzeugendenfunktionen (Potenzreihen) im Anhang

# Fibonacci Zahlen, Induktiver Beweis

$$F_i \stackrel{!}{=} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i) \quad [*] \quad \left( \phi = \frac{1+\sqrt{5}}{2}, \hat{\phi} = \frac{1-\sqrt{5}}{2} \right).$$

**1** Klar für  $i = 0, i = 1$ .

**2** Sei  $i > 2$  und Behauptung  $[*]$  wahr für alle  $F_j, j < i$ .

$$\begin{aligned} F_i &\stackrel{\text{def}}{=} F_{i-1} + F_{i-2} \stackrel{[*]}{=} \frac{1}{\sqrt{5}}(\phi^{i-1} - \hat{\phi}^{i-1}) + \frac{1}{\sqrt{5}}(\phi^{i-2} - \hat{\phi}^{i-2}) \\ &= \frac{1}{\sqrt{5}}(\phi^{i-1} + \phi^{i-2}) - \frac{1}{\sqrt{5}}(\hat{\phi}^{i-1} + \hat{\phi}^{i-2}) = \frac{1}{\sqrt{5}}\phi^{i-2}(\phi + 1) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi} + 1) \end{aligned}$$

$(\phi, \hat{\phi}$  erfüllen  $x + 1 = x^2$ )

$$= \frac{1}{\sqrt{5}}\phi^{i-2}(\phi^2) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi}^2) = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i).$$

# Baumhöhe

Da  $|\hat{\phi}| < 1$ , gilt insgesamt

$$N(h) \in \Theta \left( \left( \frac{1 + \sqrt{5}}{2} \right)^h \right) \subseteq \Omega(1.618^h)$$

und somit

$$\begin{aligned} N(h) &\geq c \cdot 1.618^h \\ \Rightarrow h &\leq 1.44 \log_2 n + c'. \end{aligned}$$

Ein AVL Baum ist asymptotisch nicht mehr als 44% höher als ein perfekt balancierter Baum.<sup>31</sup>

<sup>31</sup>Ein perfekt balancierter Baum hat Höhe  $\lceil \log_2 n + 1 \rceil$



# Einfügen

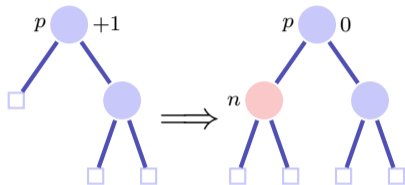
## Balancieren

- Speichern der Balance für jeden Knoten
- Baum rebalancieren bei jeder Update-Operation

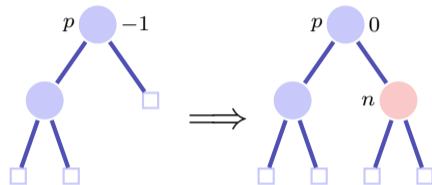
Neuer Knoten  $n$  wird eingefügt:

- Zuerst einfügen wie bei Suchbaum.
- Prüfe die Balance-Bedingung für alle Knoten aufsteigend von  $n$  zur Wurzel.

# Balance am Einfügeort



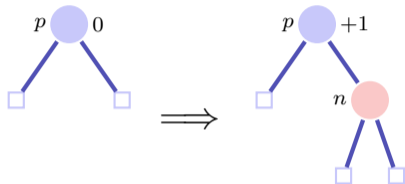
Fall 1:  $\text{bal}(p) = +1$



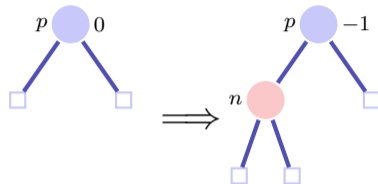
Fall 2:  $\text{bal}(p) = -1$

Fertig in beiden Fällen, denn der Teilbaum ist nicht gewachsen.

# Balance am Einfügeort



Fall 3.1:  $\text{bal}(p) = 0$  rechts



Fall 3.2:  $\text{bal}(p) = 0$ , links

In beiden Fällen noch nicht fertig. Aufruf von `upin(p)`.

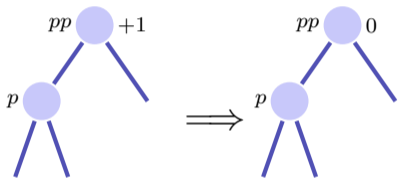
# upin(p) - Invariante

Beim Aufruf von `upin(p)` gilt, dass

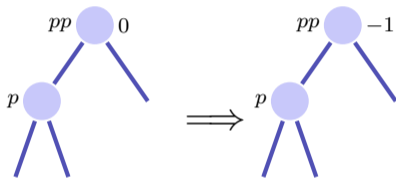
- der Teilbaum ab  $p$  gewachsen ist und
- $\text{bal}(p) \in \{-1, +1\}$

# upin(p)

Annahme:  $p$  ist linker Sohn von  $pp$ <sup>32</sup>



Fall 1:  $\text{bal}(pp) = +1$ , fertig.



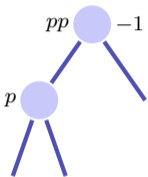
Fall 2:  $\text{bal}(pp) = 0$ , **upin(pp)**

In beiden Fällen gilt nach der Operation die AVL-Bedingung für den Teilbaum ab  $pp$

<sup>32</sup>Ist  $p$  rechter Sohn: symmetrische Fälle unter Vertauschung von +1 und -1

# upin(p)

Annahme:  $p$  ist linker Sohn von  $pp$



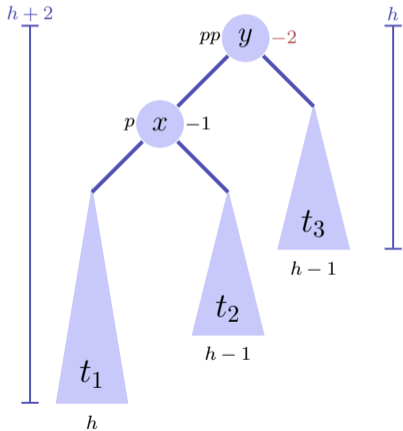
Fall 3:  $\text{bal}(pp) = -1$ ,

Dieser Fall ist problematisch: das Hinzufügen von  $n$  im Teilbaum ab  $pp$  hat die AVL-Bedingung verletzt. Rebalancieren!

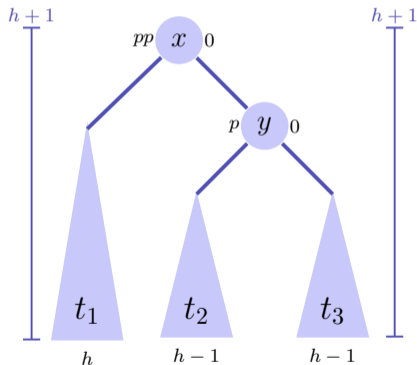
Zwei Fälle  $\text{bal}(p) = -1$ ,  $\text{bal}(p) = +1$

# Rotationen

Fall 1.1  $\text{bal}(p) = -1$ .<sup>33</sup>



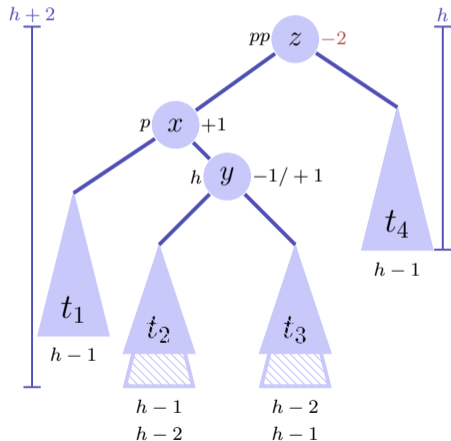
$\Rightarrow$   
Rotation  
nach  
rechts



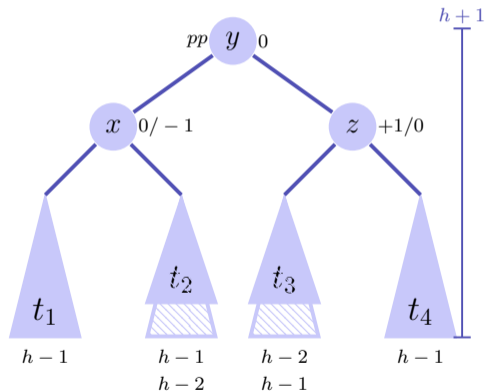
<sup>33</sup> $p$  rechter Sohn  $\Rightarrow \text{bal}(pp) = \text{bal}(p) = +1$ , Linksrotation

# Rotationen

Fall 1.2  $\text{bal}(p) = +1$ .<sup>34</sup>



$\implies$   
Doppel-  
rotation  
links-  
rechts



<sup>34</sup> $p$  rechter Sohn  $\implies \text{bal}(pp) = +1, \text{bal}(p) = -1$ , Doppelrotation rechts links



# Analyse

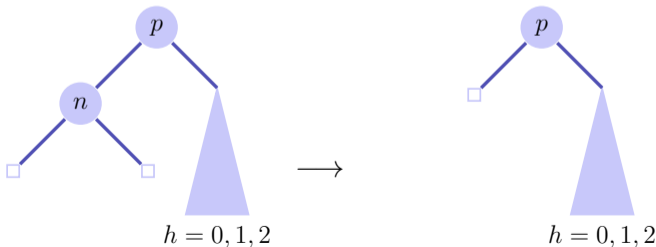
- Höhe des Baumes:  $\mathcal{O}(\log n)$ .
- Einfügen wie beim binären Suchbaum.
- Balancieren durch Rekursion vom Knoten zur Wurzel. Maximale Pfadlänge  $\mathcal{O}(\log n)$ .

Das Einfügen im AVL-Baum hat Laufzeitkosten von  $\mathcal{O}(\log n)$ .

# Löschen

Fall 1: Knoten  $n$  hat zwei Blätter als Kinder Sei  $p$  Elternknoten von  $n$ .  
 $\Rightarrow$  Anderer Teilbaum hat Höhe  $h' = 0, 1$  oder  $2$

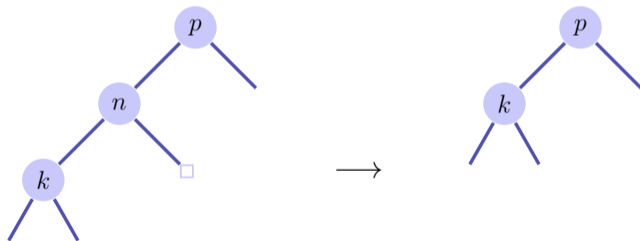
- $h' = 1$ :  $\text{bal}(p)$  anpassen.
- $h' = 0$ :  $\text{bal}(p)$  anpassen. Aufruf  $\text{upout}(p)$ .
- $h' = 2$ : Rebalancieren des Teilbaumes. Aufruf  $\text{upout}(p)$ .



# Löschen

Fall 2: Knoten  $n$  hat einen inneren Knoten  $k$  als Kind

- Ersetze  $n$  durch  $k$ . `upout(k)`



Fall 3: Knoten  $n$  hat zwei inneren Knoten als Kinder

- Ersetze  $n$  durch symmetrischen Nachfolger. `upout(k)`
- Löschen des symmetrischen Nachfolgers wie in Fall 1 oder 2.

# upout(p)

Sei  $pp$  der Elternknoten von  $p$

(a)  $p$  linkes Kind von  $pp$

1  $\text{bal}(pp) = -1 \Rightarrow \text{bal}(pp) \leftarrow 0$ . **upout(pp)**

2  $\text{bal}(pp) = 0 \Rightarrow \text{bal}(pp) \leftarrow +1$ .

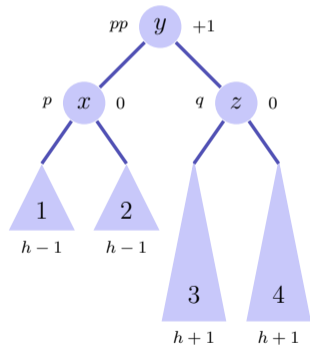
3  $\text{bal}(pp) = +1 \Rightarrow$  nächste Folien.

(b)  $p$  rechtes Kind von  $pp$ : Symmetrische Fälle unter Vertauschung von  $+1$  und  $-1$ .

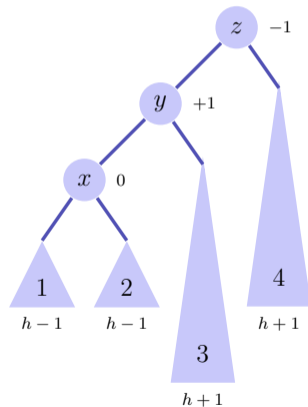
# upout (p)

Fall (a).3:  $\text{bal}(pp) = +1$ . Sei  $q$  Bruder von  $p$

(a).3.1:  $\text{bal}(q) = 0$ .<sup>35</sup>



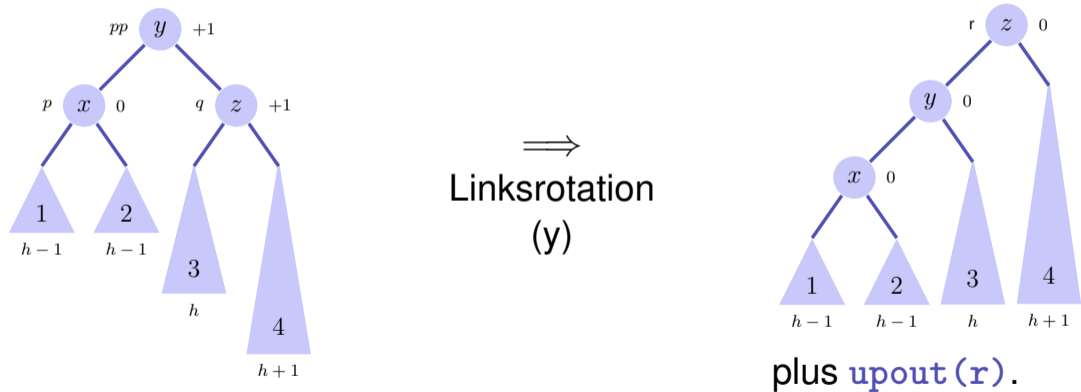
$\implies$   
Linksrotation  
(y)



<sup>35</sup>(b).3.1:  $\text{bal}(pp) = -1$ ,  $\text{bal}(q) = -1$ , Rechtsrotation.

# upout (p)

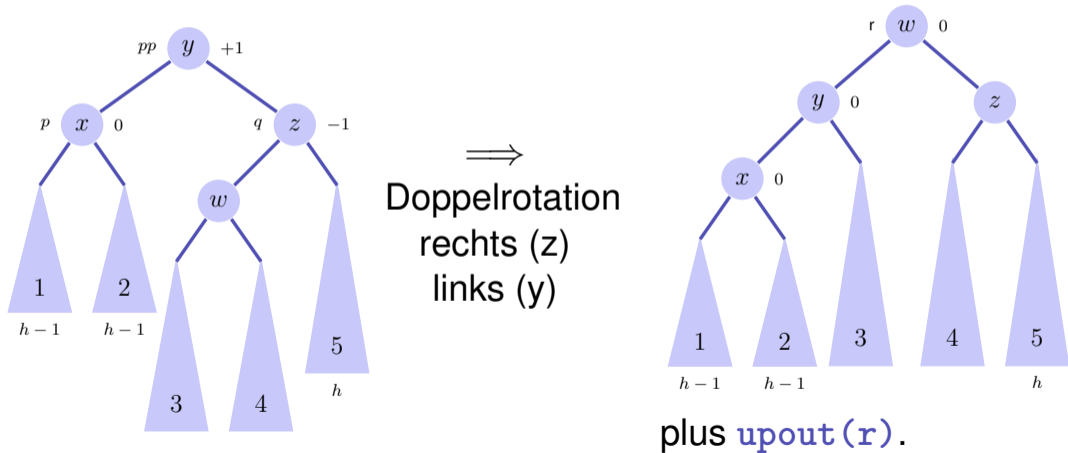
Fall (a).3:  $\text{bal}(pp) = +1$ . (a).3.2:  $\text{bal}(q) = +1$ .<sup>36</sup>



<sup>36</sup>(b).3.2:  $\text{bal}(pp) = -1$ ,  $\text{bal}(q) = +1$ , Rechtsrotation+upout

# upout (p)

Fall (a).3:  $\text{bal}(pp) = +1$ . (a).3.3:  $\text{bal}(q) = -1$ .<sup>37</sup>



<sup>37</sup>(b).3.3:  $\text{bal}(pp) = -1$ ,  $\text{bal}(q) = -1$ , Links-Rechts-Rotation + upout



# Zusammenfassung

- AVL-Bäume haben asymptotische Laufzeit von  $\mathcal{O}(\log n)$  (schlechtester Fall) für das Suchen, Einfügen und Löschen von Schlüsseln
- Einfügen und Löschen ist verhältnismässig aufwändig und für kleine Probleme relativ langsam.

# 17.5 Anhang

Herleitung einiger mathematischen Formeln

# [Fibonacci Zahlen: geschlossene Form]

Geschlossene Form der Fibonacci Zahlen: Berechnung über erzeugende Funktionen:

## 1 Potenzreihenansatz

$$f(x) := \sum_{i=0}^{\infty} F_i \cdot x^i$$

# [Fibonacci Zahlen: geschlossene Form]

- 2 Für Fibonacci Zahlen gilt  $F_0 = 0$ ,  $F_1 = 1$ ,  
 $F_i = F_{i-1} + F_{i-2} \forall i > 1$ . Daher:

$$\begin{aligned} f(x) &= x + \sum_{i=2}^{\infty} F_i \cdot x^i = x + \sum_{i=2}^{\infty} F_{i-1} \cdot x^i + \sum_{i=2}^{\infty} F_{i-2} \cdot x^i \\ &= x + x \sum_{i=2}^{\infty} F_{i-1} \cdot x^{i-1} + x^2 \sum_{i=2}^{\infty} F_{i-2} \cdot x^{i-2} \\ &= x + x \sum_{i=0}^{\infty} F_i \cdot x^i + x^2 \sum_{i=0}^{\infty} F_i \cdot x^i \\ &= x + x \cdot f(x) + x^2 \cdot f(x). \end{aligned}$$

# [Fibonacci Zahlen: geschlossene Form]

3 Damit:

$$f(x) \cdot (1 - x - x^2) = x.$$
$$\Leftrightarrow f(x) = \frac{x}{1 - x - x^2} = -\frac{x}{x^2 + x - 1}$$

Mit den Wurzeln  $-\phi$  und  $-\hat{\phi}$  von  $x^2 + x - 1$ ,

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.6, \quad \hat{\phi} = \frac{1 - \sqrt{5}}{2} \approx -0.6.$$

gilt  $\phi \cdot \hat{\phi} = -1$  und somit

$$f(x) = -\frac{x}{(x + \phi) \cdot (x + \hat{\phi})} = \frac{x}{(1 - \phi x) \cdot (1 - \hat{\phi} x)}$$

# [Fibonacci Zahlen: geschlossene Form]

4 Es gilt:

$$(1 - \hat{\phi}x) - (1 - \phi x) = \sqrt{5} \cdot x.$$

Damit:

$$\begin{aligned} f(x) &= \frac{1}{\sqrt{5}} \frac{(1 - \hat{\phi}x) - (1 - \phi x)}{(1 - \phi x) \cdot (1 - \hat{\phi}x)} \\ &= \frac{1}{\sqrt{5}} \left( \frac{1}{1 - \phi x} - \frac{1}{1 - \hat{\phi}x} \right) \end{aligned}$$

# [Fibonacci Zahlen: geschlossene Form]

5 Potenzreihenentwicklung von  $g_a(x) = \frac{1}{1-a \cdot x}$  ( $a \in \mathbb{R}$ ):

$$\frac{1}{1 - a \cdot x} = \sum_{i=0}^{\infty} a^i \cdot x^i.$$

Sieht man mit Taylor-Entwicklung von  $g_a(x)$  um  $x = 0$  oder so: Sei  $\sum_{i=0}^{\infty} G_i \cdot x^i$  eine Potenzreihenentwicklung von  $g$ . Mit der Identität  $g_a(x)(1 - a \cdot x) = 1$  gilt für alle  $x$  (im Konvergenzradius)

$$1 = \sum_{i=0}^{\infty} G_i \cdot x^i - a \cdot \sum_{i=0}^{\infty} G_i \cdot x^{i+1} = G_0 + \sum_{i=1}^{\infty} (G_i - a \cdot G_{i-1}) \cdot x^i$$

Für  $x = 0$  folgt  $G_0 = 1$  und für  $x \neq 0$  folgt dann  $G_i = a \cdot G_{i-1} \Rightarrow G_i = a^i$ .

# [Fibonacci Zahlen: geschlossene Form]

6 Einsetzen der Potenzreihenentwicklung:

$$\begin{aligned} f(x) &= \frac{1}{\sqrt{5}} \left( \frac{1}{1 - \phi x} - \frac{1}{1 - \hat{\phi} x} \right) = \frac{1}{\sqrt{5}} \left( \sum_{i=0}^{\infty} \phi^i x^i - \sum_{i=0}^{\infty} \hat{\phi}^i x^i \right) \\ &= \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}} (\phi^i - \hat{\phi}^i) x^i \end{aligned}$$

Koeffizientenvergleich mit  $f(x) = \sum_{i=0}^{\infty} F_i \cdot x^i$  liefert

$$F_i = \frac{1}{\sqrt{5}} (\phi^i - \hat{\phi}^i).$$

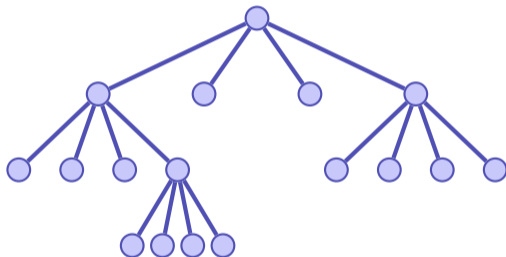


# 18. Quadrees

Quadrees, Kollisionsdetektion, Bildsegmentierung

# Quadtree

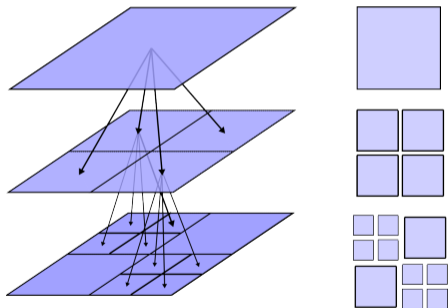
Ein Quadtree ist ein Baum der Ordnung 4.



... und ist als solcher nicht besonders interessant, ausser man verwendet ihn zur...

# Quadtree - Interpretation und Nutzen

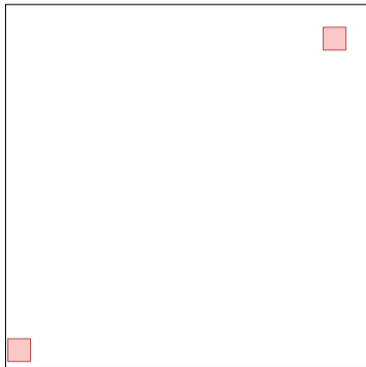
Partitionierung eines zweidimensionalen Bereiches in 4 gleich grosse Teile.



[Analog für drei Dimensionen mit einem *Octtree* (Baum der Ordnung 8)]

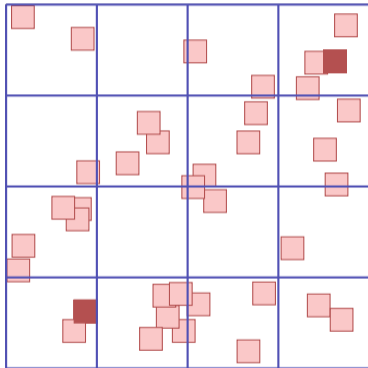
# Beispiel 1: Erkennung von Kollisionen

- Objekte in der 2D-Ebene, z.B. Teilchensimulation auf dem Bildschirm.
- Ziel: Erkennen von Kollisionen



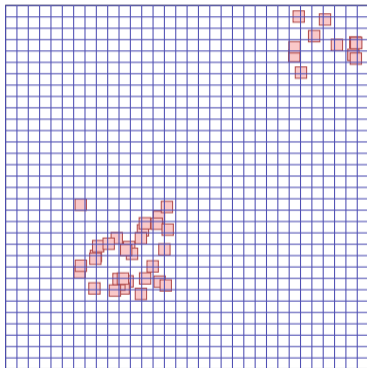
# Idee

- Viele Objekte:  $n^2$  Vergleiche (naiv)
- Verbesserung?
- Offensichtlich: keine Kollisionsdetektion für weit entfernte Objekte nötig.
- Was ist „weit entfernt“?
- Gitter ( $m \times m$ )
- Kollisionsdetektion pro Gitterzelle



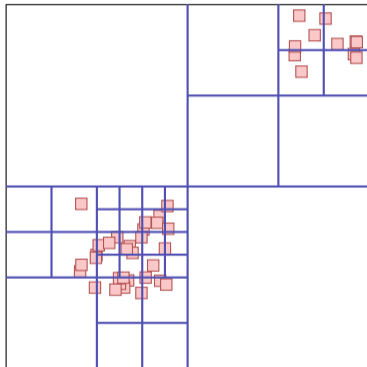
# Gitter

- Gitter hilft oft, aber nicht immer
- Verbesserung?
- Gitter verfeinern?
- Zu viele Gitterzellen!



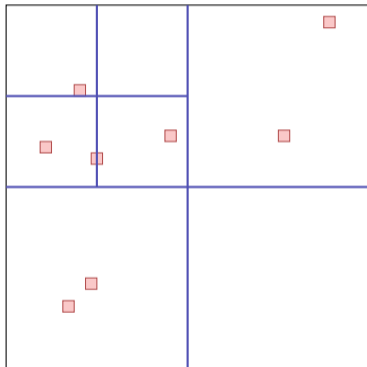
# Adaptive Gitter

- Gitter hilft oft, aber nicht immer
- Verbesserung?
- Gitter *adaptiv* verfeinern!
- Quadtree!



# Algorithmus: Einfügen

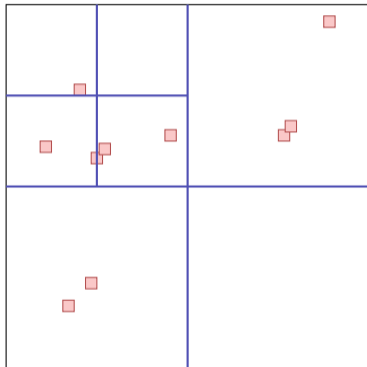
- Quadtree startet mit einem einzigen Knoten
- Objekte werden zu dem Knoten hinzugefügt. Wenn in einem Knoten zu viele Objekte sind, wird der Knoten geteilt.
- Objekte, die beim Split auf dem Rand zu liegen kommen, werden im höher gelegenen Knoten belassen.



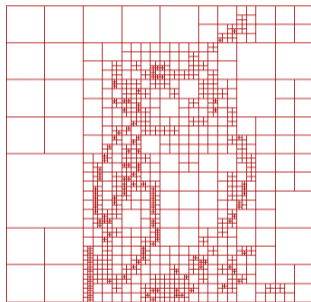
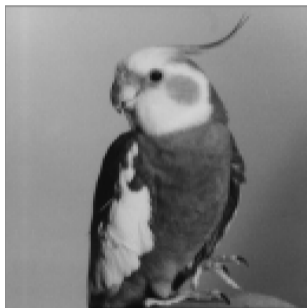


# Algorithmus: Kollisionsdetektion

- Durchlaufe den Quadtree rekursiv. Für jeden Knoten teste die Kollision der enthaltenen Objekte mit Objekten im selben Knoten oder (rekursiv) enthaltenen Knoten.

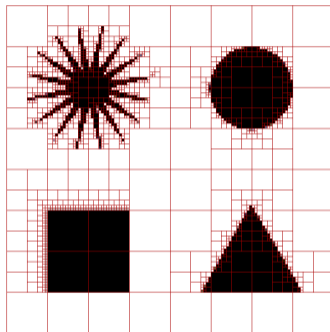
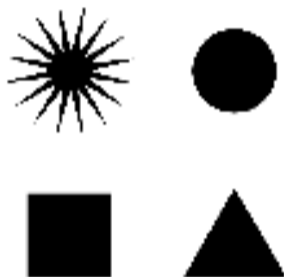


## Beispiel 2: Bildsegmentierung



(Mögliche Anwendungen: Kompression, Entrauschen, Kantendetektion)

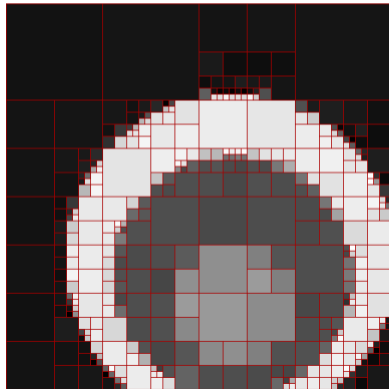
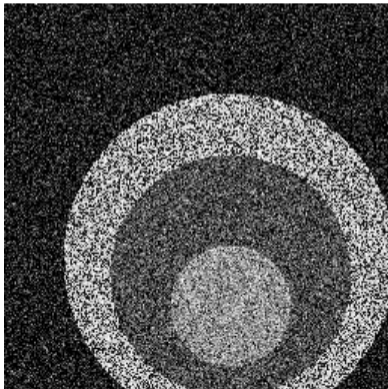
# Quadtree auf Einfarbenbild



Erzeugung des Quadtree ähnlich wie oben: unterteile Knoten rekursiv bis jeder Knoten nur Pixel einer Farbe enthält.

# Quadtree mit Approximation

Wenn mehr als zwei Farbewerte vorhanden sind, wird der Quadtree oft sehr gross.  $\Rightarrow$  Komprimierte Darstellung: *approximiere* das Bild stückweise konstant auf Rechtecken eines Quadrees.



# Stückweise konstante Approximation

(Graustufen-)Bild  $z \in \mathbb{R}^S$  auf den Pixelindizes  $S$ .<sup>38</sup>

Rechteck  $r \subset S$ .

Ziel: bestimme

$$\arg \min_{x \in \mathbb{R}} \sum_{s \in r} (z_s - x)^2$$

Lösung: das arithmetische Mittel  $\mu_r = \frac{1}{|r|} \sum_{s \in r} z_s$

---

<sup>38</sup>Wir nehmen an, dass  $S$  ein Quadrat ist mit Seitenlänge  $2^k$  für ein  $k \geq 0$

# Zwischenergebnis

Die im Sinne des mittleren quadratischen Fehlers beste Approximation

$$\mu_r = \frac{1}{|r|} \sum_{s \in r} z_s$$

und der dazugehörige Fehler

$$\sum_{s \in r} (z_s - \mu_r)^2 =: \|z_r - \mu_r\|_2^2$$

können nach einer  $\mathcal{O}(|S|)$  Tabellierung schnell berechnet werden:  
Präfixsummen!

# Welcher Quadtree?

## Konflikt

- *Möglichst nahe an den Daten*  $\Rightarrow$  kleine Rechtecke, grosser Quadtree. Extremer Fall: ein Knoten pro Pixel. Approximation = Original
- *Möglichst wenige Knoten*  $\Rightarrow$  Grosse Rechtecke, kleiner Quadtree. Extremfall: ein einziges Rechteck. Approximation = ein Grauwert

# Welcher Quadtree?

Idee: wähle zwischen Datentreue und Komplexität durch Einführung eines Regularisierungsparameters  $\gamma \geq 0$

Wähle Quadtree  $T$  mit Blättern<sup>39</sup>  $L(T)$  so, dass  $T$  folgenden Funktion minimiert

$$H_\gamma(T, z) := \gamma \cdot \underbrace{|L(T)|}_{\text{Anzahl Blätter}} + \underbrace{\sum_{r \in L(T)} \|z_r - \mu_r\|_2^2}_{\text{Summierter Approximationsfehler aller Blätter}} .$$

---

<sup>39</sup>hier: Blatt = Knoten mit Nullkindern,



# Regularisierung

Sei  $T$  ein Quadtree über einem Rechteck  $S_T$  und seien  $T_{ll}, T_{lr}, T_{ul}, T_{ur}$  vier mögliche Unterbäume und

$$\hat{H}_\gamma(T, z) := \min_T \gamma \cdot |L(T)| + \sum_{r \in L(T)} \|z_r - \mu_r\|_2^2$$

Extremfälle:

$\gamma = 0 \Rightarrow$  Originaldaten;

$\gamma \rightarrow \infty \Rightarrow$  ein Rechteck

# Beobachtung: Rekursion

- Wenn der (Sub-)Quadtree  $T$  nur ein Pixel hat, so kann nicht aufgeteilt werden und es gilt

$$\hat{H}_\gamma(T, z) = \gamma$$

- Andernfalls seien

$$M_1 := \gamma + \|z_{S_T} - \mu_{S_T}\|_2^2$$

$$M_2 := \hat{H}_\gamma(T_{ll}, z) + \hat{H}_\gamma(T_{lr}, z) + \hat{H}_\gamma(T_{ul}, z) + \hat{H}_\gamma(T_{ur}, z)$$

Dann

$$\hat{H}_\gamma(T, z) = \min \left\{ \underbrace{M_1(T, \gamma, z)}_{\text{kein Split}}, \underbrace{M_2(T, \gamma, z)}_{\text{Split}} \right\}$$

# Algorithmus: Minimize( $z, r, \gamma$ )

**Input:** Bilddaten  $z \in \mathbb{R}^S$ , Rechteck  $r \subset S$ , Regularisierung  $\gamma > 0$

**Output:**  $\min_T \gamma |L(T)| + \|z - \mu_{L(T)}\|_2^2$

**if**  $|r| = 0$  **then return** 0

$m \leftarrow \gamma + \sum_{s \in r} (z_s - \mu_r)^2$

**if**  $|r| > 1$  **then**

    Split  $r$  into  $r_{ll}, r_{lr}, r_{ul}, r_{ur}$

$m_1 \leftarrow \text{Minimize}(z, r_{ll}, \gamma)$ ;  $m_2 \leftarrow \text{Minimize}(z, r_{lr}, \gamma)$

$m_3 \leftarrow \text{Minimize}(z, r_{ul}, \gamma)$ ;  $m_4 \leftarrow \text{Minimize}(z, r_{ur}, \gamma)$

$m' \leftarrow m_1 + m_2 + m_3 + m_4$

**else**

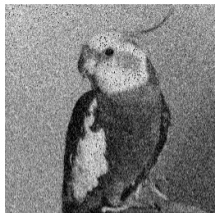
$m' \leftarrow \infty$

**if**  $m' < m$  **then**  $m \leftarrow m'$

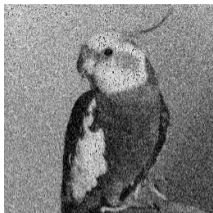
**return**  $m$

Der Minimierungsalgorithmus über dyadische Partitionen (Quadtree) benötigt  $\mathcal{O}(|S| \log |S|)$  Schritte.

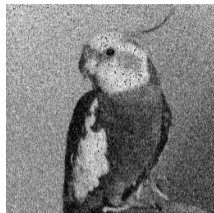
# Anwendung: Entrauschen (zusätzlich mit Wedgelets)



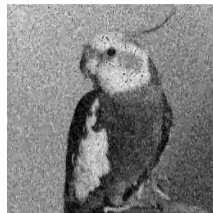
noised



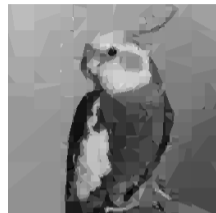
$\gamma = 0.003$



$\gamma = 0.01$



$\gamma = 0.03$



$\gamma = 0.1$



$\gamma = 0.3$



$\gamma = 1$

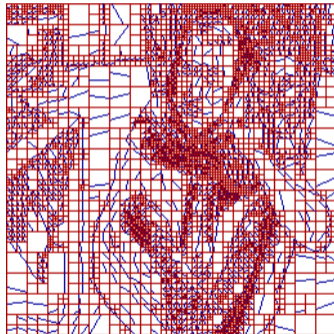


$\gamma = 3$



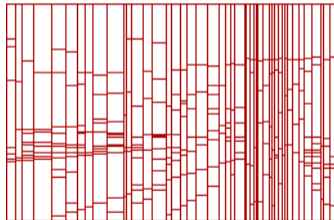
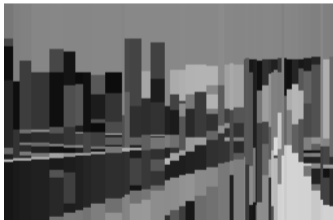
$\gamma = 10$

# Erweiterungen: Affine Regression + Wedgelets



# Andere Ideen

kein Quadtree: hierarchisch-eindimensionales Modell (benötigt Dynamic Programming)



# 19. Dynamische Programmierung I

Memoisieren, Optimale Substruktur, Überlappende Teilprobleme, Abhängigkeiten, Allgemeines Vorgehen. Beispiele: Fibonacci, Schneiden von Eisenstangen, Längste aufsteigende Teilfolge, längste gemeinsame Teilfolge, Editierdistanz, Matrixkettenmultiplikation, Matrixmultiplikation nach Strassen  
[Ottman/Widmayer, Kap. 1.2.3, 7.1, 7.4, Cormen et al, Kap. 15]



# Fibonacci Zahlen



(schon wieder)

$$F_n := \begin{cases} n & \text{wenn } n < 2 \\ F_{n-1} + F_{n-2} & \text{wenn } n \geq 2. \end{cases}$$

Analyse: warum ist der rekursive Algorithmus so langsam.

# Algorithmus FibonacciRecursive( $n$ )

**Input:**  $n \geq 0$

**Output:**  $n$ -te Fibonacci Zahl

**if**  $n < 2$  **then**

  |  $f \leftarrow n$

**else**

  |  $f \leftarrow \text{FibonacciRecursive}(n - 1) + \text{FibonacciRecursive}(n - 2)$

**return**  $f$

# Analyse

$T(n)$ : Anzahl der ausgeführten Operationen.

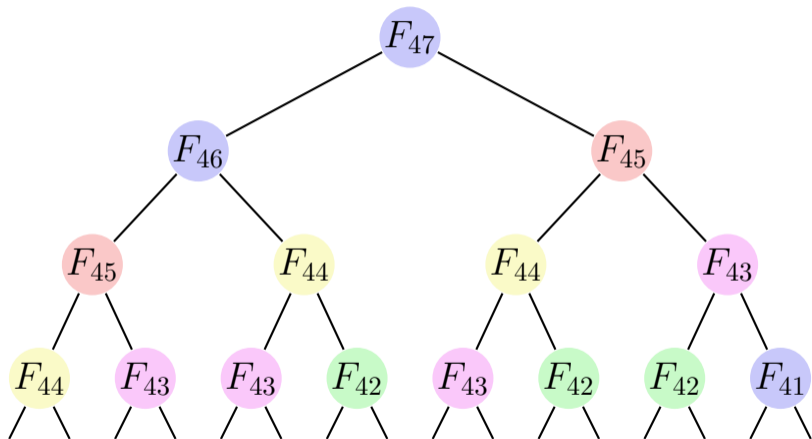
■  $n = 0, 1: T(n) = \Theta(1)$

■  $n \geq 2: T(n) = T(n - 2) + T(n - 1) + c.$

$$T(n) = T(n - 2) + T(n - 1) + c \geq 2T(n - 2) + c \geq 2^{n/2}c' = (\sqrt{2})^n c'$$

Algorithmus ist *exponentiell (!)* in  $n$ .

# Grund, visualisiert



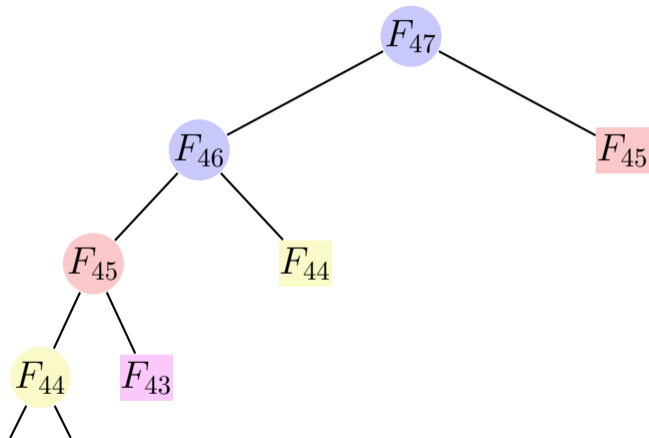
Knoten mit denselben Werten werden (zu) oft ausgewertet.

# Memoization

*Memoization* (sic) Abspeichern von Zwischenergebnissen.

- Bevor ein Teilproblem gelöst wird, wird Existenz eines entsprechenden Zwischenergebnis geprüft.
- Existiert ein gespeichertes Zwischenergebnis bereits, so wird dieses verwendet.
- Andernfalls wird der Algorithmus ausgeführt und das Ergebnis wird entsprechend gespeichert.

# Memoization bei Fibonacci



Rechteckige Knoten wurden bereits ausgewertet.

# Algorithmus FibonacciMemoization( $n$ )

**Input:**  $n \geq 0$

**Output:**  $n$ -te Fibonacci Zahl

**if**  $n \leq 2$  **then**

|  $f \leftarrow 1$

**else if**  $\exists \text{memo}[n]$  **then**

|  $f \leftarrow \text{memo}[n]$

**else**

|  $f \leftarrow \text{FibonacciMemoization}(n - 1) + \text{FibonacciMemoization}(n - 2)$

|  $\text{memo}[n] \leftarrow f$

**return**  $f$

# Analyse

Berechnungsaufwand:

$$T(n) = T(n - 1) + c = \dots = \mathcal{O}(n).$$

denn nach dem Aufruf von  $f(n - 1)$  wurde  $f(n - 2)$  bereits berechnet.

Das lässt sich auch so sehen: Für jedes  $n$  wird  $f(n)$  maximal einmal rekursiv berechnet. Laufzeitkosten:  $n$  Aufrufe mal  $\Theta(1)$  Kosten pro Aufruf  $n \cdot c \in \Theta(n)$ . Die Rekursion verschwindet aus der Berechnung der Laufzeit.

Algorithmus benötigt  $\Theta(n)$  Speicher.<sup>40</sup>

<sup>40</sup>Allerdings benötigt der naive Algorithmus auch  $\Theta(n)$  Speicher für die Rekursionsverwaltung.



## Genauer hingesehen ...

... berechnet der Algorithmus der Reihe nach die Werte  $F_1, F_2, F_3,$   
... verkleidet im *Top-Down* Ansatz der Rekursion.

Man kann den Algorithmus auch gleich *Bottom-Up* hinschreiben.  
Das ist charakteristisch für die *dynamische Programmierung*.

# Algorithmus FibonacciBottomUp(n)

**Input:**  $n \geq 0$

**Output:**  $n$ -te Fibonacci Zahl

$F[1] \leftarrow 1$

$F[2] \leftarrow 1$

**for**  $i \leftarrow 3, \dots, n$  **do**

$F[i] \leftarrow F[i - 1] + F[i - 2]$

**return**  $F[n]$

# Dynamische Programmierung: Idee

- Aufteilen eines komplexen Problems in eine vernünftige Anzahl kleinerer Teilprobleme
- Die Lösung der Teilprobleme wird zur Lösung des komplexeren Problems verwendet
- Identische Teilprobleme werden nur einmal gerechnet

# Dynamische Programmierung: Konsequenz

Identische Teilprobleme werden nur einmal gerechnet

⇒ Resultate werden zwischengespeichert

Arbeitsspeicher



192.-

HyperX Fury (2x, 8GB,  
DDR4-2400, DIMM 288)

★★★★ 16

Wir tauschen Laufzeit  
gegen Speicherplatz

# Dynamic Programming: Beschreibung

- 1 Verwalte *DP-Tabelle* mit Information zu den Teilproblemen.  
Dimension der Tabelle? Bedeutung der Einträge?
- 2 Berechnung der *Randfälle*.  
Welche Einträge hängen nicht von anderen ab?
- 3 *Berechnungsreihenfolge* bestimmen.  
In welcher Reihenfolge können Einträge berechnet werden, so dass benötigte Einträge jeweils vorhanden sind?
- 4 Auslesen der *Lösung*.  
Wie kann sich Lösung aus der Tabelle konstruieren lassen?

Laufzeit (typisch) = Anzahl Einträge der Tabelle mal Aufwand pro Eintrag.

# Dynamic Programming: Beschreibung am Beispiel

1 Dimension der Tabelle? Bedeutung der Einträge?

Tabelle der Grösse  $n \times 1$ .  $n$ -ter Eintrag enthält  $n$ -te Fibonacci Zahl.

2 Welche Einträge hängen nicht von anderen ab?

Werte  $F_1$  und  $F_2$  sind unabhängig einfach "berechenbar".

3 In welcher Reihenfolge können Einträge berechnet werden, so dass benötigte Einträge jeweils vorhanden sind?

$F_i$  mit aufsteigenden  $i$ .

4 Wie kann sich Lösung aus der Tabelle konstruieren lassen?

$F_n$  ist die  $n$ -te Fibonacci-Zahl.

# Dynamic Programming = Divide-And-Conquer ?

- In beiden Fällen ist das Ursprungsproblem (einfacher) lösbar, indem Lösungen von Teilproblemen herangezogen werden können. Das Problem hat *optimale Substruktur*.
- Bei Divide-And-Conquer Algorithmen (z.B. Mergesort) sind Teilprobleme unabhängig; deren Lösungen werden im Algorithmus nur einmal benötigt.
- Beim DP sind Teilprobleme nicht unabhängig. Das Problem hat *überlappende Teilprobleme*, welche im Algorithmus mehrfach gebraucht werden.
- Damit sie nur einmal gerechnet werden müssen, werden Resultate tabelliert. Dafür darf es *zwischen Teilproblemen keine zirkulären Abhängigkeiten* geben.

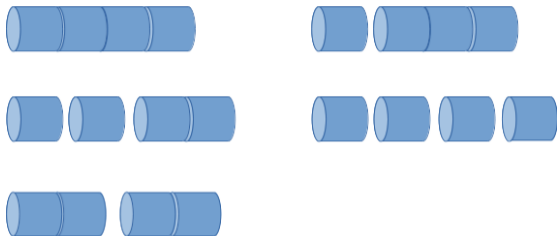
# Schneiden von Eisenstäben

- Metallstäbe werden zerschnitten und verkauft.
- Metallstäbe der Länge  $n \in \mathbb{N}$  verfügbar. Zerschneiden kostet nichts.
- Für jede Länge  $l \in \mathbb{N}$ ,  $l \leq n$  bekannt: Wert  $v_l \in \mathbb{R}^+$
- Ziel: Zerschneide die Stange so (in  $k \in \mathbb{N}$  Stücke), dass

$$\sum_{i=1}^k v_{l_i} \text{ maximal unter } \sum_{i=1}^k l_i = n.$$



# Schneiden von Eisenstäben: Beispiel



Arten, einen Stab der Länge 4 zu zerschneiden (ohne Permutationen)

Länge	0	1	2	3	4
Preis	0	2	3	8	9

⇒ Bester Schnitt: 3 + 1 mit Wert 10.

# Wie findet man den DP Algorithmus

- 0 Genaue Formulierung der gesuchten Lösung
- 1 Definiere Teilprobleme (und bestimme deren Anzahl)
- 2 Raten / Aufzählen (und bestimme die Laufzeit für das Raten)
- 3 Rekursion: verbinde die Teilprobleme
- 4 Memoisieren / Tabellieren. Bestimme die Abhängigkeiten der Teilprobleme
- 5 Lösung des Problems  
Laufzeit = #Teilprobleme  $\times$  Zeit/Teilproblem

# Struktur des Problems

- 0 **Gesucht:**  $r_n$  = maximal erreichbarer Wert von (ganzem oder geschnittenem) Stab mit Länge  $n$ .
- 1 **Teilprobleme:** maximal erreichbarer Wert  $r_k$  für alle  $0 \leq k < n$
- 2 **Rate** Länge des ersten Stückes
- 3 **Rekursion**

$$r_k = \max \{v_i + r_{k-i} : 0 < i \leq k\}, \quad k > 0$$

$$r_0 = 0$$

- 4 **Abhängigkeit:**  $r_k$  hängt (nur) ab von den Werten  $v_i, l \leq i \leq k$  und den optimalen Schnitten  $r_i, i < k$
- 5 **Lösung** in  $r_n$

# Algorithmus RodCut( $v, n$ )

**Input:**  $n \geq 0$ , Preise  $v$

**Output:** bester Wert

$q \leftarrow 0$

**if**  $n > 0$  **then**

**for**  $i \leftarrow 1, \dots, n$  **do**  
         $q \leftarrow \max\{q, v_i + \text{RodCut}(v, n - i)\};$

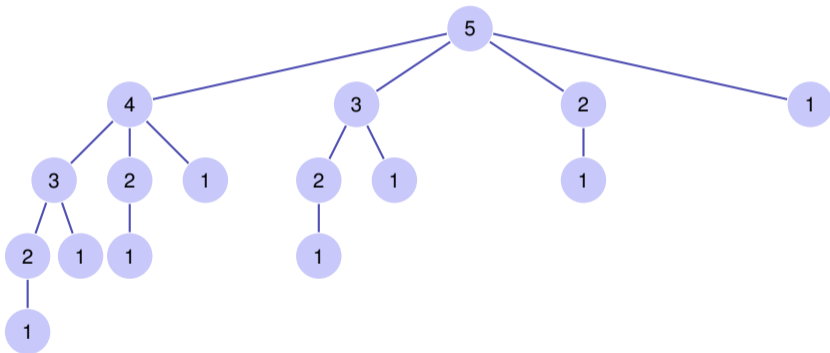
**return**  $q$

Laufzeit  $T(n) = \sum_{i=0}^{n-1} T(i) + c \Rightarrow^{41} T(n) \in \Theta(2^n)$

---

<sup>41</sup> $T(n) = T(n-1) + \sum_{i=0}^{n-2} T(i) + c = T(n-1) + (T(n-1) - c) + c = 2T(n-1) \quad (n > 0)$

# Rekursionsbaum



# Algorithmus RodCutMemoized( $m, v, n$ )

**Input:**  $n \geq 0$ , Preise  $v$ , Memoization Tabelle  $m$

**Output:** bester Wert

$q \leftarrow 0$

**if**  $n > 0$  **then**

**if**  $\exists m[n]$  **then**

$q \leftarrow m[n]$

**else**

**for**  $i \leftarrow 1, \dots, n$  **do**

$q \leftarrow \max\{q, v_i + \text{RodCutMemoized}(m, v, n - i)\};$

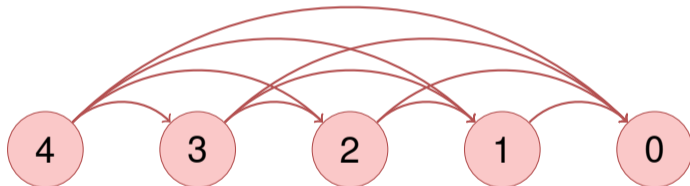
$m[n] \leftarrow q$

**return**  $q$

Laufzeit  $\sum_{i=1}^n i = \Theta(n^2)$

# Teilproblem-Graph

beschreibt die Abhängigkeiten der Teilprobleme untereinander



und darf keine Zyklen enthalten

# Konstruktion des optimalen Schnittes

- Während der (rekursiven) Berechnung der optimalen Lösung für jedes  $k \leq n$  bestimmt der rekursive Algorithmus die optimale Länge des ersten Stabes
- Speichere die Länge des ersten Stabes für jedes  $k \leq n$  in einer Tabelle mit  $n$  Einträgen.



# Bottom-Up Beschreibung am Beispiel

Dimension der Tabelle? Bedeutung der Einträge?

- 1 Tabelle der Grösse  $n \times 1$ .  $n$ -ter Eintrag enthält besten Wert eines Stabes der Länge  $n$ .

Welche Einträge hängen nicht von anderen ab?

- 2 Wert  $r_0$  ist 0.

In welcher Reihenfolge können Einträge berechnet werden, so dass benötigte Einträge jeweils vorhanden sind?

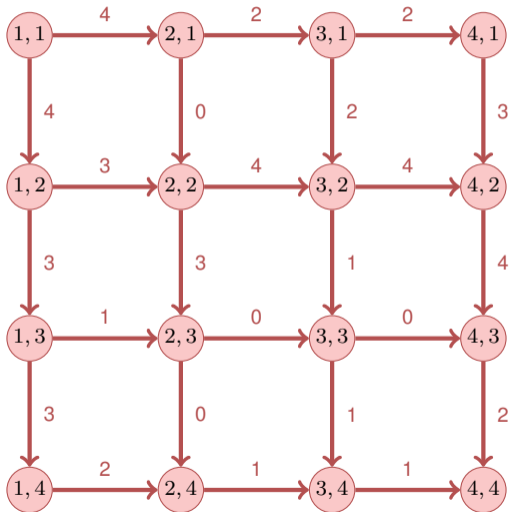
- 3  $r_i, i = 1, \dots, n$ .

Wie kann sich Lösung aus der Tabelle konstruieren lassen?

- 4  $r_n$  ist der beste Wert für eine Stange der Länge  $n$

# Kaninchen!

Ein Kaninchen sitzt auf Platz  $(1, 1)$  eines  $n \times n$  Gitters. Es kann nur nach Osten oder nach Süden gehen. Auf jedem Wegstück liegt eine Anzahl Rüben. Wie viele Rüben sammelt das Kaninchen maximal ein?



# Kaninchen!

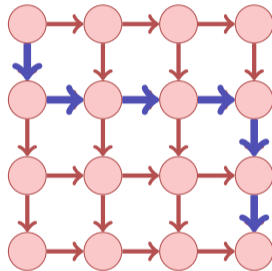
Anzahl mögliche Pfade?

- Auswahl von  $n - 1$  Wegen nach Süden aus  $2n - 2$  Wegen insgesamt.



$$\binom{2n - 2}{n - 1} \in \Omega(2^n)$$

⇒ Naiver Algorithmus hat keine Chance



Der Weg 100011  
(1:nach Süden, 0:nach Osten)

# Rekursion

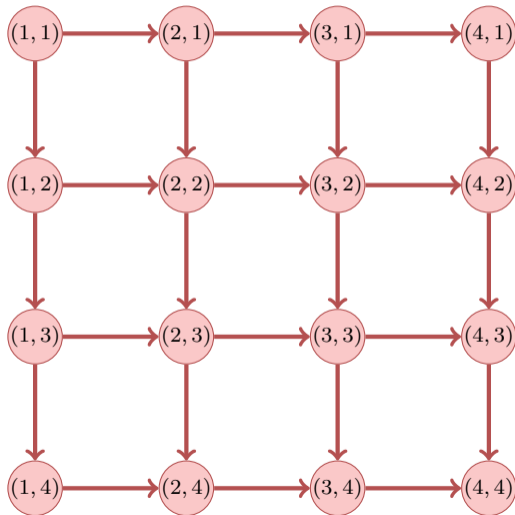
Gesucht:  $T_{0,0} = \text{Maximale Anzahl Rüben von } (0, 0) \text{ nach } (n, n)$ .

Sei  $w_{(i,j)-(i',j')}$  Anzahl Rüben auf Kante von  $(i, j)$  nach  $(i', j')$ .

Rekursion (maximale Anzahl Rüben von  $(i, j)$  nach  $(n, n)$ )

$$T_{ij} = \begin{cases} \max\{w_{(i,j)-(i,j+1)} + T_{i,j+1}, w_{(i,j)-(i+1,j)} + T_{i+1,j}\}, & i < n, j < n \\ w_{(i,j)-(i,j+1)} + T_{i,j+1}, & i = n, j < n \\ w_{(i,j)-(i+1,j)} + T_{i+1,j}, & i < n, j = n \\ 0 & i = j = n \end{cases}$$

# Teilproblemabhängigkeitsgraph



# Bottom-Up Beschreibung am Beispiel

Dimension der Tabelle? Bedeutung der Einträge?

- 1 Tabelle  $T$  der Grösse  $n \times n$ . Eintrag bei  $i, j$  enthält die maximale Anzahl Rüben von  $(i, j)$  nach  $(n, n)$ .

Welche Einträge hängen nicht von anderen ab?

- 2 Wert  $T_{n,n}$  ist 0.

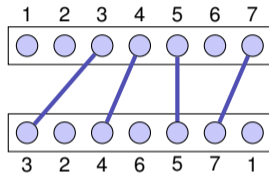
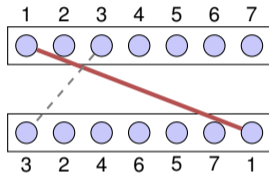
In welcher Reihenfolge können Einträge berechnet werden, so dass benötigte Einträge jeweils vorhanden sind?

- 3  $T_{i,j}$  mit  $i = n \searrow 1$  und für jedes  $i: j = n \searrow 1$ , (oder umgekehrt:  $j = n \searrow 1$  und für jedes  $j: i = n \searrow 1$ ).

Wie kann sich Lösung aus der Tabelle konstruieren lassen?

- 4  $T_{1,1}$  enthält die maximale Anzahl Rüben

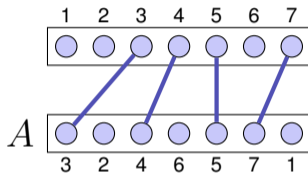
# Längste aufsteigende Teilfolge (LAT)



Verbinde so viele passende Anschlüsse wie möglich, ohne dass sich die Anschlüsse kreuzen.

# Formalisieren

- Betrachte Folge  $A_n = (a_1, \dots, a_n)$ .
- Suche eine längste aufsteigende Teilfolge von  $A_n$ .
- Beispiele aufsteigender Teilfolgen:  
 $(3, 4, 5)$ ,  $(2, 4, 5, 7)$ ,  $(3, 4, 5, 7)$ ,  $(3, 7)$ .



**Verallgemeinerung:** Lasse Zahlen ausserhalb von  $1, \dots, n$  zu, auch mit Mehrfacheinträgen. (Weitehrhin aber nur strikt aufsteigende Teilfolgen) Beispiel:  $(2,3,3,3,5,1)$  mit aufsteigender Teilfolge  $(2,3,5)$ .



# Erster Entwurf

Sei  $L_i = \text{längste Teilfolge von } A_i, (1 \leq i \leq n)$ .

Annahme: LAT  $L_k$  von  $A_k$  für bekannt. Wollen nun LAT  $L_{k+1}$  für  $A_{k+1}$  berechnen.

Wenn  $a_{k+1}$  zu  $L_k$  passt, dann  $L_{k+1} = L_k \oplus a_{k+1}$ ?

Gegenbeispiel:  $A_5 = (1, 2, 5, 3, 4)$ . Sei  $A_3 = (1, 2, 5)$  mit  $L_3 = A_3$ .  
Bestimme  $L_4$  aus  $L_3$ ?

So kommen wir nicht weiter: können nicht von  $L_k$  auf  $L_{k+1}$  schließen.

## Zweiter Entwurf

Sei  $L_i = \text{längste Teilfolge von } A_i, (1 \leq i \leq n)$ .

Annahme: eine LAT  $L_j$  für alle  $j \leq k$  bekannt. Wollen nun LAT  $L_{k+1}$  für  $k + 1$  berechnen.

Betrachte alle passenden  $L_{k+1} = L_j \oplus a_{k+1} (j \leq k)$  und wähle eine längste solche Folge.

Gegenbeispiel:  $A_5 = (1, 2, 5, 3, 4)$ . Sei  $A_4 = (1, 2, 5, 3)$  mit  $L_1 = (1)$ ,  $L_2 = (1, 2)$ ,  $L_3 = (1, 2, 5)$ ,  $L_4 = (1, 2, 5)$ . Bestimme  $L_5$  aus  $L_1, \dots, L_4$ ?

So kommen wir nicht weiter: können nicht von *jeweils nur einer beliebigen Lösung*  $L_j$  auf  $L_{k+1}$  schliessen. Wir müssten alle möglichen LAT betrachten. Zu viel!

# Dritter Entwurf

Sei  $M_{n,i}$  = *längste Teilfolge von  $A_n$  der Länge  $i$  ( $1 \leq i \leq n$ )*

Annahme: die LAT  $M_{k,j}$  für  $A_k$ , *welche mit kleinstem Element enden* seien für alle Längen  $1 \leq j \leq k$  bekannt.

Betrachte nun alle passenden  $M_{k,j} \oplus a_{k+1}$  ( $j \leq k$ ) und aktualisiere die Tabelle der längsten aufsteigenden Folgen, welche mit kleinstem Element enden.

# Dritter Entwurf Beispiel

Beispiel:  $A = (1, 1000, 1001, 4, 5, 2, 6, 7)$

$A$	LAT $M_{k,\cdot}$
1	(1)
+ 1000	(1), (1, 1000)
+ 1001	(1), (1, 1000), (1, 1000, 1001)
+ 4	(1), (1, 4), (1, 1000, 1001)
+ 5	(1), (1, 4), (1, 4, 5)
+ 2	(1), (1, 2), (1, 4, 5)
+ 6	(1), (1, 2), (1, 4, 5), (1, 4, 5, 6)
+ 7	(1), (1, 2), (1, 4, 5), (1, 4, 5, 6), (1, 4, 5, 6, 7)

# DP Table

- Idee: speichere jeweils nur das letzte Element der aufsteigenden Folge  $M_{k,j}$  am Slot  $j$ .
- Beispielfolge:  
3 2 5 1 6 4
- Problem: **Tabelle** enthält zum Schluss nicht die Folge, nur den letzten Wert.
- Lösung: **Zweite Tabelle** mit den Vorgängern.

Index	1	2	3	4	5	6
Wert	3	2	5	1	6	4
Vorgänger	$-\infty$	$-\infty$	2	$-\infty$	5	1

Index	0	1	2	3	4	...
$(L_j)_j$	$-\infty$	1	4	6	$\infty$	

# Dynamic Programming Algorithmus LAT

## Dimension der Tabelle? Bedeutung der Einträge?

Zwei Tabellen  $T[0, \dots, n]$  und  $V[1, \dots, n]$ .

1  $T[j]$ : letztes Element der aufsteigenden Folge  $M_{n,j}$

$V[j]$ : Wert des Vorgängers von  $a_j$ .

Zu Beginn  $T[0] \leftarrow -\infty, T[i] \leftarrow \infty \forall i > 1$

## Berechnung eines Eintrags

2 Einträge in  $T$  aufsteigend sortiert. Für jeden Neueintrag  $a_{k+1}$  binäre Suche nach  $l$ , so dass  $T[l] < a_k < T[l + 1]$ . Setze  $T[l + 1] \leftarrow a_{k+1}$ . Setze  $V[k] = T[l]$ .

# Dynamic Programming Algorithmus LAT

## Berechnungsreihenfolge

- 3 Beim Traversieren der Liste werden die Einträge  $T[k]$  und  $V[k]$  mit aufsteigendem  $k$  berechnet.

## Wie kann sich Lösung aus der Tabelle konstruieren lassen?

- 4 Suche das grösste  $l$  mit  $T[l] < \infty$ .  $l$  ist der letzte Index der LAT. Suche von  $l$  ausgehend den Index  $i < l$ , so dass  $V[l] = a_i$ ,  $i$  ist der Vorgänger von  $l$ .  
Repetiere mit  $l \leftarrow i$  bis  $T[l] = -\infty$

# Analyse

## ■ Berechnung Tabelle:

- Initialisierung:  $\Theta(n)$  Operationen
- Berechnung  $k$ -ter Eintrag: Binäre Suche auf Positionen  $\{1, \dots, k\}$  plus konstante Anzahl Zuweisungen.

$$\sum_{k=1}^n (\log k + \mathcal{O}(1)) = \mathcal{O}(n) + \sum_{k=1}^n \log(k) = \Theta(n \log n).$$

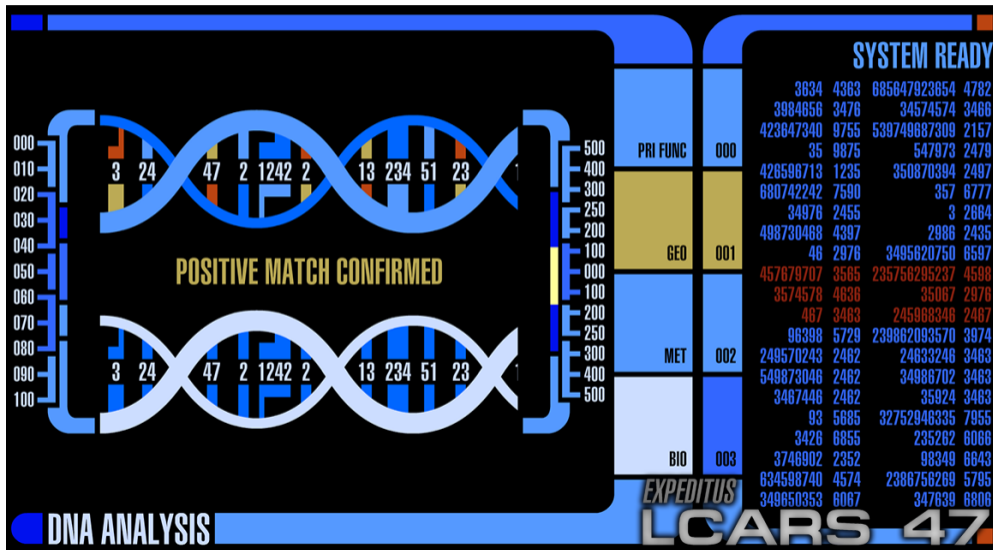
- **Rekonstruktion:** Traversiere  $A$  von rechts nach links:  $\mathcal{O}(n)$ .

Somit Gesamtlaufzeit

$$\Theta(n \log n).$$



# DNA - Vergleich (Star Trek)



# DNA - Vergleich

- DNA besteht aus Sequenzen von vier verschiedenen Nukleotiden **A**denin **G**uanin **T**hymine **C**ytosin
- DNA-Sequenzen (Gene) werden mit Zeichenketten aus A, G, T und C beschrieben.
- Ein möglicher Vergleich zweier Gene: Bestimme **Längste gemeinsame Teilfolge**

Das Problem, die längste gemeinsame Teilfolge zu finden ist ein Spezialfall der minimalen Editierdistanz. Die folgenden Folien werden daher in der Vorlesung nicht behandelt.

# [Längste Gemeinsame Teilfolge]

Teilfolgen einer Zeichenkette:

Teilfolgen(*KUH*):  $()$ ,  $(K)$ ,  $(U)$ ,  $(H)$ ,  $(KU)$ ,  $(KH)$ ,  $(UH)$ ,  $(KUH)$

Problem:

- **Eingabe:** Zwei Zeichenketten  $A = (a_1, \dots, a_m)$ ,  $B = (b_1, \dots, b_n)$  der Längen  $m > 0$  und  $n > 0$ .
- **Gesucht:** Eine längste gemeinsame Teilfolge (LGT) von  $A$  und  $B$ .

# [Längste Gemeinsame Teilfolge]

Beispiele:

$LGT(IGEL, KATZE) = E$ ,  $LGT(TIGER, ZIEGE) = IGE$

Ideen zur Lösung?

T	I		G	E	R
Z	I	E	G	E	

# [Rekursives Vorgehen]

**Annahme:** Lösungen  $L(i, j)$  bekannt für  $A[1, \dots, i]$  und  $B[1, \dots, j]$  für alle  $1 \leq i \leq m$  und  $1 \leq j \leq n$ , jedoch nicht für  $i = m$  und  $j = n$ .

T I G E R  
Z I E G E

Betrachten Zeichen  $a_m, b_n$ . Drei Möglichkeiten:

- 1  $A$  wird um ein Leerzeichen erweitert.  $L(m, n) = L(m, n - 1)$
- 2  $B$  wird um ein Leerzeichen erweitert.  $L(m, n) = L(m - 1, n)$
- 3  $L(m, n) = L(m - 1, n - 1) + \delta_{mn}$  mit  $\delta_{mn} = 1$  wenn  $a_m = b_n$  und  $\delta_{mn} = 0$  sonst

# [Rekursion]

$$L(m, n) \leftarrow \max \{L(m - 1, n - 1) + \delta_{mn}, L(m, n - 1), L(m - 1, n)\}$$

für  $m, n > 0$  und Randfälle  $L(\cdot, 0) = 0, L(0, \cdot) = 0$ .

	$\emptyset$	Z	I	E	G	E
$\emptyset$	0	0	0	0	0	0
T	0	0	0	0	0	0
I	0	0	1	1	1	1
G	0	0	1	1	2	2
E	0	0	1	2	2	3
R	0	0	1	2	2	3

# [Dynamic Programming Algorithmus LGT]

## Dimension der Tabelle? Bedeutung der Einträge?

- 1 Tabelle  $L[0, \dots, m][0, \dots, n]$ .  $L[i, j]$ : Länge einer LGT der Zeichenketten  $(a_1, \dots, a_i)$  und  $(b_1, \dots, b_j)$

## Berechnung eines Eintrags

- 2  $L[0, i] \leftarrow 0 \forall 0 \leq i \leq m$ ,  $L[j, 0] \leftarrow 0 \forall 0 \leq j \leq n$ . Berechnung von  $L[i, j]$  sonst mit  $L[i, j] = \max(L[i-1, j-1] + \delta_{ij}, L[i, j-1], L[i-1, j])$ .

# [Dynamic Programming Algorithmus LGT]

## Berechnungsreihenfolge

- 3 Abhängigkeiten berücksichtigen: z.B. Zeilen aufsteigend und innerhalb von Zeilen Spalten aufsteigend.

## Wie kann sich Lösung aus der Tabelle konstruieren lassen?

- 4 Beginne bei  $j = m, i = n$ . Falls  $a_i = b_j$  gilt, gib  $a_i$  aus und fahre fort mit  $(j, i) \leftarrow (j - 1, i - 1)$ ; sonst, falls  $L[i, j] = L[i, j - 1]$  fahre fort mit  $j \leftarrow j - 1$ ; sonst, falls  $L[i, j] = L[i - 1, j]$  fahre fort mit  $i \leftarrow i - 1$ . Terminiere für  $i = 0$  oder  $j = 0$ .



# [Analyse LGT]

- Anzahl Tabelleneinträge:  $(m + 1) \cdot (n + 1)$ .
- Berechnung jeweils mit konstanter Anzahl Zuweisungen und Vergleichen. Anzahl Schritte  $\mathcal{O}(mn)$
- Bestimmen der Lösung: jeweils Verringerung von  $i$  oder  $j$ . Maximal  $\mathcal{O}(n + m)$  Schritte.

Laufzeit insgesamt:

$$\mathcal{O}(mn).$$

# Minimale Editierdistanz

Editierdistanz von zwei Zeichenketten  $A_n = (a_1, \dots, a_m)$ ,  
 $B_m = (b_1, \dots, b_m)$ .

## Editieroperationen:

- Einfügen eines Zeichens
- Löschen eines Zeichens
- Änderung eines Zeichens

Frage: Wie viele Editieroperationen sind mindestens nötig, um eine gegebene Zeichenkette  $A$  in eine Zeichenkette  $B$  zu überführen.

*TIGER ZIGER ZIEGER ZIEGE*

# Minimale Editierdistanz

Gesucht: Günstigste zeichenweise Transformation  $A_n \rightarrow B_m$  mit Kosten

Operation	Levenshtein	LGT <sup>42</sup>	allgemein
$c$ einfügen	1	1	ins( $c$ )
$c$ löschen	1	1	del( $c$ )
Ersetzen $c \rightarrow c'$	$\mathbb{1}(c \neq c')$	$\infty \cdot \mathbb{1}(c \neq c')$	repl( $c, c'$ )

## Beispiel

T I G E R  
Z I E G E

T I \_ G E R  
Z I E G E \_

T $\rightarrow$ Z +E -R  
Z $\rightarrow$ T -E +R

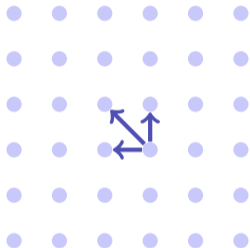
<sup>42</sup>Längste gemeinsame Teilfolge – Spezialfall des Editierproblems

# DP

- 0  $E(n, m)$  = minimale Anzahl Editieroperationen (ED Kosten) für  $a_{1\dots n} \rightarrow b_{1\dots m}$
- 1 Teilprobleme  $E(i, j)$  = ED von  $a_{1\dots i}$   $b_{1\dots j}$ . #TP =  $n \cdot m$
- 2 Raten/Probieren Kosten  $\Theta(1)$
- $a_{1\dots i} \rightarrow a_{1\dots i-1}$  (löschen)
  - $a_{1\dots i} \rightarrow a_{1\dots i}b_j$  (einfügen)
  - $a_{1\dots i} \rightarrow a_{1\dots i_1}b_j$  (ersetzen)
- 3 Rekursion

$$E(i, j) = \min \begin{cases} \text{del}(a_i) + E(i - 1, j), \\ \text{ins}(b_j) + E(i, j - 1), \\ \text{repl}(a_i, b_j) + E(i - 1, j - 1) \end{cases}$$

## 4 Abhängigkeiten



⇒ Berechnung von links oben nach rechts unten. Zeilen- oder Spaltenweise.

## 5 Lösung steht in $E(n, m)$

# Beispiel (Levenshteinabstand)

$$E[i, j] \leftarrow \min \{ E[i-1, j] + 1, E[i, j-1] + 1, E[i-1, j-1] + \mathbb{1}(a_i \neq b_j) \}$$

	$\emptyset$	Z	I	E	G	E
$\emptyset$	0	1	2	3	4	5
T	1	1	2	3	4	5
I	2	2	1	2	3	4
G	3	3	2	2	2	3
E	4	4	3	2	3	2
R	5	5	4	3	3	3

Editierschritte: von rechts unten nach links oben, der Rekursion folgend. Bottom-Up Beschreibung des Algorithmus: Übung

# Bottom-Up DP Algorithmus ED]

## Dimension der Tabelle? Bedeutung der Einträge?

- 1 Tabelle  $E[0, \dots, m][0, \dots, n]$ .  $E[i, j]$ : Minimaler Editierabstand der Zeichenketten  $(a_1, \dots, a_i)$  und  $(b_1, \dots, b_j)$

## Berechnung eines Eintrags

- 2  $E[0, i] \leftarrow i \forall 0 \leq i \leq m$ ,  $E[j, 0] \leftarrow j \forall 0 \leq j \leq n$ . Berechnung von  $E[i, j]$  sonst mit  $E[i, j] = \min\{\text{del}(a_i) + E(i-1, j), \text{ins}(b_j) + E(i, j-1), \text{repl}(a_i, b_j) + E(i-1, j-1)\}$

# Bottom-Up DP Algorithmus ED

## Berechnungsreihenfolge

- 3 Abhängigkeiten berücksichtigen: z.B. Zeilen aufsteigend und innerhalb von Zeilen Spalten aufsteigend.

## Wie kann sich Lösung aus der Tabelle konstruieren lassen?

- 4 Beginne bei  $j = m, i = n$ . Falls  $E[i, j] = \text{repl}(a_i, b_j) + E(i - 1, j - 1)$  gilt, gib  $a_i \rightarrow b_j$  aus und fahre fort mit  $(j, i) \leftarrow (j - 1, i - 1)$ ; sonst, falls  $E[i, j] = \text{del}(a_i) + E(i - 1, j)$  gib  $\text{del}(a_i)$  aus fahre fort mit  $j \leftarrow j - 1$ ; sonst, falls  $E[i, j] = \text{ins}(b_j) + E(i, j - 1)$ , gib  $\text{ins}(b_j)$  aus und fahre fort mit  $i \leftarrow i - 1$ . Terminiere für  $i = 0$  und  $j = 0$ .



# Matrix-Kettenmultiplikation

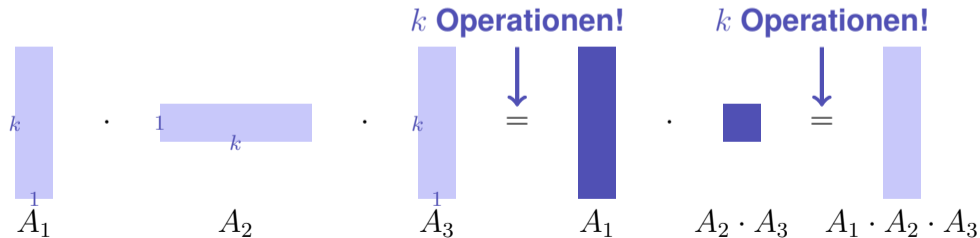
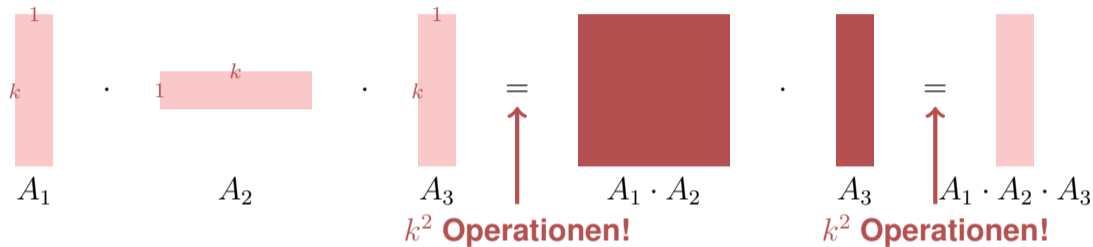
Aufgabe: Berechnung des Produktes  $A_1 \cdot A_2 \cdot \dots \cdot A_n$  von Matrizen  $A_1, \dots, A_n$ .

Matrizenmultiplikation ist assoziativ, d.h. Klammerung kann beliebig gewählt werden.

Ziel: möglichst effiziente Berechnung des Produktes.

Annahme: Multiplikation einer  $(r \times s)$ -Matrix mit einer  $(s \times u)$ -Matrix hat Kosten  $r \cdot s \cdot u$ .

# Macht das einen Unterschied?



# Rekursion

- Annahme, dass die bestmögliche Berechnung von  $(A_1 \cdot A_2 \cdots A_i)$  und  $(A_{i+1} \cdot A_{i+2} \cdots A_n)$  für jedes  $i$  bereits bekannt ist.
- Bestimme bestes  $i$ , fertig.

$n \times n$ -Tabelle  $M$ . Eintrag  $M[p, q]$  enthält Kosten der besten Klammerung von  $(A_p \cdot A_{p+1} \cdots A_q)$ .

$$M[p, q] \leftarrow \min_{p \leq i < q} (M[p, i] + M[i + 1, q] + \text{Kosten letzte Multiplikation})$$

# Berechnung der DP-Tabelle

- Randfälle:  $M[p, p] \leftarrow 0$  für alle  $1 \leq p \leq n$ .
- Berechnung von  $M[p, q]$  hängt ab von  $M[i, j]$  mit  $p \leq i \leq j \leq q$ ,  $(i, j) \neq (p, q)$ .  
Insbesondere hängt  $M[p, q]$  höchstens ab von Einträgen  $M[i, j]$  mit  $i - j < q - p$ .  
Folgerung: Fülle die Tabelle von der Diagonale ausgehend.

# Analyse

DP-Tabelle hat  $n^2$  Einträge. Berechnung eines Eintrages bedingt Betrachten von bis zu  $n - 1$  anderen Einträgen.

Gesamtlaufzeit  $\mathcal{O}(n^3)$ .

Auslesen der Reihenfolge aus  $M$ : Übung!

# Exkurs: Matrixmultiplikation

Betrachten Multiplikation zweier  $n \times n$ -Matrizen.

Seien

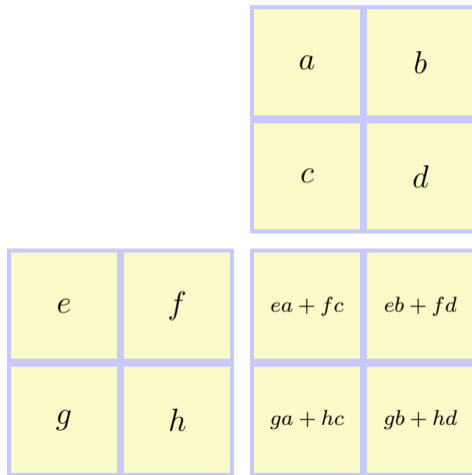
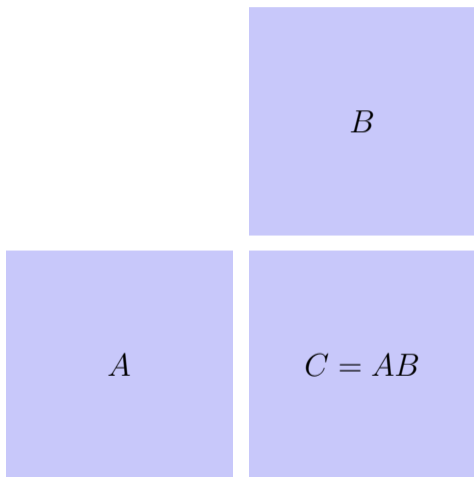
$$A = (a_{ij})_{1 \leq i, j \leq n}, B = (b_{ij})_{1 \leq i, j \leq n}, C = (c_{ij})_{1 \leq i, j \leq n}, \\ C = A \cdot B$$

dann

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}.$$

Naiver Algorithmus benötigt  $\Theta(n^3)$  elementare Multiplikationen.

# Divide and Conquer



# Divide and Conquer

- Annahme  $n = 2^k$ .
- Anzahl elementare Multiplikationen:  
 $M(n) = 8M(n/2)$ ,  $M(1) = 1$ .
- Ergibt  $M(n) = 8^{\log_2 n} = n^{\log_2 8} = n^3$ . Kein Gewinn 😞

		$a$	$b$
		$c$	$d$
$e$	$f$	$ea + fc$	$eb + fd$
$g$	$h$	$ga + hc$	$gb + hd$



# Strassens Matrixmultiplikation

- **Nichttriviale Beobachtung von Strassen (1969):** Es genügt die Berechnung der sieben Produkte  $A = (e + h) \cdot (a + d)$ ,  $B = (g + h) \cdot a$ ,  $C = e \cdot (b - d)$ ,  $D = h \cdot (c - a)$ ,  $E = (e + f) \cdot d$ ,  $F = (g - e) \cdot (a + b)$ ,  $G = (f - h) \cdot (c + d)$ . Denn:  
 $ea + fc = A + D - E + G$ ,  $eb + fd = C + E$ ,  
 $ga + hc = B + D$ ,  $gb + hd = A - B + C + F$ .
- **Damit ergibt sich**

$$M'(n) = 7M(n/2), M'(1) = 1.$$

$$\text{Also } M'(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}.$$

- **Schnellster bekannter Algorithmus:**

$$\mathcal{O}(n^{2.37})$$

		a	b
		c	d
e	f	ea + fc	eb + fd
g	h	ga + hc	gb + hd

## 20. Dynamic Programming II

Subset Sum Problem, Rucksackproblem, Greedy Algorithmus vs dynamische Programmierung [Ottman/Widmayer, Kap. 7.2, 7.3, 5.7, Cormen et al, Kap. 15,35.5]

# Aufgabe



Teile obige “Gegenstände” so auf zwei Mengen auf, dass beide Mengen den gleichen Wert haben.

Eine Lösung:



# Subset Sum Problem

Seien  $n \in \mathbb{N}$  Zahlen  $a_1, \dots, a_n \in \mathbb{N}$  gegeben.

Ziel: Entscheide, ob eine Auswahl  $I \subseteq \{1, \dots, n\}$  existiert mit

$$\sum_{i \in I} a_i = \sum_{i \in \{1, \dots, n\} \setminus I} a_i.$$

# Naiver Algorithmus

Prüfe für jeden Bitvektor  $b = (b_1, \dots, b_n) \in \{0, 1\}^n$ , ob

$$\sum_{i=1}^n b_i a_i \stackrel{?}{=} \sum_{i=1}^n (1 - b_i) a_i$$

Schlechtester Fall:  $n$  Schritte für jeden der  $2^n$  Bitvektoren  $b$ . Anzahl Schritte:  $\mathcal{O}(n \cdot 2^n)$ .

# Algorithmus mit Aufteilung

- Zerlege Eingabe in zwei gleich grosse Teile:  $a_1, \dots, a_{n/2}$  und  $a_{n/2+1}, \dots, a_n$ .
- Iteriere über alle Teilmengen der beiden Teile und berechne Teilsummen  $S_1^k, \dots, S_{2^{n/2}}^k$  ( $k = 1, 2$ ).
- Sortiere die Teilsummen:  $S_1^k \leq S_2^k \leq \dots \leq S_{2^{n/2}}^k$ .
- Prüfe ob es Teilsummen gibt, so dass  $S_i^1 + S_j^2 = \frac{1}{2} \sum_{i=1}^n a_i =: h$ 
  - Beginne mit  $i = 1, j = 2^{n/2}$ .
  - Gilt  $S_i^1 + S_j^2 = h$  dann fertig
  - Gilt  $S_i^1 + S_j^2 > h$  dann  $j \leftarrow j - 1$
  - Gilt  $S_i^1 + S_j^2 < h$  dann  $i \leftarrow i + 1$

# Beispiel

Menge  $\{1, 6, 2, 3, 4\}$  mit Wertesumme 16 hat 32 Teilmengen.

Aufteilung in  $\{1, 6\}$ ,  $\{2, 3, 4\}$  ergibt folgende 12 Teilmengen mit Wertesummen:

$\{1, 6\}$					$\{2, 3, 4\}$							
$\{\}$	$\{1\}$	$\{6\}$	$\{1, 6\}$		$\{\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{2, 3\}$	$\{2, 4\}$	$\{3, 4\}$	$\{2, 3, 4\}$
0	1	6	7		0	2	3	4	5	6	7	9

$\Leftrightarrow$  Eine Lösung:  $\{1, 3, 4\}$

# Analyse

- Teilsummegenerierung in jedem Teil:  $\mathcal{O}(2^{n/2} \cdot n)$ .
- Sortieren jeweils:  $\mathcal{O}(2^{n/2} \log(2^{n/2})) = \mathcal{O}(n2^{n/2})$ .
- Zusammenführen:  $\mathcal{O}(2^{n/2})$

Gesamtlaufzeit

$$\mathcal{O}\left(n \cdot 2^{n/2}\right) = \mathcal{O}\left(n \left(\sqrt{2}\right)^n\right).$$

Wesentliche Verbesserung gegenüber ganz naivem Verfahren –  
aber immer noch exponentiell!



# Dynamische Programmierung

**Aufgabe:** sei  $z = \frac{1}{2} \sum_{i=1}^n a_i$ . Suche Auswahl  $I \subset \{1, \dots, n\}$ , so dass  $\sum_{i \in I} a_i = z$ .

**DP-Tabelle:**  $[0, \dots, n] \times [0, \dots, z]$ -Tabelle  $T$  mit Wahrheitseinträgen.  $T[k, s]$  gibt an, ob es eine Auswahl  $I_k \subset \{1, \dots, k\}$  gibt, so dass  $\sum_{i \in I_k} a_i = s$ .

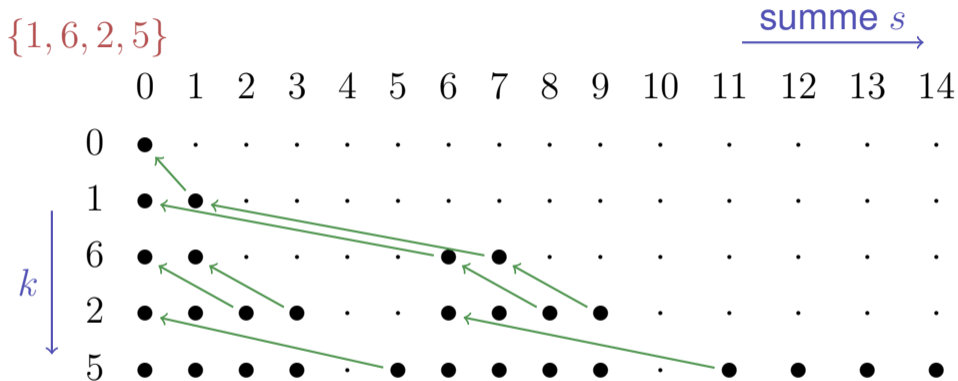
**Initialisierung:**  $T[0, 0] = \text{true}$ .  $T[0, s] = \text{false}$  für  $s > 0$ .

**Berechnung:**

$$T[k, s] \leftarrow \begin{cases} T[k-1, s] & \text{falls } s < a_k \\ T[k-1, s] \vee T[k-1, s - a_k] & \text{falls } s \geq a_k \end{cases}$$

für aufsteigende  $k$  und innerhalb  $k$  dann  $s$ .

# Beispiel



Auslesen der Lösung: wenn  $T[k, s] = T[k - 1, s]$  dann  $a_k$  nicht benutzt und bei  $T[k - 1, s]$  weiterfahren, andernfalls  $a_k$  benutzt und bei  $T[k - 1, s - a_k]$  weiterfahren.

# Rätselhaftes

Der Algorithmus benötigt  $\mathcal{O}(n \cdot z)$  Elementaroperationen.

Was ist denn jetzt los? Hat der Algorithmus plötzlich polynomielle Laufzeit?

# Aufgelöst

Der Algorithmus hat nicht unbedingt eine polynomielle Laufzeit.  $z$  ist eine *Zahl* und keine *Anzahl*!

Eingabelänge des Algorithmus  $\cong$  Anzahl Bits zur *vernünftigen* Repräsentation der Daten. Bei der Zahl  $z$  wäre das  $\zeta = \log z$ .

Also: Algorithmus benötigt  $\mathcal{O}(n \cdot 2^\zeta)$  Elementaroperationen und hat exponentielle Laufzeit in  $\zeta$ .

Sollte  $z$  allerdings polynomiell sein in  $n$ , dann hat der Algorithmus polynomielle Laufzeit in  $n$ . Das nennt man *pseudopolynomiell*.

# NP

Man weiss, dass der Subset-Sum Algorithmus zur Klasse der *NP*-vollständigen Probleme gehört (und somit *NP-schwer* ist).

*P*: Menge aller in Polynomialzeit lösbarer Probleme.

*NP*: Menge aller **N**ichtdeterministisch in **P**olynomialzeit lösbarer Probleme.

Implikationen:

- NP enthält P.
- Probleme in Polynomialzeit *verifizierbar*.
- Unter der (noch?) unbewiesenen<sup>43</sup> Annahme, dass  $NP \neq P$ , gibt es für das Problem *keinen Algorithmus mit polynomieller Laufzeit*.

<sup>43</sup>Die bedeutendste ungelöste Frage der theoretischen Informatik!

# Das Rucksackproblem

Wir packen unseren Koffer und nehmen mit ...

- |                       |                       |                       |
|-----------------------|-----------------------|-----------------------|
| ■ Zahnbürste          | ■ Zahnbürste          | ■ Zahnbürste          |
| ■ Hantelset           | ■ Luftballon          | ■ Kaffemaschine       |
| ■ Kaffemaschine       | ■ Taschenmesser       | ■ Taschenmesser       |
| ■ Oh jeh – zu schwer. | ■ Ausweis             | ■ Ausweis             |
|                       | ■ Hantelset           | ■ Oh jeh – zu schwer. |
|                       | ■ Oh jeh – zu schwer. |                       |

Wollen möglichst viel mitnehmen. Manche Dinge sind uns aber wichtiger als andere.

# Rucksackproblem (engl. Knapsack problem)

## Gegeben:

- Menge von  $n \in \mathbb{N}$  Gegenständen  $\{1, \dots, n\}$ .
- Jeder Gegenstand  $i$  hat Nutzwert  $v_i \in \mathbb{N}$  und Gewicht  $w_i \in \mathbb{N}$ .
- Maximalgewicht  $W \in \mathbb{N}$ .
- Bezeichnen die Eingabe mit  $E = (v_i, w_i)_{i=1, \dots, n}$ .

## Gesucht:

eine Auswahl  $I \subseteq \{1, \dots, n\}$  die  $\sum_{i \in I} v_i$  maximiert unter  $\sum_{i \in I} w_i \leq W$ .

# Gierige (engl. greedy) Heuristik

Sortiere die Gegenstände absteigend nach Nutzen pro Gewicht  
 $v_i/w_i$ : Permutation  $p$  mit  $v_{p_i}/w_{p_i} \geq v_{p_{i+1}}/w_{p_{i+1}}$

Füge Gegenstände in dieser Reihenfolge hinzu ( $I \leftarrow I \cup \{p_i\}$ ),  
sofern das zulässige Gesamtgewicht dadurch nicht überschritten  
wird.

Das ist schnell:  $\Theta(n \log n)$  für Sortieren und  $\Theta(n)$  für die Auswahl.  
Aber ist es auch gut?



# Gegenbeispiel zur greedy strategy

$$v_1 = 1 \quad w_1 = 1 \quad v_1/w_1 = 1$$

$$v_2 = W - 1 \quad w_2 = W \quad v_2/w_2 = \frac{W-1}{W}$$

Greedy Algorithmus wählt  $\{v_1\}$  mit Nutzwert 1.

Beste Auswahl:  $\{v_2\}$  mit Nutzwert  $W - 1$  und Gewicht  $W$ .

Greedy *kann* also beliebig schlecht sein.

# Dynamic Programming

Unterteile das Maximalgewicht.

Dreidimensionale Tabelle  $m[i, w, v]$  (“machbar”) aus Wahrheitswerten.

$m[i, w, v] = \text{true}$  genau dann wenn

- Auswahl der ersten  $i$  Teile existiert ( $0 \leq i \leq n$ )
- deren Gesamtgewicht höchstens  $w$  ( $0 \leq w \leq W$ ) und
- Nutzen mindestens  $v$  ( $0 \leq v \leq \sum_{i=1}^n v_i$ ) ist.

# Berechnung der DP Tabelle

## Initial

- $m[i, w, 0] \leftarrow \text{true}$  für alle  $i \geq 0$  und alle  $w \geq 0$ .
- $m[0, w, v] \leftarrow \text{false}$  für alle  $w \geq 0$  und alle  $v > 0$ .

## Berechnung

$$m[i, w, v] \leftarrow \begin{cases} m[i-1, w, v] \vee m[i-1, w-w_i, v-v_i] & \text{falls } w \geq w_i \text{ und } v \geq v_i \\ m[i-1, w, v] & \text{sonst.} \end{cases}$$

aufsteigend nach  $i$  und für festes  $i$  aufsteigend nach  $w$  und für festes  $i$  und  $w$  aufsteigend nach  $v$ .

Lösung: Grösstes  $v$ , so dass  $m[i, w, v] = \text{true}$  für ein  $i$  und  $w$ .

# Beobachtung

Nach der Definition des Problems gilt offensichtlich, dass

- für  $m[i, w, v] = \text{true}$  gilt:
  - $m[i', w, v] = \text{true} \quad \forall i' \geq i$ ,
  - $m[i, w', v] = \text{true} \quad \forall w' \geq w$ ,
  - $m[i, w, v'] = \text{true} \quad \forall v' \leq v$ .
- für  $m[i, w, v] = \text{false}$  gilt:
  - $m[i', w, v] = \text{false} \quad \forall i' \leq i$ ,
  - $m[i, w', v] = \text{false} \quad \forall w' \leq w$ ,
  - $m[i, w, v'] = \text{false} \quad \forall v' \geq v$ .

Das ist ein starker Hinweis darauf, dass wir keine 3d-Tabelle benötigen.

# DP Tabelle mit 2 Dimensionen

Tabelleneintrag  $t[i, w]$  enthält statt Wahrheitswerten das jeweils grösste  $v$ , das erreichbar ist<sup>44</sup> mit

- den Gegenständen  $1, \dots, i$  ( $0 \leq i \leq n$ )
- bei höchstem zulässigem Gewicht  $w$  ( $0 \leq w \leq W$ ).

---

<sup>44</sup>So etwas ähnliches hätten wir beim Subset Sum Problem auch machen können, um die dünnbesetzte Tabelle etwas zu verkleinern

# Berechnung

Initial

- $t[0, w] \leftarrow 0$  für alle  $w \geq 0$ .

Berechnung

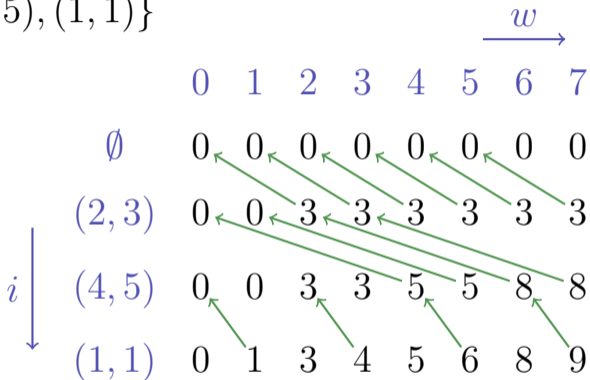
$$t[i, w] \leftarrow \begin{cases} t[i-1, w] & \text{falls } w < w_i \\ \max\{t[i-1, w], t[i-1, w - w_i] + v_i\} & \text{sonst.} \end{cases}$$

aufsteigend nach  $i$  und für festes  $i$  aufsteigend nach  $w$ .

Lösung steht in  $t[n, w]$

# Beispiel

$$E = \{(2, 3), (4, 5), (1, 1)\}$$



Auslesen der Lösung: wenn  $t[i, w] = t[i - 1, w]$  dann Gegenstand  $i$  nicht benutzt und bei  $t[i - 1, w]$  weiterfahren, andernfalls benutzt und bei  $t[i - 1, s - w_i]$  weiterfahren.

# Analyse

Die beiden Algorithmen für das Rucksackproblem haben eine Laufzeit in  $\Theta(n \cdot W \cdot \sum_{i=1}^n v_i)$  (3d-Tabelle) und  $\Theta(n \cdot W)$  (2d-Tabelle) und sind beide damit pseudopolynomiell, liefern aber das bestmögliche Resultat.

Der greedy Algorithmus ist sehr schnell, liefert aber unter Umständen beliebig schlechte Resultate.

Im folgenden beschäftigen wir uns mit einer Lösung dazwischen.



# 21. Dynamic Programming II

FPTAS [Ottman/Widmayer, Kap. 7.2, 7.3, Cormen et al, Kap. 15,35.5]

# Approximation

Sei ein  $\varepsilon \in (0, 1)$  gegeben. Sei  $I_{\text{opt}}$  eine bestmögliche Auswahl.  
Suchen eine gültige Auswahl  $I$  mit

$$\sum_{i \in I} v_i \geq (1 - \varepsilon) \sum_{i \in I_{\text{opt}}} v_i.$$

Summe der Gewichte darf  $W$  natürlich in keinem Fall überschreiten.

# Andere Formulierung des Algorithmus

**Bisher:** Gewichtsschranke  $w \rightarrow$  maximaler Nutzen  $v$

Umkehrung Nutzen  $v \rightarrow$  minimales Gewicht  $w$

$\Rightarrow$  **Alternative Tabelle:**  $g[i, v]$  gibt das minimale Gewicht an, welches

- eine Auswahl der ersten  $i$  Gegenstände ( $0 \leq i \leq n$ ) hat, die
- einen Nutzen von genau  $v$  aufweist ( $0 \leq v \leq \sum_{i=1}^n v_i$ ).

# Berechnung

## Initial

- $g[0, 0] \leftarrow 0$
- $g[0, v] \leftarrow \infty$  (Nutzen  $v$  kann mit 0 Gegenständen nie erreicht werden.).

## Berechnung

$$g[i, v] \leftarrow \begin{cases} g[i-1, v] & \text{falls } v < v_i \\ \min\{g[i-1, v], g[i-1, v-v_i] + w_i\} & \text{sonst.} \end{cases}$$

aufsteigend nach  $i$  und für festes  $i$  aufsteigend nach  $v$ .

Lösung ist der grösste Index  $v$  mit  $g[n, v] \leq w$ .

# Beispiel

$$E = \{(2, 3), (4, 5), (1, 1)\}$$

		$v \rightarrow$									
		0	1	2	3	4	5	6	7	8	9
$i \downarrow$	$\emptyset$	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
	(2, 3)	0	$\infty$	$\infty$	2	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
	(4, 5)	0	$\infty$	$\infty$	2	$\infty$	4	$\infty$	$\infty$	6	$\infty$
	(1, 1)	0	1	$\infty$	2	3	4	5	$\infty$	6	7

Auslesen der Lösung: wenn  $g[i, v] = g[i - 1, v]$  dann Gegenstand  $i$  nicht benutzt und bei  $g[i - 1, v]$  weiterfahren, andernfalls benutzt und bei  $g[i - 1, v - w_i]$  weiterfahren.

# Der Approximationstrick

Pseudopolynomielle Laufzeit wird polynomiell, wenn vorkommenden Werte in Polynom der Eingabelänge beschränkt werden können.

Sei  $K > 0$  *geeignet* gewählt. Ersetze die Nutzwerte  $v_i$  durch “gerundete Werte”  $\tilde{v}_i = \lfloor v_i/K \rfloor$  und erhalte eine neue Eingabe  $E' = (w_i, \tilde{v}_i)_{i=1\dots n}$ .

Wenden nun den Algorithmus auf Eingabe  $E'$  mit derselben Gewichtsschranke  $W$  an.

# Idee

**Beispiel**  $K = 5$

Eingabe Nutzwerte

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ..., 98, 99, 100

→

0, 0, 0, 0, 1, 1, 1, 1, 1, 2, ..., 19, 19, 20

Offensichtlich weniger unterschiedliche Nutzwerte

# Eigenschaften des neuen Algorithmus

- Auswahl von Gegenständen aus  $E'$  ist genauso gültig wie die aus  $E$ . Gewicht unverändert!
- Laufzeit des Algorithmus ist beschränkt durch  $\mathcal{O}(n^2 \cdot v_{\max}/K)$   
( $v_{\max} := \max\{v_i | 1 \leq i \leq n\}$ )



# Wie gut ist die Approximation?

Es gilt

$$v_i - K \leq K \cdot \left\lfloor \frac{v_i}{K} \right\rfloor = K \cdot \tilde{v}_i \leq v_i$$

Sei  $I'_{opt}$  eine optimale Lösung von  $E'$ . Damit

$$\begin{aligned} \left( \sum_{i \in I_{opt}} v_i \right) - n \cdot K &\stackrel{|I_{opt}| \leq n}{\leq} \sum_{i \in I_{opt}} (v_i - K) \leq \sum_{i \in I_{opt}} (K \cdot \tilde{v}_i) = K \sum_{i \in I_{opt}} \tilde{v}_i \\ &\stackrel{I'_{opt} \text{ optimal}}{\leq} K \sum_{i \in I'_{opt}} \tilde{v}_i = \sum_{i \in I'_{opt}} K \cdot \tilde{v}_i \leq \sum_{i \in I'_{opt}} v_i. \end{aligned}$$

# Wahl von $K$

Forderung:

$$\sum_{i \in I'} v_i \geq (1 - \varepsilon) \sum_{i \in I_{\text{opt}}} v_i.$$

Ungleichung von oben:

$$\sum_{i \in I'_{\text{opt}}} v_i \geq \left( \sum_{i \in I_{\text{opt}}} v_i \right) - n \cdot K$$

Also:  $K = \varepsilon \frac{\sum_{i \in I_{\text{opt}}} v_i}{n}.$

# Wahl von $K$

Wähle  $K = \varepsilon \frac{\sum_{i \in I_{\text{opt}}} v_i}{n}$ . Die optimale Summe ist aber unbekannt, daher wählen wir  $K' = \varepsilon \frac{v_{\text{max}}}{n}$ .<sup>45</sup>

Es gilt  $v_{\text{max}} \leq \sum_{i \in I_{\text{opt}}} v_i$  und somit  $K' \leq K$  und die Approximation ist sogar etwas besser.

Die Laufzeit des Algorithmus ist beschränkt durch

$$\mathcal{O}(n^2 \cdot v_{\text{max}}/K') = \mathcal{O}(n^2 \cdot v_{\text{max}}/(\varepsilon \cdot v_{\text{max}}/n)) = \mathcal{O}(n^3/\varepsilon).$$

---

<sup>45</sup>Wir können annehmen, dass vorgängig alle Gegenstände  $i$  mit  $w_i > W$  entfernt wurden.

# FPTAS

Solche Familie von Algorithmen nennt man *Approximationsschema*: die Wahl von  $\varepsilon$  steuert Laufzeit und Approximationsgüte.

Die Laufzeit  $\mathcal{O}(n^3/\varepsilon)$  ist ein Polynom in  $n$  und in  $\frac{1}{\varepsilon}$ . Daher nennt man das Verfahren auch ein voll polynomielles Approximationsschema  
*FPTAS - Fully Polynomial Time Approximation Scheme*

# 21. Dynamic Programming II

Optimale Suchbäume [Ottman/Widmayer, Kap. 5.7]

# Optimale binäre Suchbäume

Gegeben: Suchwahrscheinlichkeiten  $p_i$  zu jedem Schlüssel  $k_i$  ( $i = 1, \dots, n$ ) und  $q_i$  zu jedem Intervall  $d_i$  ( $i = 0, \dots, n$ ) zwischen Suchschlüsseln eines binären Suchbaumes.  $\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$ .

Gesucht: Optimaler Suchbaum  $T$  mit Schlüsseltiefen  $\text{depth}(\cdot)$ , welcher die erwarteten Suchkosten

$$\begin{aligned} C(T) &= \sum_{i=1}^n p_i \cdot (\text{depth}(k_i) + 1) + \sum_{i=0}^n q_i \cdot (\text{depth}(d_i) + 1) \\ &= 1 + \sum_{i=1}^n p_i \cdot \text{depth}(k_i) + \sum_{i=0}^n q_i \cdot \text{depth}(d_i) \end{aligned}$$

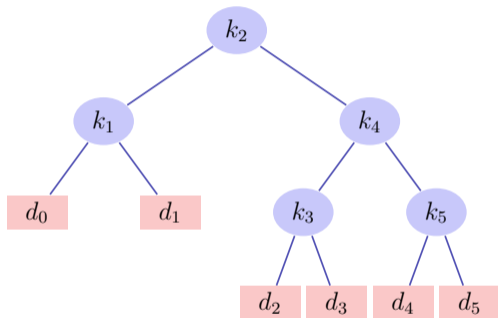
minimiert.

# Beispiel

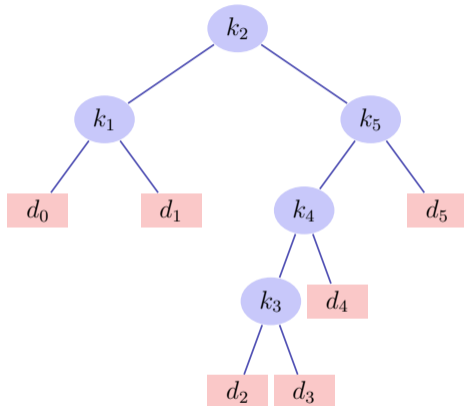
## Erwartete Häufigkeiten

$i$	0	1	2	3	4	5
$p_i$		0.15	0.10	0.05	0.10	0.20
$q_i$	0.05	0.10	0.05	0.05	0.05	0.10

# Beispiel



Suchbaum mit erwarteten  
Kosten 2.8



Suchbaum mit erwarteten  
Kosten 2.75



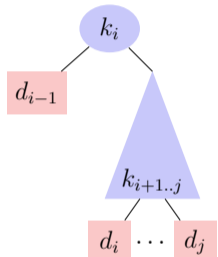
# Struktur eines optimalen Suchbaumes

- Teilsuchbaum mit Schlüsseln  $k_i, \dots, k_j$  und Intervallschlüsseln  $d_{i-1}, \dots, d_j$  muss für das entsprechende Teilproblem optimal sein.<sup>46</sup>
- Betrachten aller Teilsuchbäume mit Wurzel  $k_r, i \leq r \leq j$  und optimalen Teilbäumen  $k_i, \dots, k_{r-1}$  und  $k_{r+1}, \dots, k_j$

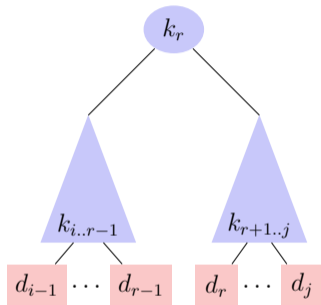
---

<sup>46</sup>Das übliche Argument: wäre er nicht optimal, könnte er durch eine bessere Lösung ersetzt werden, welche die Gesamtlösung verbessert.

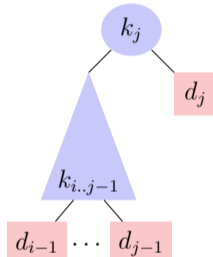
# Teilsuchbäume



leerer linker  
Teilsuchbaum



Teilsuchbäume links  
und rechts nichtleer



leerer rechter  
Teilsuchbaum

# Erwartete Suchkosten

Sei  $\text{depth}_T(k)$  die Tiefe des Knotens im Teilbaum  $T$ . Sei  $k_r$  die Wurzel eines Teilbaumes  $T_r$  und  $T_{L_r}$  und  $T_{R_r}$  der linke und rechte Teilbaum von  $T_r$ . Dann

$$\text{depth}_T(k_i) = \text{depth}_{T_{L_r}}(k_i) + 1, \quad (i < r)$$

$$\text{depth}_T(k_i) = \text{depth}_{T_{R_r}}(k_i) + 1, \quad (i > r)$$

# Erwartete Suchkosten

Seien  $e[i, j]$  die Kosten eines optimalen Suchbaumes mit Knoten  $k_i, \dots, k_j$ .

Basisfall:  $e[i, i - 1]$ , erwartete Suchkosten  $d_{i-1}$

Sei  $w(i, j) = \sum_{l=i}^j p_l + \sum_{l=i-1}^j q_l$ .

Wenn  $k_r$  die Wurzel eines optimalen Teilbaumes mit Schlüsseln  $k_i, \dots, k_j$ , dann

$$e[i, j] = p_r + (e[i, r - 1] + w(i, r - 1)) + (e[r + 1, j] + w(r + 1, j))$$

mit  $w(i, j) = w(i, r - 1) + p_r + w(r + 1, j)$ :

$$e[i, j] = e[i, r - 1] + e[r + 1, j] + w(i, j).$$

# Dynamic Programming

$$e[i, j] = \begin{cases} q_{i-1} & \text{falls } j = i - 1, \\ \min_{i \leq r \leq j} \{e[i, r - 1] + e[r + 1, j] + w[i, j]\} & \text{falls } i \leq j \end{cases}$$

# Berechnung

Tabellen  $e[1 \dots n + 1, 0 \dots n]$ ,  $w[1 \dots n + 1, 0 \dots m]$ ,  $r[1 \dots n, 1 \dots n]$

Initial

■  $e[i, i - 1] \leftarrow q_{i-1}$ ,  $w[i, i - 1] \leftarrow q_{i-1}$  für alle  $1 \leq i \leq n + 1$ .

Berechnung

$$w[i, j] = w[i, j - 1] + p_j + q_j$$

$$e[i, j] = \min_{i \leq r \leq j} \{e[i, r - 1] + e[r + 1, j] + w[i, j]\}$$

$$r[i, j] = \arg \min_{i \leq r \leq j} \{e[i, r - 1] + e[r + 1, j] + w[i, j]\}$$

für Intervalle  $[i, j]$  mit ansteigenden Längen  $l = 1, \dots, n$ , jeweils für  $i = 1, \dots, n - l + 1$ . Resultat steht in  $e[1, n]$ , Rekonstruktion via  $r$ .

Laufzeit  $\Theta(n^3)$ .

# Beispiel

$i$	0	1	2	3	4	5
$p_i$		0.15	0.10	0.05	0.10	0.20
$q_i$	0.05	0.10	0.05	0.05	0.05	0.10

$w$

$j$							$i$
0	0.05						
1	0.30	0.10					
2	0.45	0.25	0.05				
3	0.55	0.35	0.15	0.05			
4	0.70	0.50	0.30	0.20	0.05		
5	1.00	0.80	0.60	0.50	0.35	0.10	

$e$

$j$							$i$
0	0.05						
1	0.45	0.10					
2	0.90	0.40	0.05				
3	1.25	0.70	0.25	0.05			
4	1.75	1.20	0.60	0.30	0.05		
5	2.75	2.00	1.30	0.90	0.50	0.10	

$r$

$j$						$i$
1	1					
2	1	2				
3	2	2	3			
4	2	2	4	4		
5	2	4	5	5	5	

## 22. Gierige (Greedy) Algorithmen

Gebrochenes Rucksack Problem, Huffman Coding [Cormen et al, Kap. 16.1, 16.3]



# Das Gebrochene Rucksackproblem

Menge von  $n \in \mathbb{N}$  Gegenständen  $\{1, \dots, n\}$  gegeben. Jeder Gegenstand  $i$  hat Nutzwert  $v_i \in \mathbb{N}$  und Gewicht  $w_i \in \mathbb{N}$ . Das Maximalgewicht ist gegeben als  $W \in \mathbb{N}$ . Bezeichnen die Eingabe mit  $E = (v_i, w_i)_{i=1, \dots, n}$ .

**Gesucht:** Anteile  $0 \leq q_i \leq 1$  ( $1 \leq i \leq n$ ) die die Summe  $\sum_{i=1}^n q_i \cdot v_i$  maximieren unter  $\sum_{i=1}^n q_i \cdot w_i \leq W$ .

# Gierige (Greedy) Heuristik

Sortiere die Gegenstände absteigend nach Nutzen pro Gewicht  $v_i/w_i$ .

Annahme  $v_i/w_i \geq v_{i+1}/w_{i+1}$

Sei  $j = \max\{0 \leq k \leq n : \sum_{i=1}^k w_i \leq W\}$ . Setze

- $q_i = 1$  für alle  $1 \leq i \leq j$ .
- $q_{j+1} = \frac{W - \sum_{i=1}^j w_i}{w_{j+1}}$ .
- $q_i = 0$  für alle  $i > j + 1$ .

Das ist schnell:  $\Theta(n \log n)$  für Sortieren und  $\Theta(n)$  für die Berechnung der  $q_i$ .

# Korrektheit

Annahme: Optimale Lösung  $(r_i)$  ( $1 \leq i \leq n$ ).

Der Rucksack wird immer ganz gefüllt:  $\sum_i r_i \cdot w_i = \sum_i q_i \cdot w_i = W$ .

Betrachte  $k$ : kleinstes  $i$  mit  $r_i \neq q_i$ . Die gierige Heuristik nimmt per Definition so viel wie möglich:  $q_k > r_k$ . Sei  $x = q_k - r_k > 0$ .

Konstruiere eine neue Lösung  $(r'_i)$ :  $r'_i = r_i \forall i < k$ .  $r'_k = q_k$ . Entferne Gewicht  $\sum_{i=k+1}^n \delta_i = x \cdot w_k$  von den Gegenständen  $k + 1$  bis  $n$ . Das geht, denn  $\sum_{i=k}^n r_i \cdot w_i = \sum_{i=k}^n q_i \cdot w_i$ .

# Korrektheit

$$\begin{aligned}\sum_{i=k}^n r'_i v_i &= r_k v_k + x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^n (r_i w_i - \delta_i) \frac{v_i}{w_i} \\ &\geq r_k v_k + x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^n r_i w_i \frac{v_i}{w_i} - \delta_i \frac{v_k}{w_k} \\ &= r_k v_k + x w_k \frac{v_k}{w_k} - x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^n r_i w_i \frac{v_i}{w_i} = \sum_{i=k}^n r_i v_i.\end{aligned}$$

Also ist  $(r'_i)$  auch optimal. Iterative Anwendung dieser Idee erzeugt die Lösung  $(q_i)$ .

# Huffman-Codierungen

Ziel: Speicherplatzeffizientes Speichern einer Folge von Zeichen mit einem binären *Zeichencode* aus *Codewörtern*.

## Beispiel

File aus 100.000 Buchstaben aus dem Alphabet  $\{a, \dots, f\}$

	a	b	c	d	e	f
Häufigkeit (Tausend)	45	13	12	16	9	5
Codewort fester Länge	000	001	010	011	100	101
Codewort variabler Länge	0	101	100	111	1101	1100

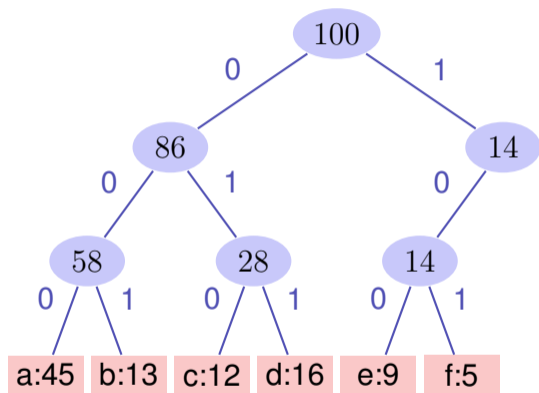
Speichergrösse (Code fixe Länge): 300.000 bits.

Speichergrösse (Code variabler Länge): 224.000 bits.

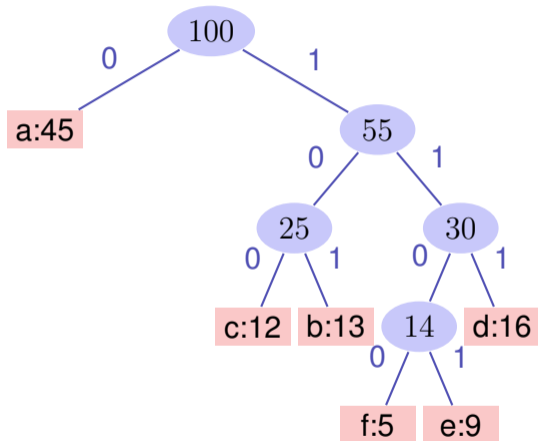
# Huffman-Codierungen

- Betrachten *Präfixcodes*: kein Codewort kann mit einem anderen Codewort beginnen.
- Präfixcodes können im Vergleich mit allen Codes die optimale *Datenkompression* erreichen (hier ohne Beweis).
- Codierung: Verkettung der Codewörter ohne Zwischenzeichen (Unterschied zum Morsen!)  
 $af fe \rightarrow 0 \cdot 1100 \cdot 1100 \cdot 1101 \rightarrow 0110011001101$
- Decodierung einfach da Präfixcode  
 $0110011001101 \rightarrow 0 \cdot 1100 \cdot 1100 \cdot 1101 \rightarrow af fe$

# Codebäume



Codewörter fixer Länge



Codewörter variabler Länge

# Eigenschaften der Codebäume

- Optimale Codierung eines Files wird immer durch vollständigen binären Baum dargestellt: jeder innere Knoten hat zwei Kinder.
- Sei  $C$  die Menge der Codewörter,  $f(c)$  die Häufigkeit eines Codeworts  $c$  und  $d_T(c)$  die Tiefe eines Wortes im Baum  $T$ . Definieren die **Kosten** eines Baumes als

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c).$$

(Kosten = Anzahl Bits des codierten Files)

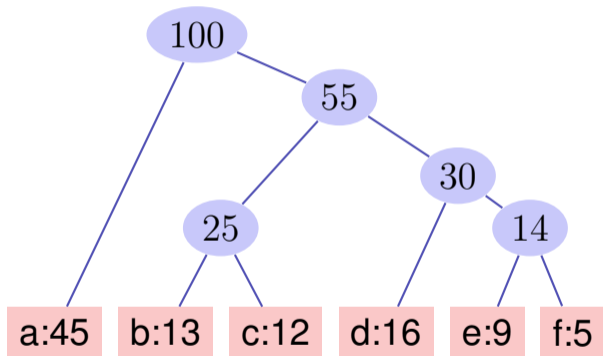
Bezeichnen im folgenden einen Codebaum als optimal, wenn er die Kosten minimiert.



# Algorithmus Idee

Baum Konstruktion von unten nach oben

- Starte mit der Menge  $C$  der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



# Algorithmus Huffman( $C$ )

**Input:** Codewörter  $c \in C$

**Output:** Wurzel eines optimalen Codebaums

$n \leftarrow |C|$

$Q \leftarrow C$

**for**  $i = 1$  **to**  $n - 1$  **do**

    Alloziere neuen Knoten  $z$

$z.\text{left} \leftarrow \text{ExtractMin}(Q)$       // Extrahiere Wort mit minimaler Häufigkeit.

$z.\text{right} \leftarrow \text{ExtractMin}(Q)$

$z.\text{freq} \leftarrow z.\text{left}.\text{freq} + z.\text{right}.\text{freq}$

    Insert( $Q, z$ )

**return** ExtractMin( $Q$ )

# Analyse

Verwendung eines Heaps: Heap bauen in  $\mathcal{O}(n)$ . Extract-Min in  $\mathcal{O}(\log n)$  für  $n$  Elemente. Somit Laufzeit  $\mathcal{O}(n \log n)$ .

# Das gierige Verfahren ist korrekt

## Theorem

*Seien  $x, y$  zwei Symbole mit kleinsten Frequenzen in  $C$  und sei  $T'(C')$  der optimale Baum zum Alphabet  $C' = C - \{x, y\} + \{z\}$  mit neuem Symbol  $z$  mit  $f(z) = f(x) + f(y)$ . Dann ist der Baum  $T(C)$  der aus  $T'(C')$  entsteht, indem der Knoten  $z$  durch einen inneren Knoten mit Kindern  $x$  und  $y$  ersetzt wird, ein optimaler Codebaum zum Alphabet  $C$ .*

# Beweis

Es gilt  $f(x) \cdot d_T(x) + f(y) \cdot d_T(y) = (f(x) + f(y)) \cdot (d_{T'}(z) + 1) = f(z) \cdot d_{T'}(x) + f(x) + f(y)$ . Also  $B(T') = B(T) - f(x) - f(y)$ .

Annahme:  $T$  sei nicht optimal. Dann existiert ein optimaler Baum  $T''$  mit  $B(T'') < B(T)$ . Annahme:  $x$  und  $y$  Brüder in  $T''$ .  $T'''$  sei der Baum  $T''$  in dem der innere Knoten mit Kindern  $x$  und  $y$  gegen  $z$  getauscht wird. Dann gilt

$$B(T''') = B(T'') - f(x) - f(y) < B(T) - f(x) - f(y) = B(T').$$

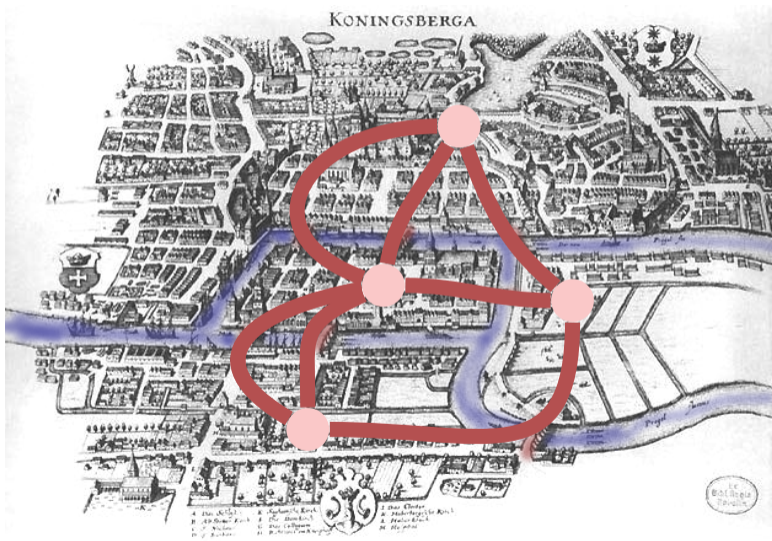
Widerspruch zur Optimalität von  $T'$ .

Die Annahme, dass  $x$  und  $y$  Brüder sind in  $T''$  kann man rechtfertigen, da ein Tausch der Elemente mit kleinster Häufigkeit auf die unterste Ebene den Wert von  $B$  höchstens verkleinern kann.

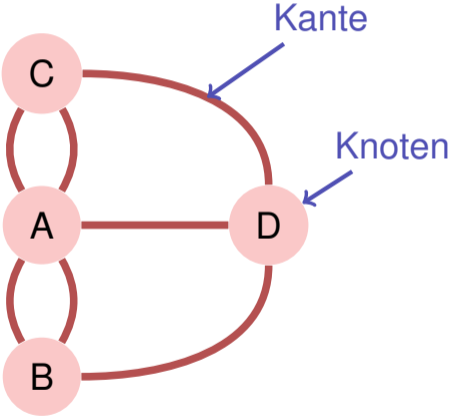
# 23. Graphen

Notation, Repräsentation, Traversieren (DFS, BFS), Topologisches Sortieren , Reflexive transitive Hülle, Zusammenhangskomponenten [Ottman/Widmayer, Kap. 9.1 - 9.4, Cormen et al, Kap. 22]

# Königsberg 1736



# [Multi]Graph

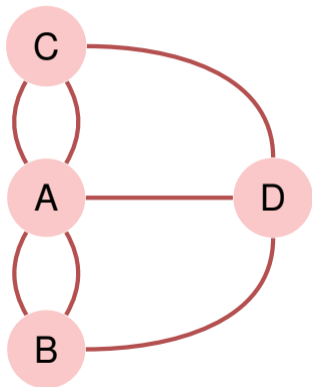




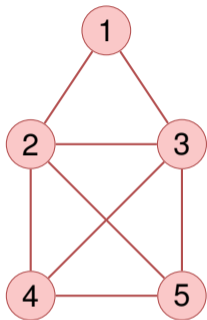
# Zyklen

- Gibt es einen Rundweg durch die Stadt (den Graphen), welcher jede Brücke (jede Kante) genau einmal benutzt?
- Euler (1736): nein.
- Solcher Rundweg (*Zyklus*) heisst *Eulerscher Kreis*.
- Eulerzyklus  $\Leftrightarrow$  jeder Knoten hat gerade Anzahl Kanten (jeder Knoten hat einen *geraden Grad*).

“ $\Rightarrow$ ” ist sofort klar, “ $\Leftarrow$ ” ist etwas schwieriger, aber auch elementar.



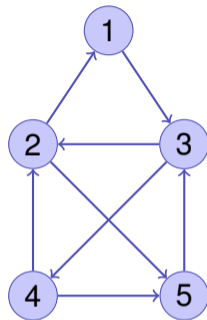
# Notation



ungerichtet

$$V = \{1, 2, 3, 4, 5\}$$

$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \\ \{2, 5\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}$$



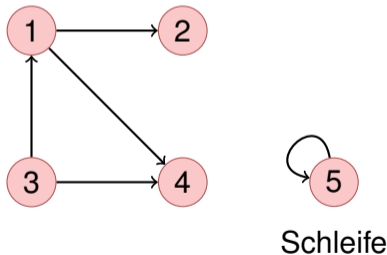
gerichtet

$$V = \{1, 2, 3, 4, 5\}$$

$$E = \{(1, 3), (2, 1), (2, 5), (3, 2), \\ (3, 4), (4, 2), (4, 5), (5, 3)\}$$

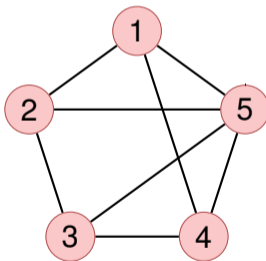
# Notation

Ein *gerichteter Graph* besteht aus einer Menge  $V = \{v_1, \dots, v_n\}$  von Knoten (*Vertices*) und einer Menge  $E \subseteq V \times V$  von Kanten (*Edges*). Gleiche Kanten dürfen nicht mehrfach enthalten sein.



# Notation

Ein *ungerichteter Graph* besteht aus einer Menge  $V = \{v_1, \dots, v_n\}$  von Knoten und einer Menge  $E \subseteq \{\{u, v\} | u, v \in V\}$  von Kanten. Kanten dürfen nicht mehrfach enthalten sein.<sup>47</sup>



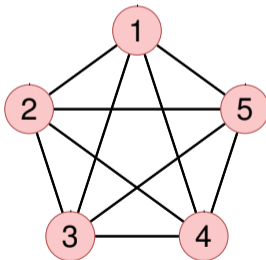
ungerichteter Graph

---

<sup>47</sup>Im Gegensatz zum Eingangsbeispiel – dann Multigraph genannt.

# Notation

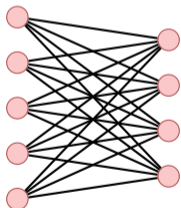
Ein ungerichteter Graph  $G = (V, E)$  ohne Schleifen in dem jeder Knoten mit jedem anderen Knoten durch eine Kante verbunden ist, heisst *vollständig*.



ein vollständiger ungerichteter Graph

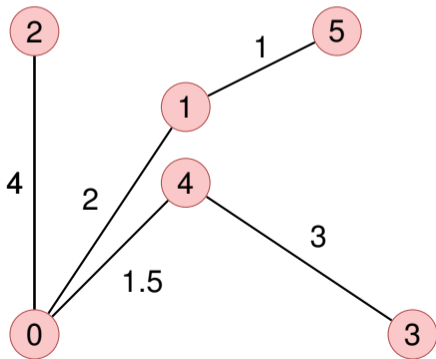
# Notation

Ein Graph, bei dem  $V$  so in disjunkte  $U$  und  $W$  aufgeteilt werden kann, dass alle  $e \in E$  einen Knoten in  $U$  und einen in  $W$  haben heisst *bipartit*.



# Notation

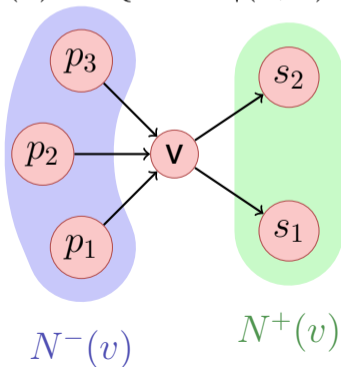
Ein *gewichteter Graph*  $G = (V, E, c)$  ist ein Graph  $G = (V, E)$  mit einer *Kantengewichtsfunktion*  $c : E \rightarrow \mathbb{R}$ .  $c(e)$  heisst *Gewicht* der Kante  $e$ .



# Notation

Für gerichtete Graphen  $G = (V, E)$

- $w \in V$  heisst *adjazent* zu  $v \in V$ , falls  $(v, w) \in E$
- *Vorgängermenge* von  $v \in V$ :  $N^-(v) := \{u \in V \mid (u, v) \in E\}$ .  
*Nachfolgermenge*:  $N^+(v) := \{u \in V \mid (v, u) \in E\}$

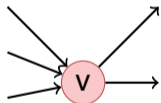




# Notation

Für gerichtete Graphen  $G = (V, E)$

- **Eingangsgrad:**  $\deg^-(v) = |N^-(v)|$ ,  
**Ausgangsgrad:**  $\deg^+(v) = |N^+(v)|$



$$\deg^-(v) = 3, \deg^+(v) = 2$$

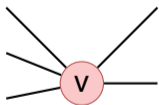


$$\deg^-(w) = 1, \deg^+(w) = 1$$

# Notation

Für ungerichtete Graphen  $G = (V, E)$ :

- $w \in V$  heisst **adjazent** zu  $v \in V$ , falls  $\{v, w\} \in E$
- **Nachbarschaft** von  $v \in V$ :  $N(v) = \{w \in V \mid \{v, w\} \in E\}$
- **Grad** von  $v$ :  $\deg(v) = |N(v)|$  mit Spezialfall Schleifen: erhöhen Grad um 2.



$$\deg(v) = 5$$



$$\deg(w) = 2$$

# Beziehung zwischen Knotengraden und Kantenzahl

In jedem Graphen  $G = (V, E)$  gilt

- 1  $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$ , falls  $G$  gerichtet
- 2  $\sum_{v \in V} \deg(v) = 2|E|$ , falls  $G$  ungerichtet.

# Wege

- **Weg**: Sequenz von Knoten  $\langle v_1, \dots, v_{k+1} \rangle$  so dass für jedes  $i \in \{1 \dots k\}$  eine Kante von  $v_i$  nach  $v_{i+1}$  existiert.
- **Länge** des Weges: Anzahl enthaltene Kanten  $k$ .
- **Gewicht** des Weges (in gewichteten Graphen):  $\sum_{i=1}^k c((v_i, v_{i+1}))$   
(bzw.  $\sum_{i=1}^k c(\{v_i, v_{i+1}\})$ )
- **Pfad** (auch: einfacher Pfad): Weg der keinen Knoten mehrfach verwendet.

# Zusammenhang

- Ungerichteter Graph heisst *zusammenhängend*, wenn für jedes Paar  $v, w \in V$  ein verbindender Weg existiert.
- Gerichteter Graph heisst *stark zusammenhängend*, wenn für jedes Paar  $v, w \in V$  ein verbindender Weg existiert.
- Gerichteter Graph heisst *schwach zusammenhängend*, wenn der entsprechende ungerichtete Graph zusammenhängend ist.

# Einfache Beobachtungen

- Allgemein:  $0 \leq |E| \in \mathcal{O}(|V|^2)$
- Zusammenhängender Graph:  $|E| \in \Omega(|V|)$
- Vollständiger Graph:  $|E| = \frac{|V| \cdot (|V| - 1)}{2}$  (ungerichtet)
- Maximal  $|E| = |V|^2$  (gerichtet),  $|E| = \frac{|V| \cdot (|V| + 1)}{2}$  (ungerichtet)

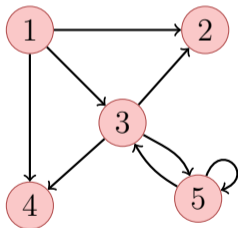
# Zyklen

- **Zyklus**: Weg  $\langle v_1, \dots, v_{k+1} \rangle$  mit  $v_1 = v_{k+1}$
- **Kreis**: Zyklus mit paarweise verschiedenen  $v_1, \dots, v_k$ , welcher keine Kante mehrfach verwendet.
- **Kreisfrei (azyklisch)**: Graph ohne jegliche Kreise.

Eine Folgerung: Ungerichtete Graphen können keinen Kreis der Länge 2 enthalten (Schleifen haben Länge 1).

# Repräsentation mit Matrix

Graph  $G = (V, E)$  mit Knotenmenge  $v_1, \dots, v_n$  gespeichert als **Adjazenzmatrix**  $A_G = (a_{ij})_{1 \leq i, j \leq n}$  mit Einträgen aus  $\{0, 1\}$ .  $a_{ij} = 1$  genau dann wenn Kante von  $v_i$  nach  $v_j$ .



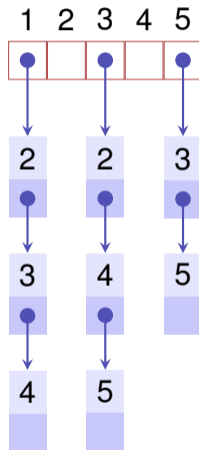
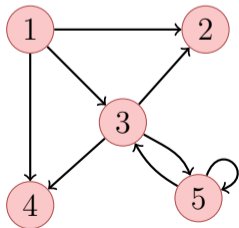
$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Speicherbedarf  $\Theta(|V|^2)$ .  $A_G$  ist symmetrisch, wenn  $G$  ungerichtet.



# Repräsentation mit Liste

Viele Graphen  $G = (V, E)$  mit Knotenmenge  $v_1, \dots, v_n$  haben deutlich weniger als  $n^2$  Kanten. Repräsentation mit **Adjazenzliste**: Array  $A[1], \dots, A[n]$ ,  $A_i$  enthält verkettete Liste aller Knoten in  $N^+(v_i)$ .



Speicherbedarf  $\Theta(|V| + |E|)$ .

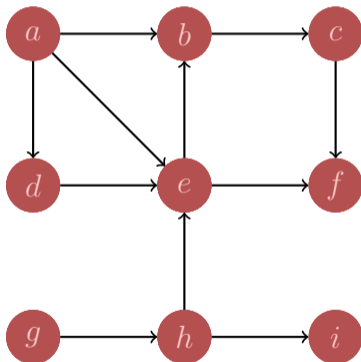
# Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden	$\Theta(n)$	$\Theta(\deg^+ v)$
$v \in V$ ohne Nachbar/Nachfolger finden	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$ ?	$\Theta(1)$	$\Theta(\deg^+ v)$
Kante einfügen	$\Theta(1)$	$\Theta(1)$
Kante löschen	$\Theta(1)$	$\Theta(\deg^+ v)$



# Graphen Traversieren: Tiefensuche

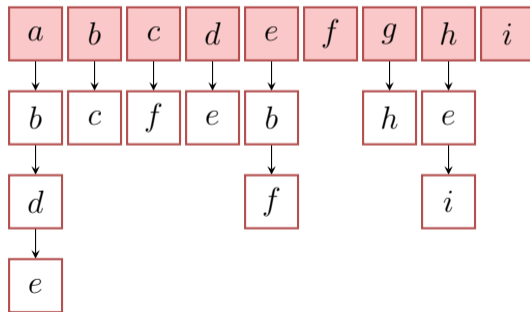
Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.



Reihenfolge

$a, b, c, f, d, e, g, h, i$

Adjazenzliste



## Konzeptuelle Färbung der Knoten

- **Weiss:** Knoten wurde noch nicht entdeckt.
- **Grau:** Knoten wurde entdeckt und zur Traversierung vorgemerkt / in Bearbeitung.
- **Schwarz:** Knoten wurde entdeckt und vollständig bearbeitet

# Algorithmus Tiefensuche DFS-Visit( $G, v$ )

**Input:** Graph  $G = (V, E)$ , Knoten  $v$ .

$v.color \leftarrow \text{grey}$

**foreach**  $w \in N^+(v)$  **do**

**if**  $w.color = \text{white}$  **then**  
        └ DFS-Visit( $G, w$ )

$v.color \leftarrow \text{black}$

Tiefensuche ab Knoten  $v$ . Laufzeit (ohne Rekursion):  $\Theta(\text{deg}^+ v)$

# Algorithmus Tiefensuche DFS-Visit( $G$ )

**Input:** Graph  $G = (V, E)$

**foreach**  $v \in V$  **do**

└  $v.color \leftarrow \text{white}$

**foreach**  $v \in V$  **do**

└ **if**  $v.color = \text{white}$  **then**  
└ DFS-Visit( $G, v$ )

Tiefensuche für alle Knoten eines Graphen. Laufzeit

$$\Theta(|V| + \sum_{v \in V} (\deg^+(v) + 1)) = \Theta(|V| + |E|).$$

# Iteratives DFS-Visit( $G, v$ )

**Input:** Graph  $G = (V, E)$ ,  $v \in V$  mit  $v.color = \text{white}$

Stack  $S \leftarrow \emptyset$

$v.color \leftarrow \text{grey}$ ;  $S.push(v)$  // invariant: grey nodes always on stack

**while**  $S \neq \emptyset$  **do**

$w \leftarrow \text{nextWhiteSuccessor}(v)$  // code: next slide

**if**  $w \neq \text{null}$  **then**

$w.color \leftarrow \text{grey}$ ;  $S.push(w)$

$v \leftarrow w$  // work on  $w$ . parent remains on the stack

**else**

$v.color \leftarrow \text{black}$  // no grey successors,  $v$  becomes black

**if**  $S \neq \emptyset$  **then**

$v \leftarrow S.pop()$  // visit/revisit next node

**if**  $v.color = \text{grey}$  **then**  $S.push(v)$

Speicherbedarf Stack  $\Theta(|V|)$



# nextWhiteSuccessor( $v$ )

**Input:** Knoten  $v \in V$

**Output:** Nachfolgeknoten  $u$  von  $v$  mit  $u.color = \text{white}$ , null sonst

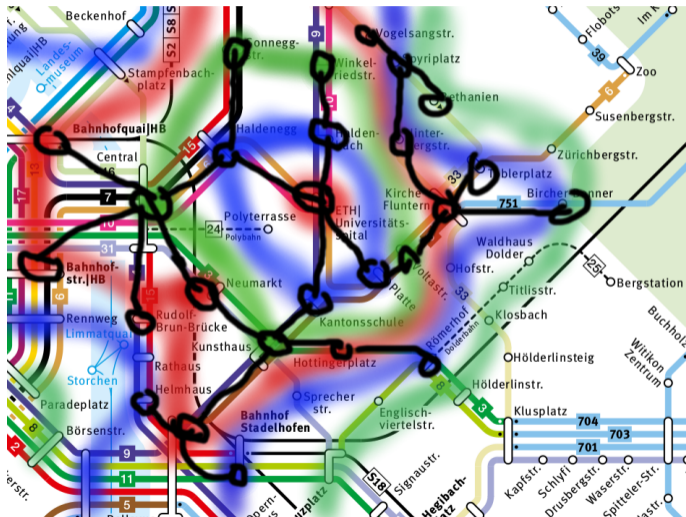
```
foreach  $u \in N^+(v)$  do
  if  $u.color = \text{white}$  then
    return  $u$ 
return null
```

# Interpretation der Farben

Beim Traversieren des Graphen wird ein Baum (oder Wald) aufgebaut. Beim Entdecken von Knoten gibt es drei Fälle

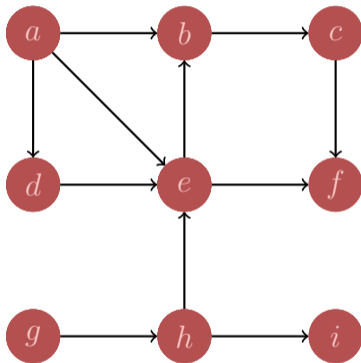
- Weisser Knoten: neue Baumkante
- Grauer Knoten: Zyklus („Rückwärtskante“)
- Schwarzer Knoten: Vorwärts-/Seitwärtskante

# Breitensuche



# Graphen Traversieren: Breitensuche

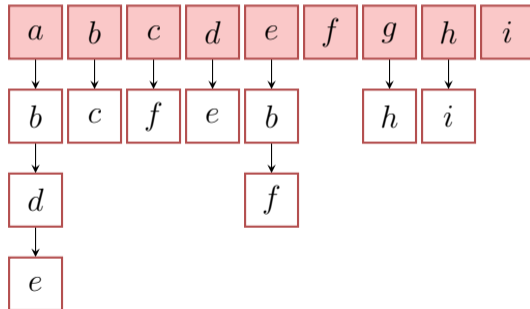
Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.



Reihenfolge

$a, b, d, e, c, f, g, h, i$

Adjazenzliste



# (Iteratives) BFS-Visit( $G, v$ )

**Input:** Graph  $G = (V, E)$

Queue  $Q \leftarrow \emptyset$

$v.color \leftarrow \text{grey}$

enqueue( $Q, v$ )

**while**  $Q \neq \emptyset$  **do**

$w \leftarrow \text{dequeue}(Q)$

**foreach**  $c \in N^+(w)$  **do**

**if**  $c.color = \text{white}$  **then**

$c.color \leftarrow \text{grey}$

            enqueue( $Q, c$ )

$w.color \leftarrow \text{black}$

Algorithmus kommt mit  $\mathcal{O}(|V|)$  Extraplatz aus.

# Rahmenprogramm BFS-Visit( $G$ )

**Input:** Graph  $G = (V, E)$

**foreach**  $v \in V$  **do**

└  $v.color \leftarrow \text{white}$

**foreach**  $v \in V$  **do**

└ **if**  $v.color = \text{white}$  **then**  
└ BFS-Visit( $G, v$ )

Breitensuche für alle Knoten eines Graphen. Laufzeit  $\Theta(|V| + |E|)$ .

# Topologisches Sortieren

The screenshot shows an Excel spreadsheet with the following data:

	A	B	C	D	E	F	G	H	I
1		Task 1	Task 2	Task 3	Task 4	Total		Note	
2	TOTAL	8	8	10	10	36			
3	Arleen	4	5	6	9	24		4	
4	Hans	1	3	2	3	9		1.5	
5	Mike	2	7	5	4	18		3	
6	Selina	6	5	8	2	21		3.5	
7									
8					Durchschnitt	18		3	
9									
10									
11									
12									
13									
14									

Auswertungsreihenfolge?

# Topologische Sortierung

*Topologische Sortierung* eines azyklischen gerichteten Graphen  $G = (V, E)$ :

Bijektive Abbildung

$$\text{ord} : V \rightarrow \{1, \dots, |V|\}$$

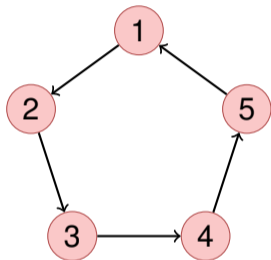
so dass

$$\text{ord}(v) < \text{ord}(w) \quad \forall (v, w) \in E.$$

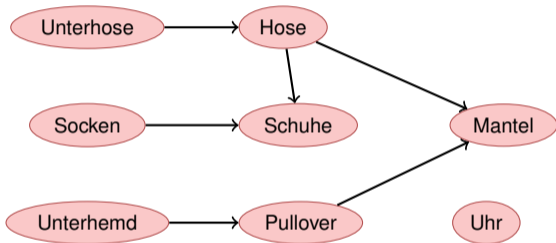
Identifizieren Wert  $i$  mit dem Element  $v_i := \text{ord}^{-1}(i)$ . Topologische Sortierung  $\hat{=} \langle v_1, \dots, v_{|V|} \rangle$ .



# (Gegen-)Beispiele



Zyklischer Graph: kann nicht topologisch sortiert werden.



Eine mögliche Topologische Sortierung des Graphen:  
Unterhemd,Pullover,Unterhose,Uhr,Hose,Mantel,Socken,Schuhe

# Beobachtung

## Theorem

*Ein gerichteter Graph  $G = (V, E)$  besitzt genau dann eine topologische Sortierung, wenn er kreisfrei ist*

Beweis " $\Rightarrow$ ": Wenn  $G$  einen Kreis besitzt, so besitzt er keine topologische Sortierung. Denn in einem Kreis  $\langle v_{i_1}, \dots, v_{i_m} \rangle$  gälte  $v_{i_1} < \dots < v_{i_m} < v_{i_1}$ .

# Induktiver Beweis Gegenrichtung

- Anfang ( $n = 1$ ): Graph mit einem Knoten ohne Schleife ist topologisch sortierbar. Setze  $\text{ord}(v_1) = 1$ .
- Hypothese: Graph mit  $n$  Knoten kann topologisch sortiert werden.
- Schritt ( $n \rightarrow n + 1$ ):
  - 1  $G$  enthält einen Knoten  $v_q$  mit Eingangsgrad  $\text{deg}^-(v_q) = 0$ . Andernfalls verfolge iterativ Kanten rückwärts – nach spätestens  $n + 1$  Iterationen würde man einen Knoten besuchen, welcher bereits besucht wurde. Widerspruch zur Zyklenfreiheit.
  - 2 Graph ohne Knoten  $v_q$  und ohne dessen Eingangskanten kann nach Hypothese topologisch sortiert werden. Verwende diese Sortierung, setze  $\text{ord}(v_i) \leftarrow \text{ord}(v_i) + 1$  für alle  $i \neq q$  und setze  $\text{ord}(v_q) \leftarrow 1$ .

# Algorithmus, vorläufiger Entwurf

Graph  $G = (V, E)$ .  $d \leftarrow 1$

- 1 Traversiere von beliebigem Knoten rückwärts bis ein Knoten  $v_q$  mit Eingangsgrad 0 gefunden ist.
- 2 Wird kein Knoten mit Eingangsgrad 0 gefunden ( $n$  Schritte), dann Zyklus gefunden.
- 3 Setze  $\text{ord}(v_q) \leftarrow d$ .
- 4 Entferne  $v_q$  und seine Kanten von  $G$ .
- 5 Wenn  $V \neq \emptyset$ , dann  $d \leftarrow d + 1$ , gehe zu Schritt 1.

Laufzeit im schlechtesten Fall:  $\Theta(|V|^2)$ .

# Verbesserung

Idee?

Berechne die Eingangsgrade der Knoten im Voraus und durchlaufe dann jeweils die Knoten mit Eingangsgrad 0 die Eingangsgrade der Nachfolgeknoten korrigierend.

# Algorithmus Topological-Sort( $G$ )

**Input:** Graph  $G = (V, E)$ .

**Output:** Topologische Sortierung ord

Stack  $S \leftarrow \emptyset$

**foreach**  $v \in V$  **do**  $A[v] \leftarrow 0$

**foreach**  $(v, w) \in E$  **do**  $A[w] \leftarrow A[w] + 1$  // Eingangsgrade berechnen

**foreach**  $v \in V$  with  $A[v] = 0$  **do**  $\text{push}(S, v)$  // Merke Nodes mit Eingangsgrad 0

$i \leftarrow 1$

**while**  $S \neq \emptyset$  **do**

$v \leftarrow \text{pop}(S)$ ;  $\text{ord}[v] \leftarrow i$ ;  $i \leftarrow i + 1$  // Wähle Knoten mit Eingangsgrad 0

**foreach**  $(v, w) \in E$  **do** // Verringere Eingangsgrad der Nachfolger

$A[w] \leftarrow A[w] - 1$

**if**  $A[w] = 0$  **then**  $\text{push}(S, w)$

**if**  $i = |V| + 1$  **then return** ord **else return** "Cycle Detected"

# Algorithmus Korrektheit

## Theorem

*Sei  $G = (V, E)$  ein gerichteter, kreisfreier Graph. Der Algorithmus  $\text{TopologicalSort}(G)$  berechnet in Zeit  $\Theta(|V| + |E|)$  eine topologische Sortierung  $\text{ord}$  für  $G$ .*

Beweis: folgt im wesentlichen aus vorigem Theorem:

- 1 Eingangsgrad verringern entspricht Knotenentfernen.
- 2 Im Algorithmus gilt für jeden Knoten  $v$  mit  $A[v] = 0$  dass entweder der Knoten Eingangsgrad 0 hat oder dass zuvor alle Vorgänger einen Wert  $\text{ord}[u] \leftarrow i$  zugewiesen bekamen und somit  $\text{ord}[v] > \text{ord}[u]$  für alle Vorgänger  $u$  von  $v$ . Knoten werden nur einmal auf den Stack gelegt.
- 3 Laufzeit: Inspektion des Algorithmus (mit Argumenten wie beim Traversieren).

# Algorithmus Korrektheit

## Theorem

*Sei  $G = (V, E)$  ein gerichteter, nicht kreisfreier Graph. Der Algorithmus  $\text{TopologicalSort}(G)$  terminiert in Zeit  $\Theta(|V| + |E|)$  und detektiert Zyklus.*

Beweis: Sei  $\langle v_{i_1}, \dots, v_{i_k} \rangle$  ein Kreis in  $G$ . In jedem Schritt des Algorithmus bleibt  $A[v_{i_j}] \geq 1$  für alle  $j = 1, \dots, k$ . Also werden  $k$  Knoten nie auf den Stack gelegt und somit ist zum Schluss  $i \leq V + 1 - k$ .

Die Laufzeit des zweiten Teils des Algorithmus kann kürzer werden, jedoch kostet die Berechnung der Eingangsgrade bereits  $\Theta(|V| + |E|)$ .



# Alternative: Algorithmus DFS-Topsort( $G, v$ )

**Input:** Graph  $G = (V, E)$ , Knoten  $v$ , Knotenliste  $L$ .

**if**  $v.color = grey$  **then**

  | stop (Zyklus)

**if**  $v.color = black$  **then**

  | **return**

$v.color \leftarrow grey$

**foreach**  $w \in N^+(v)$  **do**

  | DFS-Topsort( $G, w$ )

$v.color \leftarrow black$

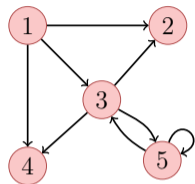
Füge  $v$  am Anfang der Liste  $L$  ein

Rufe Algorithmus für jeden noch nicht besuchten Knoten auf.

Asymptotische Laufzeit  $\Theta(|V| + |E|)$ .

# Adjazenzmatrizen multipliziert

$$B := A_G^2 = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}^2 = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 2 \end{pmatrix}$$



# Interpretation

## Theorem

*Sei  $G = (V, E)$  ein Graph und  $k \in \mathbb{N}$ . Dann gibt das Element  $a_{i,j}^{(k)}$  der Matrix  $(a_{i,j}^{(k)})_{1 \leq i,j \leq n} = (A_G)^k$  die Anzahl der Wege mit Länge  $k$  von  $v_i$  nach  $v_j$  an.*

# Beweis

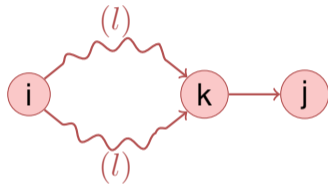
Per Induktion.

**Anfang:** Klar für  $k = 1$ .  $a_{i,j} = a_{i,j}^{(1)}$ .

**Hypothese:** Aussage wahr für alle  $k \leq l$

**Schritt ( $l \rightarrow l + 1$ ):**

$$a_{i,j}^{(l+1)} = \sum_{k=1}^n a_{i,k}^{(l)} \cdot a_{k,j}$$



$a_{k,j} = 1$  g.d.w. Kante von  $k$  nach  $j$ , 0 sonst. Summe zählt die Anzahl Wege der Länge  $l$  vom Knoten  $v_i$  zu allen Knoten  $v_k$  welche direkte Verbindung zu Knoten  $v_j$  haben, also alle Wege der Länge  $l + 1$ .

# Beispiel: Kürzester Weg

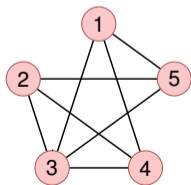
*Frage:* existiert Weg von  $i$  nach  $j$ ? Wie lang ist der kürzeste Weg?

*Antwort:* Potenziere  $A_G$  bis für ein  $k < n$  gilt  $a_{i,j}^{(k)} > 0$ .  $k$  gibt die Weglänge des kürzesten Weges. Wenn  $a_{i,j}^{(k)} = 0$  für alle  $1 \leq k < n$ , so gibt es keinen Weg von  $i$  nach  $j$ .

# Beispiel: Anzahl Dreiecke

**Frage:** Wie viele Dreieckswege enthält ein ungerichteter Graph?

**Antwort:** Entferne alle Zyklen (Diagonaleinträge). Berechne  $A_G^3$ .  $a_{ii}^{(3)}$  bestimmt die Anzahl Wege der Länge 3, die  $i$  enthalten. Es gibt 6 verschiedene Permutationen eines Dreiecks. Damit Anzahl Dreiecke:  $\sum_{i=1}^n a_{ii}^{(3)} / 6$ .



$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}^3 = \begin{pmatrix} 4 & 4 & 8 & 8 & 8 \\ 4 & 4 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 4 & 4 \\ 8 & 8 & 8 & 4 & 4 \end{pmatrix} \Rightarrow 24/6 = 4 \text{ Dreiecke.}$$

# Relation

Gegeben: endliche Menge  $V$

(Binäre) **Relation**  $R$  auf  $V$ : Teilmenge des kartesischen Produkts  $V \times V = \{(a, b) | a \in V, b \in V\}$

Relation  $R \subseteq V \times V$  heisst

- *reflexiv*, wenn  $(v, v) \in R$  für alle  $v \in V$
- *symmetrisch*, wenn  $(v, w) \in R \Rightarrow (w, v) \in R$
- *transitiv*, wenn  $(v, x) \in R, (x, w) \in R \Rightarrow (v, w) \in R$

Die *(Reflexive) Transitive Hülle*  $R^*$  von  $R$  ist die kleinste Erweiterung  $R \subseteq R^* \subseteq V \times V$  von  $R$ , so dass  $R^*$  reflexiv und transitiv ist.

# Graphen und Relationen

Graph  $G = (V, E)$

Adjazenzen  $A_G \hat{=} \text{Relation } E \subseteq V \times V \text{ auf } V$

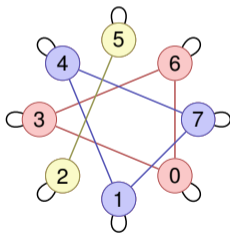
- **reflexiv**  $\Leftrightarrow a_{i,i} = 1$  für alle  $i = 1, \dots, n$ . (Schleifen)
- **symmetrisch**  $\Leftrightarrow a_{i,j} = a_{j,i}$  für alle  $i, j = 1, \dots, n$  (ungerichtet)
- **transitiv**  $\Leftrightarrow (u, v) \in E, (v, w) \in E \Rightarrow (u, w) \in E$ . (Erreichbarkeit)



# Beispiel: Äquivalenzrelation

Äquivalenzrelation  $\Leftrightarrow$  symmetrische, transitive, reflexive Relation  
 $\Leftrightarrow$  Kollektion vollständiger, ungerichteter Graphen, für den jedes Element eine Schleife hat.

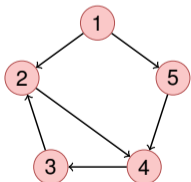
**Beispiel:** Äquivalenzklassen der Zahlen  $\{0, \dots, 7\}$  modulo 3



# Reflexive Transitive Hülle

Reflexive transitive Hülle von  $G \Leftrightarrow$  *Erreichbarkeitsrelation*  $E^*$ :  
 $(v, w) \in E^*$  gdw.  $\exists$  Weg von Knoten  $v$  zu  $w$ .

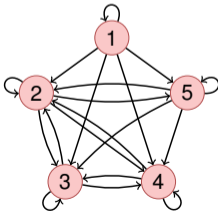
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$



$$G = (V, E)$$



$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$



$$G^* = (V, E^*)$$

# Berechnung Reflexive Transitive Hülle

**Ziel:** Berechnung von  $B = (b_{ij})_{1 \leq i, j \leq n}$  mit  $b_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E^*$

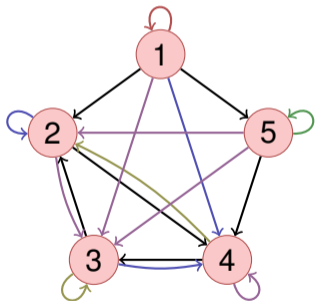
**Beobachtung:**  $a_{ij} = 1$  bedeutet bereits  $(v_i, v_j) \in E^*$ .

Erste Idee:

- Starte mit  $B \leftarrow A$  und setze  $b_{ii} = 1$  für alle  $i$  (Reflexivität).
- Iteriere über  $i, j, k$  und setze  $b_{ij} = 1$ , wenn  $b_{ik} = 1$  und  $b_{kj} = 1$ .  
Dann alle Wege der Länge 1 und 2 berücksichtigt
- Wiederhole Iteration  $\Rightarrow$  alle Wege der Länge 1 ... 4 berücksichtigt.
- $\lceil \log_2 n \rceil$  Wiederholungen nötig.  $\Rightarrow$  Laufzeit  $n^3 \lceil \log_2 n \rceil$

# Verbesserung: Algorithmus von Warshall (1962)

Induktiver Ansatz: Alle Wege bekannt über Knoten aus  $\{v_i : i < k\}$ .  
Hinzunahme des Knotens  $v_k$ .



$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

# Algorithmus TransitiveClosure( $A_G$ )

**Input:** Adjazenzmatrix  $A_G = (a_{ij})_{i,j=1\dots n}$

**Output:** Reflexive Transitive Hülle  $B = (b_{ij})_{i,j=1\dots n}$  von  $G$

$B \leftarrow A_G$

**for**  $k \leftarrow 1$  **to**  $n$  **do**

$a_{kk} \leftarrow 1$

// Reflexivität

**for**  $i \leftarrow 1$  **to**  $n$  **do**

**for**  $j \leftarrow 1$  **to**  $n$  **do**

$b_{ij} \leftarrow \max\{b_{ij}, b_{ik} \cdot b_{kj}\}$

// Alle Wege über  $v_k$

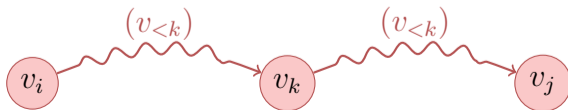
**return**  $B$

Laufzeit des Algorithmus  $\Theta(n^3)$ .

# Korrektheit des Algorithmus (Induktion)

**Invariante ( $k$ ):** alle Wege über Knoten mit maximalem Index  $< k$  berücksichtigt

- **Anfang ( $k = 1$ ):** Alle direkten Wege (alle Kanten) in  $A_G$  berücksichtigt.
- **Hypothese:** Invariante ( $k$ ) erfüllt.
- **Schritt ( $k \rightarrow k + 1$ ):** Für jeden Weg von  $v_i$  nach  $v_j$  über Knoten mit maximalen Index  $k$ : nach Hypothese  $b_{ik} = 1$  und  $b_{kj} = 1$ .  
Somit im  $k$ -ten Schleifendurchlauf:  $b_{ij} \leftarrow 1$ .



# Zusammenhangskomponenten

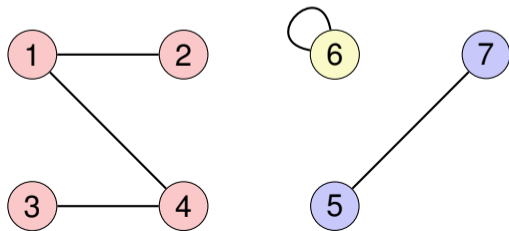
Zusammenhangskomponenten eines ungerichteten Graphen  $G$ :  
Äquivalenzklassen der reflexiven, transitiven Hülle von  $G$ .

Zusammenhangskomponente = Teilgraph  $G' = (V', E')$ ,

$E' = \{\{v, w\} \in E \mid v, w \in V'\}$  mit

$\{\{v, w\} \in E \mid v \in V' \vee w \in V'\} = E = \{\{v, w\} \in E \mid v \in V' \wedge w \in V'\}$

---



Graph mit Zusammenhangskomponenten  $\{1, 2, 3, 4\}$ ,  $\{5, 7\}$ ,  $\{6\}$ .

# Berechnung der Zusammenhangskomponenten

- Berechnung einer Partitionierung von  $V$  in paarweise disjunkte Teilmengen  $V_1, \dots, V_k$
- so dass jedes  $V_i$  die Knoten einer Zusammenhangskomponente enthält.
- Algorithmus: Tiefen- oder Breitensuche. Bei jedem Neustart von  $\text{DFSSearch}(G, v)$  oder  $\text{BFSSearch}(G, v)$  neue leere Zusammenhangskomponente erstellen und alle traversierten Knoten einfügen.



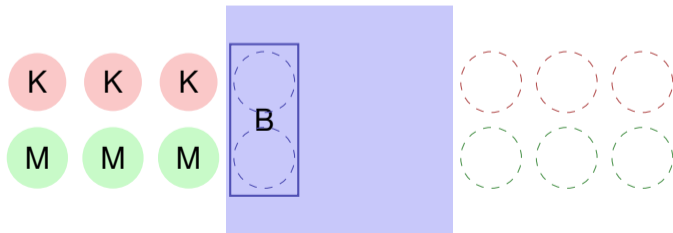
# 24. Kürzeste Wege

Motivation, Dijkstras Algorithmus auf Distanzgraphen, Algorithmus von Bellman-Ford, Algorithmus von Floyd-Warshall

[Ottman/Widmayer, Kap. 9.5 Cormen et al, Kap. 24.1-24.3, 25.2-25.3]

# Flussüberquerung (Missionare und Kannibalen)

Problem: Drei Kannibalen und drei Missionare stehen an einem Ufer eines Flusses. Ein dort bereitstehendes Boot fasst maximal zwei Personen. Zu keiner Zeit dürfen an einem Ort (Ufer oder Boot) mehr Kannibalen als Missionare sein. Wie kommen die Missionare und Kannibalen möglichst schnell über den Fluss? <sup>48</sup>



<sup>48</sup>Es gibt leichte Variationen dieses Problems, es ist auch äquivalent zum Problem der eifersüchtigen Ehemänner

# Formulierung als Graph

Zähle alle erlaubten Konfigurationen als Knoten auf und verbinde diese mit einer Kante, wenn Überfahrt möglich ist. Das Problem ist dann ein Problem des kürzesten Pfades

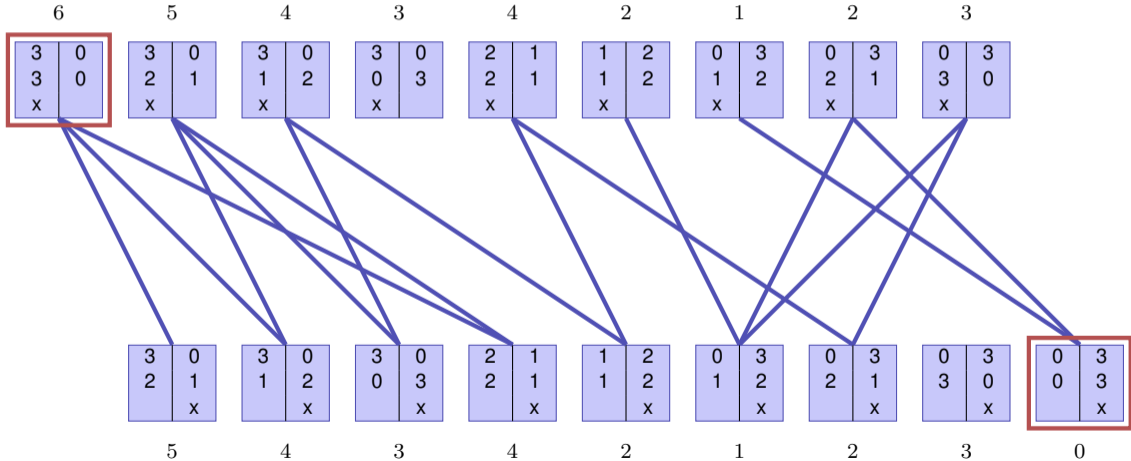
Beispiel

	links	rechts		links	rechts	
Missionare	3	0	Überfahrt möglich	Missionare	2	1
Kannibalen	3	0		Kannibalen	2	1
Boot	x			Boot		x

6 Personen am linken Ufer

4 Personen am linken Ufer

# Das ganze Problem als Graph

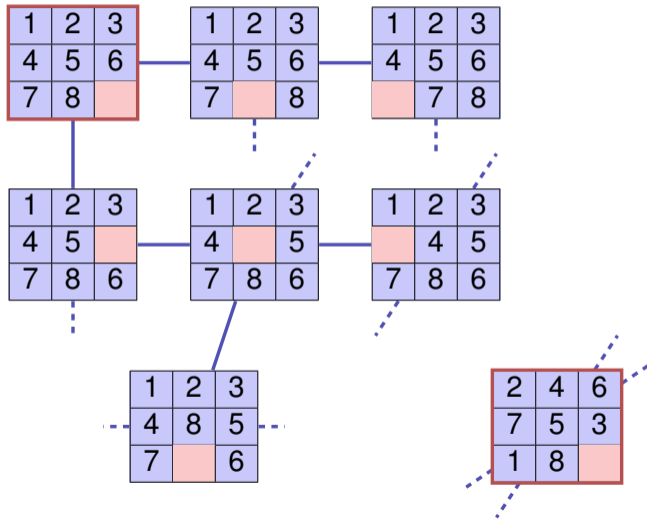


# Anderes Beispiel: Schiebepuzzle

Wollen die schnellste Lösung finden für

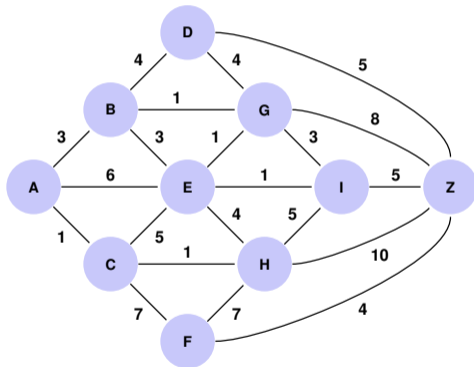


# Problem als Graph



# Routenfinder

Gegeben Städte A - Z und Distanzen zwischen den Städten.

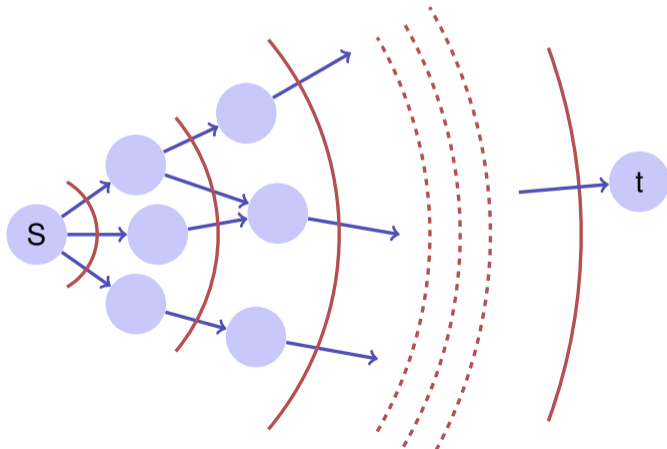


Was ist der kürzeste Weg von A nach Z?

# Einfachster Fall

Konstantes Kantengewicht 1 (oBdA)

Lösung: Breitensuche





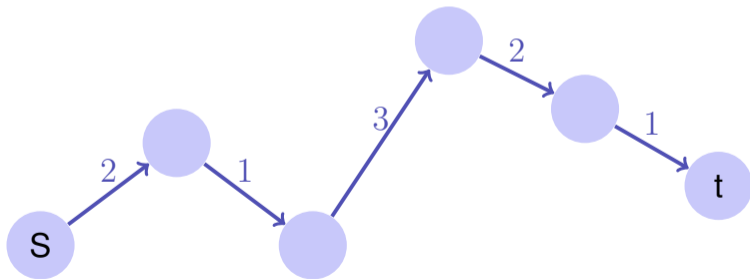
# Gewichtete Graphen

**Gegeben:**  $G = (V, E, c)$ ,  $c : E \rightarrow \mathbb{R}$ ,  $s, t \in V$ .

**Gesucht:** Länge (Gewicht) eines kürzesten Weges von  $s$  nach  $t$ .

**Weg:**  $p = \langle s = v_0, v_1, \dots, v_k = t \rangle$ ,  $(v_i, v_{i+1}) \in E$  ( $0 \leq i < k$ )

**Gewicht:**  $c(p) := \sum_{i=0}^{k-1} c((v_i, v_{i+1}))$ .



Weg mit Gewicht 9

# Kürzeste Wege

**Notation:** Wir schreiben

$$u \overset{p}{\rightsquigarrow} v \quad \text{oder} \quad p : u \rightsquigarrow v$$

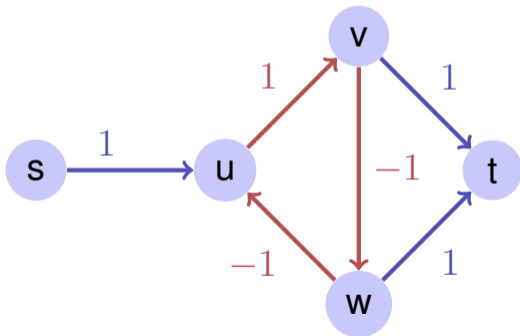
und meinen einen Weg  $p$  von  $u$  nach  $v$

**Notation:**  $\delta(u, v)$  = Gewicht eines kürzesten Weges von  $u$  nach  $v$ :

$$\delta(u, v) = \begin{cases} \infty & \text{kein Weg von } u \text{ nach } v \\ \min\{c(p) : u \overset{p}{\rightsquigarrow} v\} & \text{sonst} \end{cases}$$

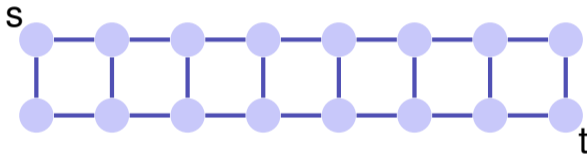
# Beobachtungen (1)

Es gibt Situationen, in denen kein kürzester Weg existiert: negative Zyklen könnten auftreten.



## Beobachtungen (2)

Es kann exponentiell viele Wege geben.



(mindestens  $2^{|V|/2}$  Wege von  $s$  nach  $t$ )

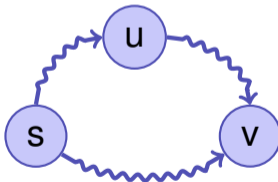
⇒ Alle Wege probieren ist zu ineffizient.

# Beobachtungen (3)

## *Dreiecksungleichung*

Für alle  $s, u, v \in V$ :

$$\delta(s, v) \leq \delta(s, u) + \delta(u, v)$$

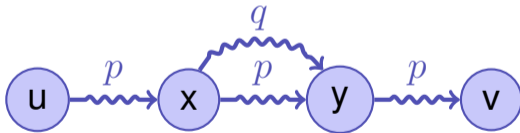


Ein kürzester Weg von  $s$  nach  $v$  (ohne weitere Einschränkungen) kann nicht länger sein als ein kürzester Weg von  $s$  nach  $v$ , der  $u$  enthalten muss.

# Beobachtungen (4)

## Optimale Substruktur

Teilpfade von kürzesten Pfaden sind kürzeste Pfade: Sei  $p = \langle v_0, \dots, v_k \rangle$  ein kürzester Pfad von  $v_0$  nach  $v_k$ . Dann ist jeder der Teilpfade  $p_{ij} = \langle v_i, \dots, v_j \rangle$  ( $0 \leq i < j \leq k$ ) ein kürzester Pfad von  $v_i$  nach  $v_j$ .



Wäre das nicht so, könnte man einen der Teilpfade kürzen, Widerspruch zur Voraussetzung.

# Beobachtungen (5)

## Kürzeste Wege enthalten keine Zyklen

- 1 Kürzester Weg enthält negativen Zyklus: es existiert kein kürzester Weg. Widerspruch.
- 2 Weg enthält positiven Zyklus: Weglassen des positiven Zyklus kann den Weg verkürzen: Widerspruch
- 3 Weg enthält Zyklus vom Gewicht 0: Weglassen des Zyklus verändert das Pfadgewicht nicht. Weglassen (Konvention).

# Zutaten für einen Algorithmus

Gesucht: Kürzeste Wege von einem Startknoten  $s$  aus.

- Gewicht des kürzesten bisher gefundenen Pfades

$$d_s : V \rightarrow \mathbb{R}$$

*Zu Beginn:*  $d_s[v] = \infty$  für alle Knoten  $v \in V$ .

*Ziel:*  $d_s[v] = \delta(s, v)$  für alle  $v \in V$ .

- Vorgänger eines Knotens

$$\pi_s : V \rightarrow V$$

Zu Beginn  $\pi_s[v]$  undefiniert für jeden Knoten  $v \in V$



# Allgemeiner Algorithmus

- 1 Initialisiere  $d_s$  und  $\pi_s$ :  $d_s[v] = \infty$ ,  $\pi_s[v] = \text{null}$  für alle  $v \in V$
- 2 Setze  $d_s[s] \leftarrow 0$
- 3 Wähle eine Kante  $(u, v) \in E$   
Relaxiere  $(u, v)$ :  
if  $d_s[v] > d_s[u] + c(u, v)$  then  
     $d_s[v] \leftarrow d_s[u] + c(u, v)$   
     $\pi_s[v] \leftarrow u$
- 4 Wiederhole 3 bis nichts mehr relaxiert werden kann.  
(bis  $d_s[v] \leq d_s[u] + c(u, v) \quad \forall (u, v) \in E$ )

# Relaxieren ist sicher

Zu jeder Zeit gilt in obigem Algorithmus

$$d_s[v] \geq \delta(s, v) \quad \forall v \in V$$

Im Relaxierschritt:

$$\delta(s, v) \leq \delta(s, u) + \delta(u, v) \quad \text{[Dreiecksungleichung].}$$

$$\delta(s, u) \leq d_s[u] \quad \text{[Induktionsvoraussetzung].}$$

$$\delta(u, v) \leq c(u, v) \quad \text{[Minimalität von } \delta \text{]}$$

$$\Rightarrow d_s[u] + c(u, v) \geq \delta(s, v)$$

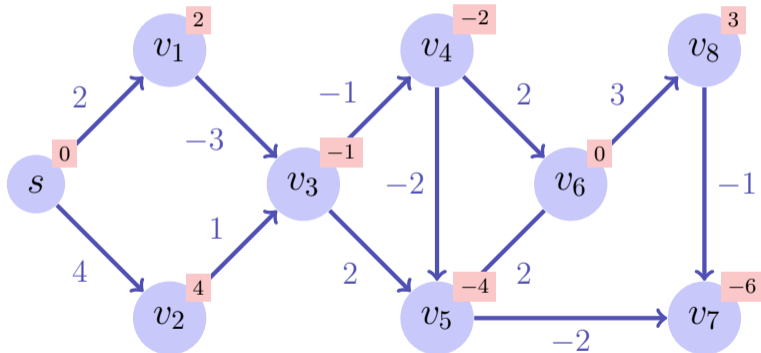
$$\Rightarrow \min\{d_s[v], d_s[u] + c(u, v)\} \geq \delta(s, v)$$

# Zentrale Frage

Wie / in welcher Reihenfolge wählt man die Kanten in obigem Algorithmus?

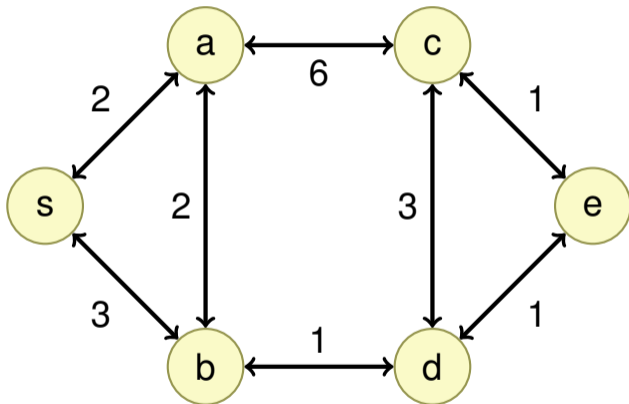
# Spezialfall: Gerichteter Azyklischer Graph (DAG)

DAG  $\Rightarrow$  Topologische Sortierung liefert optimale Besuchsreihenfolge



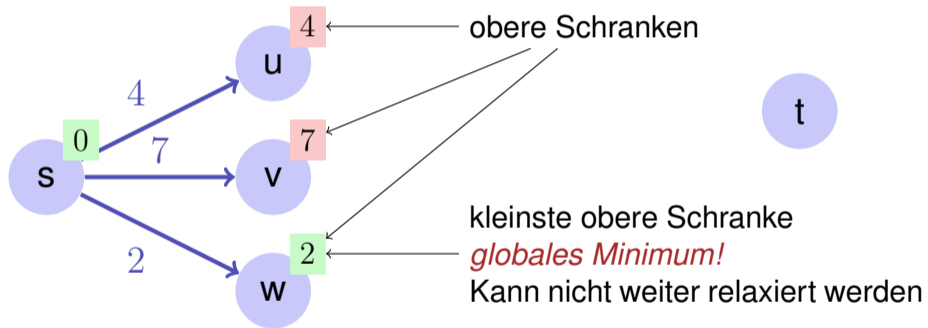
Top. Sortieren:  $\Rightarrow$  Reihenfolge  $s, v_1, v_2, v_3, v_4, v_6, v_5, v_8, v_7$ .

# Annahme (vorübergehend)



Alle Gewichte von  $G$  sind *positiv*.

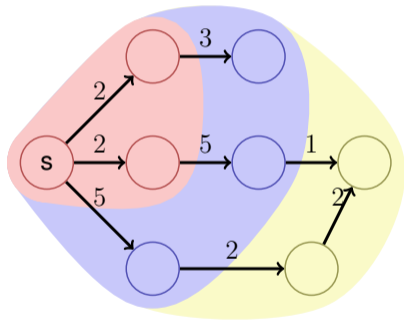
# Beobachtung (Dijkstra)



# Grundidee

Menge  $V$  aller Knoten wird unterteilt in

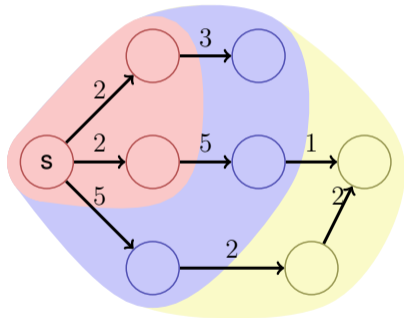
- die Menge  $M$  von Knoten, für die schon ein kürzester Weg von  $s$  bekannt ist
- die Menge  $R = \bigcup_{v \in M} N^+(v) \setminus M$  von Knoten, für die kein kürzester Weg bekannt ist, die jedoch von  $M$  direkt erreichbar sind.
- die Menge  $U = V \setminus (M \cup R)$  von Knoten, die noch nicht berücksichtigt wurden.



# Induktion

Induktion über  $|M|$ : Wähle Knoten aus  $R$  mit kleinster oberer Schranke. Nimm  $r$  zu  $M$  hinzu, und update  $R$  und  $U$ .

Korrektheit: Ist innerhalb einer “Wellenfront” einmal ein Knoten mit minimalem Pfadgewicht  $w$  gefunden, kann kein Pfad über später gefundene Knoten (mit Gewicht  $\geq w$ ) zu einer Verbesserung führen.





# Algorithmus Dijkstra( $G, s$ )

**Input:** Positiv gewichteter Graph  $G = (V, E, c)$ , Startpunkt  $s \in V$

**Output:** Minimale Gewichte  $d$  der kürzesten Pfade und Vorgängerknoten für jeden Knoten.

**foreach**  $u \in V$  **do**

$d_s[u] \leftarrow \infty$ ;  $\pi_s[u] \leftarrow \mathbf{null}$

$d_s[s] \leftarrow 0$ ;  $R \leftarrow \{s\}$

**while**  $R \neq \emptyset$  **do**

$u \leftarrow \text{ExtractMin}(R)$

**foreach**  $v \in N^+(u)$  **do**

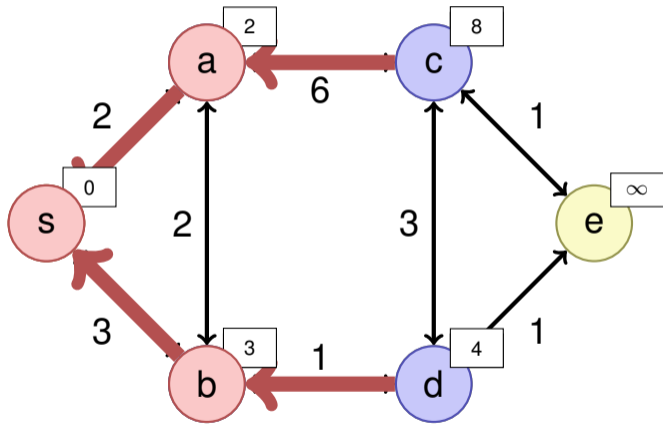
**if**  $d_s[u] + c(u, v) < d_s[v]$  **then**

$d_s[v] \leftarrow d_s[u] + c(u, v)$

$\pi_s[v] \leftarrow u$

$R \leftarrow R \cup \{v\}$

# Beispiel



$$M = \{s, a, b\}$$

$$R = \{c, d\}$$

$$U = \{e\}$$

# Zur Implementation: Datenstruktur für $R$ ?

Benötigte Operationen:

- Insert (Hinzunehmen zu  $R$ )
- ExtractMin (über  $R$ ) und DecreaseKey (Update in  $R$ )

```
foreach  $v \in N^+(u)$  do
  if  $d_s[u] + c(u, v) < d_s[v]$  then
     $d_s[v] \leftarrow d_s[u] + c(u, v)$ 
     $\pi_s[v] \leftarrow u$ 
    if  $v \in R$  then
      DecreaseKey( $R, v$ )           // Update eines  $d(v)$  im Heap zu  $R$ 
    else
       $R \leftarrow R \cup \{v\}$     // Einfügen eines neuen  $d(v)$  im Heap zu  $R$ 
```

MinHeap!

# DecreaseKey

- DecreaseKey: Aufsteigen im MinHeap in  $\mathcal{O}(\log |V|)$
- Position im Heap?
  - Möglichkeit (a): Speichern am Knoten
  - Möglichkeit (b): Hashtabelle über Knoten
  - Möglichkeit (c): Knoten nach erfolgreichem Relaxieren erneut einfügen. Knoten beim Entnehmen als "deleted" kennzeichnen (Lazy Deletion).<sup>49</sup>

---

<sup>49</sup>Für die lazy deletion benötigt man ein Paar von Kante (oder Zielknoten) und Distanz

# Laufzeit

- $|V| \times \text{ExtractMin}$ :  $\mathcal{O}(|V| \log |V|)$
- $|E| \times \text{Insert oder DecreaseKey}$ :  $\mathcal{O}(|E| \log |V|)$
- $1 \times \text{Init}$ :  $\mathcal{O}(|V|)$
- Insgesamt:  $\mathcal{O}(|E| \log |V|)$ .

Kann verbessert werden unter Verwendung einer für ExtractMin und DecreaseKey optimierten Datenstruktur (Fibonacci Heap), dann Laufzeit  $\mathcal{O}(|E| + |V| \log |V|)$ .

# Allgemeine Bewertete Graphen

Verbesserungsschritt wie bisherwie bisher, aber mit Rückgabewert:

**Relax**( $u, v$ ) ( $u, v \in V, (u, v) \in E$ )

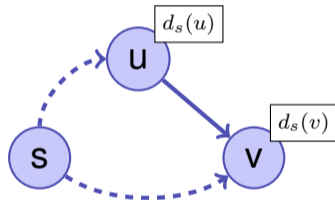
**if**  $d_s[u] + c(u, v) < d_s[v]$  **then**

$d_s[v] \leftarrow d_s[u] + c(u, v)$

$\pi_s[v] \leftarrow u$

**return true**

**return false**



Problem: Zyklen mit negativen Gewichten können Weg verkürzen:  
es muss keinen kürzesten Weg mehr geben

# Dynamic Programming Ansatz (Bellman)

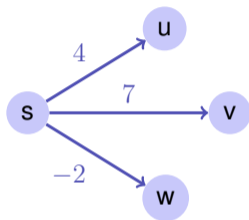
Induktion über Anzahl Kanten.  $d_s[i, v]$ : Kürzeste Weglänge von  $s$  nach  $v$  über maximal  $i$  Kanten.

$$d_s[i, v] = \min\{d_s[i - 1, v], \min_{(u,v) \in E} (d_s[i - 1, u] + c(u, v))\}$$

$$d_s[0, s] = 0, d_s[0, v] = \infty \quad \forall v \neq s.$$

# Dynamic Programming Ansatz (Bellman)

	$s$	$\dots$	$v$	$\dots$	$w$
$0$	$0$	$\infty$	$\infty$	$\infty$	$\infty$
$1$	$0$	$\infty$	$7$	$\infty$	$-2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n - 1$	$0$	$\dots$	$\dots$	$\dots$	$\dots$



Algorithmus: Iteriere über letzte Zeile bis die Relaxationsschritte keine Änderung mehr ergeben, maximal aber  $n - 1$  mal. Wenn dann noch Änderungen, dann gibt es keinen kürzesten Pfad.



# Algorithmus Bellman-Ford( $G, s$ )

**Input:** Graph  $G = (V, E, c)$ , Startpunkt  $s \in V$

**Output:** Wenn Rückgabe true, Minimale Gewichte  $d$  der kürzesten Pfade zu jedem Knoten, sonst kein kürzester Pfad.

**foreach**  $u \in V$  **do**

$d_s[u] \leftarrow \infty$ ;  $\pi_s[u] \leftarrow \text{null}$

$d_s[s] \leftarrow 0$ ;

**for**  $i \leftarrow 1$  **to**  $|V|$  **do**

$f \leftarrow \text{false}$

**foreach**  $(u, v) \in E$  **do**

$f \leftarrow f \vee \text{Relax}(u, v)$

**if**  $f = \text{false}$  **then return** true

**return** false;

# Alle kürzesten Pfade

Ziel: Berechne das Gewicht eines kürzesten Pfades für jedes Knotenpaar.

- $|V| \times$  Anwendung von Dijkstras ShortestPath:  $\mathcal{O}(|V| \cdot |E| \cdot \log |V|)$   
(Mit Fibonacci-Heap:  $\mathcal{O}(|V|^2 \log |V| + |V| \cdot |E|)$ )
- $|V| \times$  Anwendung von Bellman-Ford:  $\mathcal{O}(|E| \cdot |V|^2)$
- Es geht besser!

# Induktion über Knotennummer.<sup>50</sup>

Betrachte die Gewichte aller kürzesten Wege  $S^k$  mit Zwischenknoten in  $V^k := \{v_1, \dots, v_k\}$ , wenn Gewichte zu allen kürzesten Wegen  $S^{k-1}$  mit Zwischenknoten in  $V^{k-1}$  gegeben sind.

- $v_k$  kein Zwischenknoten eines kürzesten Pfades von  $v_i \rightsquigarrow v_j$  in  $V^k$ : Gewicht eines kürzesten Pfades  $v_i \rightsquigarrow v_j$  in  $S^{k-1}$  dann auch das Gewicht eines kürzesten Pfades in  $S^k$ .
- $v_k$  Zwischenknoten eines kürzesten Pfades  $v_i \rightsquigarrow v_j$  in  $V^k$ : Teilpfade  $v_i \rightsquigarrow v_k$  und  $v_k \rightsquigarrow v_j$  enthalten nur Zwischenknoten aus  $S^{k-1}$ .

---

<sup>50</sup>wie beim Algorithmus für die reflexive transitive Hülle von Warshall

# DP Induktion

$d^k(u, v)$  = Minimales Gewicht eines Pfades  $u \rightsquigarrow v$  mit  
Zwischenknoten aus  $V^k$

Induktion

$$d^k(u, v) = \min\{d^{k-1}(u, v), d^{k-1}(u, k) + d^{k-1}(k, v)\} (k \geq 1)$$

$$d^0(u, v) = c(u, v)$$

# DP Algorithmus Floyd-Warshall( $G$ )

**Input:** Azyklischer Graph  $G = (V, E, c)$

**Output:** Minimale Gewichte aller Pfade  $d$

$d^0 \leftarrow c$

**for**  $k \leftarrow 1$  **to**  $|V|$  **do**

**for**  $i \leftarrow 1$  **to**  $|V|$  **do**

**for**  $j \leftarrow 1$  **to**  $|V|$  **do**

$d^k(v_i, v_j) = \min\{d^{k-1}(v_i, v_j), d^{k-1}(v_i, v_k) + d^{k-1}(v_k, v_j)\}$

Laufzeit:  $\Theta(|V|^3)$

Bemerkung: Der Algorithmus kann auf einer einzigen Matrix  $d$  (in place) ausgeführt werden.

# Umgewichtung

Idee: Anwendung von Dijkstras Algorithmus auf Graphen mit negativen Gewichten durch Umgewichtung

Das folgende geht *nicht*. Die Graphen sind nicht äquivalent im Sinne der kürzesten Pfade.



# Umgewichtung

Andere Idee: “Potentialfunktion” (Höhe) auf den Knoten

- $G = (V, E, c)$  ein gewichteter Graph.
- Funktion  $h : V \rightarrow \mathbb{R}$
- Neue Gewichte

$$\tilde{c}(u, v) = c(u, v) + h(u) - h(v), \quad (u, v \in V)$$

# Umgewichtung

**Beobachtung:** Ein Pfad  $p$  ist genau dann kürzester Pfad in  $G = (V, E, c)$ , wenn er in  $\tilde{G} = (V, E, \tilde{c})$  kürzester Pfad ist.

$$\begin{aligned}\tilde{c}(p) &= \sum_{i=1}^k \tilde{c}(v_{i-1}, v_i) = \sum_{i=1}^k c(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i) \\ &= h(v_0) - h(v_k) + \sum_{i=1}^k c(v_{i-1}, v_i) = c(p) + h(v_0) - h(v_k)\end{aligned}$$

Also  $\tilde{c}(p)$  minimal unter allen  $v_0 \rightsquigarrow v_k \iff c(p)$  minimal unter allen  $v_0 \rightsquigarrow v_k$ .

Zyklengewichte sind invariant:  $\tilde{c}(v_0, \dots, v_k = v_0) = c(v_0, \dots, v_k = v_0)$



# Johnsons Algorithmus

Hinzunahme eines neuen Knotens  $s \notin V$ :

$$G' = (V', E', c')$$

$$V' = V \cup \{s\}$$

$$E' = E \cup \{(s, v) : v \in V\}$$

$$c'(u, v) = c(u, v), \quad u \neq s$$

$$c'(s, v) = 0 (v \in V)$$

# Johnsons Algorithmus

Falls keine negativen Zyklen: wähle für Höhenfunktion Gewicht der kürzesten Pfade von  $s$ ,

$$h(v) = d(s, v).$$

Für minimales Gewicht  $d$  eines Pfades gilt generell folgende Dreiecksungleichung:

$$d(s, v) \leq d(s, u) + c(u, v).$$

Einsetzen ergibt  $h(v) \leq h(u) + c(u, v)$ . Damit

$$\tilde{c}(u, v) = c(u, v) + h(u) - h(v) \geq 0.$$

# Algorithmus Johnson( $G$ )

**Input:** Gewichteter Graph  $G = (V, E, c)$

**Output:** Minimale Gewichte aller Pfade  $D$ .

Neuer Knoten  $s$ . Berechne  $G' = (V', E', c')$

**if** BellmanFord( $G', s$ ) = false **then** return “graph has negative cycles”

**foreach**  $v \in V'$  **do**

$h(v) \leftarrow d(s, v)$  //  $d$  aus BellmanFord Algorithmus

**foreach**  $(u, v) \in E'$  **do**

$\tilde{c}(u, v) \leftarrow c(u, v) + h(u) - h(v)$

**foreach**  $u \in V$  **do**

$\tilde{d}(u, \cdot) \leftarrow \text{Dijkstra}(\tilde{G}', u)$

**foreach**  $v \in V$  **do**

$D(u, v) \leftarrow \tilde{d}(u, v) + h(v) - h(u)$

# Analyse

## Laufzeiten

- Berechnung von  $G'$ :  $\mathcal{O}(|V|)$
- Bellman Ford  $G'$ :  $\mathcal{O}(|V| \cdot |E|)$
- $|V| \times$  Dijkstra  $\mathcal{O}(|V| \cdot |E| \cdot \log |V|)$   
(Mit Fibonacci-Heap:  $\mathcal{O}(|V|^2 \log |V| + |V| \cdot |E|)$ )

Insgesamt  $\mathcal{O}(|V| \cdot |E| \cdot \log |V|)$   
( $\mathcal{O}(|V|^2 \log |V| + |V| \cdot |E|)$ )

# 25. Minimale Spannbäume

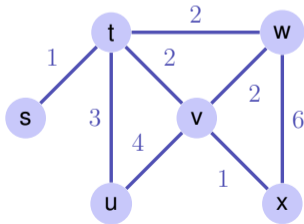
Motivation, Greedy, Algorithmus von Kruskal, Allgemeine Regeln, Union-Find Struktur, Algorithmus von Jarnik, Prim, Dijkstra, ,Algorithmus von Jarnik, Prim, Dijkstra ,Fibonacci Heaps

[Ottman/Widmayer, Kap. 9.6, 6.2, 6.1, Cormen et al, Kap. 23, 19]

# Problem

**Gegeben:** Ungerichteter, zusammenhängender, gewichteter Graph  $G = (V, E, c)$ .

**Gesucht:** Minimaler Spannbaum  $T = (V, E')$ : zusammenhängender, zyklensfreier Teilgraph  $E' \subset E$ , so dass  $\sum_{e \in E'} c(e)$  minimal.



# Beispiele von Anwendungen

- Netzwerk-Design: finde das billigste / kürzeste Netz oder Leitungssystem, welches alle Knoten miteinander verbindet.
- Approximation einer Lösung des Travelling-Salesman Problems: finde einen möglichst kurzen Rundweg, welcher jeden Knoten einmal besucht.

# Greedy Verfahren

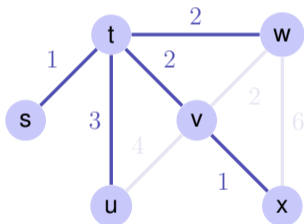
Zur Erinnerung:

- Gierige Verfahren berechnen eine Lösung schrittweise, indem lokal beste Lösungen gewählt werden.
- Die meisten Probleme sind nicht mit einer greedy Strategie lösbar.
- Das Problem des Minimalen Spannbaumes kann mit einem gierigen Verfahren effizient gelöst werden.



# Greedy Idee (Kruskal, 1956)

Konstruiere  $T$  indem immer die billigste Kante hinzugefügt wird, welche keinen Zyklus erzeugt.



(Lösung ist nicht eindeutig.)

# Algorithmus MST-Kruskal( $G$ )

**Input:** Gewichteter Graph  $G = (V, E, c)$

**Output:** Minimaler Spannbaum mit Kanten  $A$ .

Sortiere Kanten nach Gewicht  $c(e_1) \leq \dots \leq c(e_m)$

$A \leftarrow \emptyset$

**for**  $k = 1$  **to**  $|E|$  **do**

**if**  $(V, A \cup \{e_k\})$  kreisfrei **then**  
         $A \leftarrow A \cup \{e_k\}$

**return**  $(V, A, c)$

# Korrektheit

Zu jedem Zeitpunkt ist  $(V, A)$  ein Wald, eine Menge von Bäumen.  
MST-Kruskal betrachtet jede Kante  $e_k$  einmal und wählt  $e_k$  oder verwirft  $e_k$

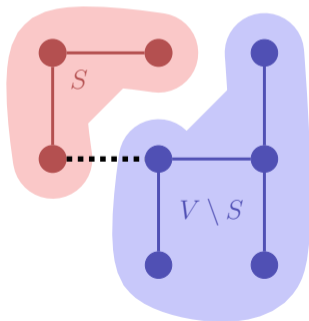
Notation (Momentaufnahme im Algorithmus)

- $A$ : Menge gewählte Kanten
- $R$ : Menge verworfener Kanten
- $U$ : Menge der noch unentschiedenen Kanten

# Schnitt

Ein Schnitt von  $G$  ist eine Partition  $S, V \setminus S$  von  $V$ . ( $S \subseteq V$ ).

Eine Kante kreuzt einen Schnitt, wenn einer Ihrer Endpunkte in  $S$  und der andere in  $V \setminus S$  liegt.



# Regeln

- 1 Auswahlregel: Wähle einen Schnitt, den keine gewählte Kante kreuzt. Unter allen unentschiedenen Kanten, welche den Schnitt kreuzen, wähle die mit minimalem Gewicht.
- 2 Verwerfregel: Wähle einen Kreis ohne verworfene Kanten. Unter allen unentschiedenen Kanten im Kreis verwerfe die mit maximalem Gewicht.

# Regeln

Kruskal wendet beide Regeln an:

- 1 Ein gewähltes  $e_k$  verbindet zwei Zusammenhangskomponenten, sonst würde ein Kreis erzeugt werden.  $e_k$  ist beim Verbinden minimal, man kann also einen Schnitt wählen, den  $e_k$  mit minimalem Gewicht kreuzt.
- 2 Ein verworfenes  $e_k$  ist Teil eines Kreises. Innerhalb des Kreises hat  $e_k$  maximales Gewicht.

## Theorem

*Jeder Algorithmus, welcher schrittweise obige Regeln anwendet bis  $U = \emptyset$  ist korrekt.*

Folgerung: MST-Kruskal ist korrekt.

# Auswahlinvariante

*Invariante:* Es gibt stets einen minimalen Spannbaum, der alle gewählten und keine der verworfenen Kanten enthält.

Wenn die beiden Regeln die Invariante erhalten, dann ist der Algorithmus sicher korrekt. Induktion:

- Zu Beginn:  $U = E, R = A = \emptyset$ . Invariante gilt offensichtlich.
- Invariante bleibt nach jedem Schritt des Algorithmus erhalten.
- Am Ende:  $U = \emptyset, R \cup A = E \Rightarrow (V, A)$  ist Spannbaum.

Beweis des Theorems: zeigen nun, dass die beiden Regeln die Invariante erhalten.



# Auswahlregel erhält Invariante

Es gibt stets einen minimalen Spannbaum  $T$ , der alle gewählten und keine der verworfenen Kanten enthält.

Wähle einen Schnitt, den keine gewählte Kante kreuzt. Unter allen unentschiedenen Kanten, welche den Schnitt kreuzen, wähle eine Kante  $e$  mit minimalem Gewicht.

- Fall 1:  $e \in T$  (fertig)
- Fall 2:  $e \notin T$ . Dann hat  $T \cup \{e\}$  einen Kreis, der  $e$  enthält. Kreis muss eine zweite Kante  $e'$  enthalten, welche den Schnitt auch kreuzt.<sup>51</sup> Da  $e' \notin R$  ist  $e' \in U$ . Somit  $c(e) \leq c(e')$  und  $T' = T \setminus \{e'\} \cup \{e\}$  ist auch minimaler Spannbaum (und  $c(e) = c(e')$ ).

---

<sup>51</sup>Ein solcher Kreis enthält mindestens einen Knoten in  $S$  und einen in  $V \setminus S$  und damit mindestens zwei Kanten zwischen  $S$  und  $V \setminus S$ .

# Verwerfregel erhält Invariante

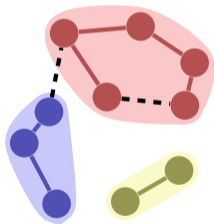
Es gibt stets einen minimalen Spannbaum  $T$ , der alle gewählten und keine der verworfenen Kanten enthält.

Wähle einen Kreis ohne verworfene Kanten. Unter allen unentschiedenen Kanten im Kreis verwerfe die Kante  $e$  mit maximalem Gewicht.

- Fall 1:  $e \notin T$  (fertig)
- Fall 2:  $e \in T$ . Entferne  $e$  von  $T$ , Das ergibt einen Schnitt. Diesen Schnitt muss eine weitere Kante  $e'$  aus dem Kreis kreuzen. Da  $c(e') \leq c(e)$  ist  $T' = T \setminus \{e\} \cup \{e'\}$  auch minimal (und  $c(e) = c(e')$ ).

# Zur Implementation

Gegeben eine Menge von Mengen  $i \equiv A_i \subset V$ . Zur Identifikation von Schnitten und Kreisen: Zugehörigkeit der beiden Endpunkte einer Kante zu einer der Mengen.



# Zur Implementation

Allgemeines Problem: Partition (Menge von Teilmengen) z.B.  
 $\{\{1, 2, 3, 9\}, \{7, 6, 4\}, \{5, 8\}, \{10\}\}$

Benötigt: Abstrakter Datentyp „Union-Find“ mit folgenden Operationen

- $\text{Make-Set}(i)$ : Hinzufügen einer neuen Menge  $i$ .
- $\text{Find}(e)$ : Name  $i$  der Menge, welche  $e$  enthält.
- $\text{Union}(i, j)$ : Vereinigung der Mengen mit Namen  $i$  und  $j$ .

# Union-Find Algorithmus MST-Kruskal( $G$ )

**Input:** Gewichteter Graph  $G = (V, E, c)$

**Output:** Minimaler Spannbaum mit Kanten  $A$ .

Sortiere Kanten nach Gewicht  $c(e_1) \leq \dots \leq c(e_m)$

$A \leftarrow \emptyset$

**for**  $k = 1$  **to**  $|V|$  **do**

$\lfloor$  MakeSet( $k$ )

**for**  $k = 1$  **to**  $m$  **do**

$(u, v) \leftarrow e_k$

**if** Find( $u$ )  $\neq$  Find( $v$ ) **then**

    Union(Find( $u$ ), Find( $v$ ))

$A \leftarrow A \cup e_k$

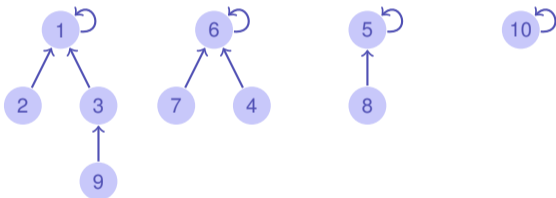
**else**

// konzeptuell:  $R \leftarrow R \cup e_k$

**return**  $(V, A, c)$

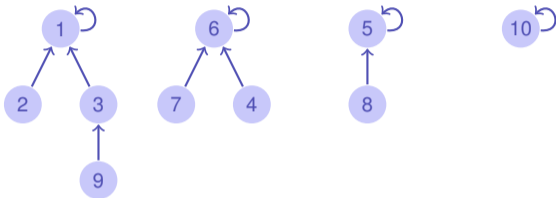
# Implementation Union-Find

Idee: Baum für jede Teilmenge in der Partition, z.B.  
 $\{\{1, 2, 3, 9\}, \{7, 6, 4\}, \{5, 8\}, \{10\}\}$



Baumwurzeln = Namen (Stellvertreter) der Mengen,  
Bäume = Elemente der Mengen

# Implementation Union-Find



Repräsentation als Array:

Index	1	2	3	4	5	6	7	8	9	10
Parent	<b>1</b>	1	1	6	<b>5</b>	<b>6</b>	5	5	3	<b>10</b>

# Implementation Union-Find

Index	1	2	3	4	5	6	7	8	9	10
Parent	<b>1</b>	1	1	6	<b>5</b>	<b>6</b>	5	5	3	<b>10</b>

---

Make-Set( $i$ )     $p[i] \leftarrow i$ ; **return**  $i$

---

Find( $i$ )        **while** ( $p[i] \neq i$ ) **do**  $i \leftarrow p[i]$   
                  **return**  $i$

---

Union( $i, j$ )<sup>52</sup>     $p[j] \leftarrow i$ ;

---

<sup>52</sup> $i$  und  $j$  müssen Namen (Wurzeln) der Mengen sein. Andernfalls verwende Union(Find( $i$ ),Find( $j$ ))



# Optimierung der Laufzeit für Find

Baum kann entarten: Beispiel Union(8, 7), Union(7, 6), Union(6, 5), ...

Index	1	2	3	4	5	6	7	8	..
Parent	<b>1</b>	1	2	3	4	5	6	7	..

Laufzeit von Find im schlechtesten Fall in  $\Theta(n)$ .

# Optimierung der Laufzeit für Find

Idee: Immer kleineren Baum unter grösseren Baum hängen.  
Benötigt zusätzliche Grösseninformation (Array)  $g$

---

**Make-Set**( $i$ )     $p[i] \leftarrow i$ ;  $g[i] \leftarrow 1$ ; **return**  $i$

---

**Union**( $i, j$ )    **if**  $g[j] > g[i]$  **then**  $\text{swap}(i, j)$   
                   $p[j] \leftarrow i$   
                  **if**  $g[i] = g[j]$  **then**  $g[i] \leftarrow g[i] + 1$

---

⇒ Baumtiefe (und schlechteste Laufzeit für Find) in  $\Theta(\log n)$

# [Beobachtung]

## Theorem

*Obiges Verfahren Vereinigung nach Grösse konserviert die folgende Eigenschaft der Bäume: ein Baum mit Höhe  $h$  hat mindestens  $2^h$  Knoten.*

Unmittelbare Folgerung: Laufzeit Find =  $\mathcal{O}(\log n)$ .

# [Beweis]

Induktion: nach Voraussetzung haben Teilbäume jeweils mindestens  $2^{h_i}$  Knoten. ObdA:  
 $h_2 \leq h_1$ .

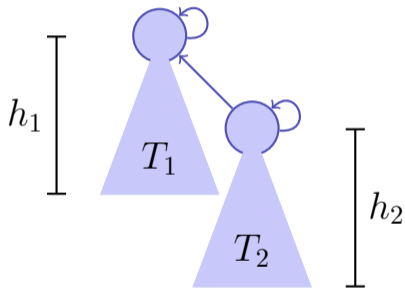
■  $h_2 < h_1$ :

$$h(T_1 \oplus T_2) = h_1 \Rightarrow g(T_1 \oplus T_2) \geq 2^{h_1}$$

■  $h_2 = h_1$ :

$$g(T_1) \geq g(T_2) \geq 2^{h_2}$$

$$\Rightarrow g(T_1 \oplus T_2) = g(T_1) + g(T_2) \geq 2 \cdot 2^{h_2} = 2^{h(T_1 \oplus T_2)}$$



# Weitere Verbesserung

Bei jedem Find alle Knoten direkt an den Wurzelknoten hängen.

Find( $i$ ):

$j \leftarrow i$

**while** ( $p[i] \neq i$ ) **do**  $i \leftarrow p[i]$

**while** ( $j \neq i$ ) **do**

$t \leftarrow j$   
 $j \leftarrow p[j]$   
 $p[t] \leftarrow i$

**return**  $i$

Laufzeit: amortisiert *fast* konstant (Inverse der Ackermannfunktion).<sup>53</sup>

---

<sup>53</sup>Wird hier nicht vertieft.

# Laufzeit des Kruskal Algorithmus

- Sortieren der Kanten:  $\Theta(|E| \log |E|) = \Theta(|E| \log |V|)$ .<sup>54</sup>
  - Initialisieren der Union-Find Datenstruktur  $\Theta(|V|)$
  - $|E| \times \text{Union}(\text{Find}(x), \text{Find}(y))$ :  $\mathcal{O}(|E| \log |E|) = \mathcal{O}(|E| \log |V|)$ .
- Insgesamt  $\Theta(|E| \log |V|)$ .

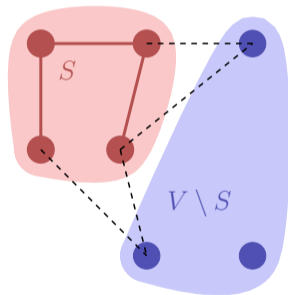
---

<sup>54</sup>da  $G$  zusammenhängend:  $|V| \leq |E| \leq |V|^2$

# Algorithmus von Jarnik (1930), Prim, Dijkstra (1959)

Idee: Starte mit einem  $v \in V$  und lasse von dort unter Verwendung der Auswahlregel einen Spannbaum wachsen:

```
A ← ∅  
S ← {v0}  
for i ← 1 to |V| do  
  Wähle billigste (u, v) mit u ∈ S, v ∉ S  
  A ← A ∪ {(u, v)}  
  S ← S ∪ {v} // (Färbung)
```



Anmerkung: man benötigt keine Union-Find Datenstruktur. Es genügt, Knoten zu färben, sobald sie zu  $S$  hinzugenommen werden.

# Laufzeit

Trivial  $\mathcal{O}(|V| \cdot |E|)$ .

Verbesserung (wie bei Dijkstras Kürzeste Pfade):

■ Mit Min-Heap, Kosten:

- Initialisierung (Knotenfärbung)  $\mathcal{O}(|V|)$
- $|V| \times \text{ExtractMin} = \mathcal{O}(|V| \log |V|)$ ,
- $|E| \times \text{Insert oder DecreaseKey} = \mathcal{O}(|E| \log |V|)$ ,

$\mathcal{O}(|E| \cdot \log |V|)$

■ Mit Fibonacci-Heap:  $\mathcal{O}(|E| + |V| \cdot \log |V|)$ .



# Fibonacci Heaps

Datenstruktur zur Verwaltung von Elementen mit Schlüsseln.  
Operationen

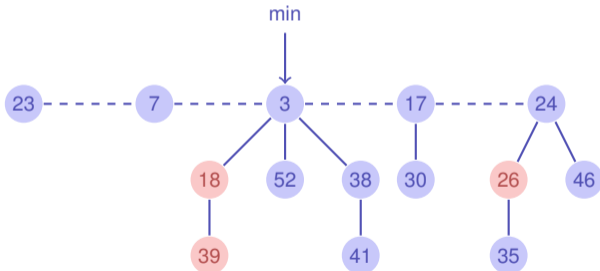
- **MakeHeap()**: Liefere neuen Heap ohne Elemente
- **Insert( $H, x$ )**: Füge  $x$  zu  $H$  hinzu
- **Minimum( $H$ )**: Liefere Zeiger auf das Element  $m$  mit minimalem Schlüssel
- **ExtractMin( $H$ )**: Liefere und entferne (von  $H$ ) Zeiger auf das Element  $m$
- **Union( $H_1, H_2$ )**: Liefere Verschmelzung zweier Heaps  $H_1$  und  $H_2$
- **DecreaseKey( $H, x, k$ )**: Verkleinere Schlüssel von  $x$  in  $H$  zu  $k$
- **Delete ( $H, x$ )**: Entferne Element  $x$  von  $H$

# Vorteil gegenüber Binary Heap?

	Binary Heap (worst-Case)	Fibonacci Heap (amortisiert)
MakeHeap	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$\Theta(1)$
ExtractMin	$\Theta(\log n)$	$\Theta(\log n)$
Union	$\Theta(n)$	$\Theta(1)$
DecreaseKey	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$\Theta(\log n)$

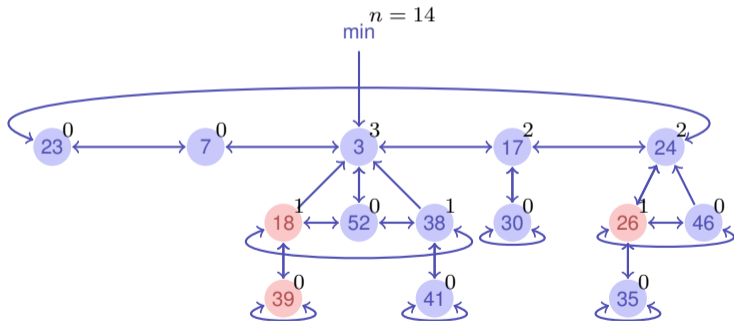
# Struktur

Menge von Bäumen, welche der Min-Heap Eigenschaft genügen.  
Markierbare Knoten.



# Implementation

Doppelt verkettete Listen von Knoten mit marked-Flag und Anzahl Kinder. Zeiger auf das minimale Element und Anzahl Knoten.



# Einfache Operationen

- MakeHeap (trivial)
- Minimum (trivial)
- Insert( $H, e$ )
  - 1 Füge neues Element in die Wurzelliste ein
  - 2 Wenn Schlüssel kleiner als Minimum, min-pointer neu setzen.
- Union ( $H_1, H_2$ )
  - 1 Wurzellisten von  $H_1$  und  $H_2$  aneinander hängen
  - 2 Min-Pointer neu setzen.
- Delete( $H, e$ )
  - 1 DecreaseKey( $H, e, -\infty$ )
  - 2 ExtractMin( $H$ )

# ExtractMin

- 1 Entferne Minimalknoten  $m$  aus der Wurzelliste
- 2 Hänge Liste der Kinder von  $m$  in die Wurzelliste
- 3 Verschmelze solange heapgeordnete Bäume gleichen Ranges, bis alle Bäume unterschiedlichen Rang haben:  
Rangarray  $a[0, \dots, n]$  von Elementen, zu Beginn leer. Für jedes Element  $e$  der Wurzelliste:
  - a Sei  $g$  der Grad von  $e$ .
  - b Wenn  $a[g] = nil$ :  $a[g] \leftarrow e$ .
  - c Wenn  $e' := a[g] \neq nil$ : Verschmelze  $e$  mit  $e'$  zu neuem  $e''$  und setze  $a[g] \leftarrow nil$ . Setze  $e''$  unmarkiert Iteriere erneut mit  $e \leftarrow e''$  vom Grad  $g + 1$ .

# DecreaseKey ( $H, e, k$ )

- 1 Entferne  $e$  von seinem Vaterknoten  $p$  (falls vorhanden) und erniedrige den Rang von  $p$  um eins.
- 2 Insert( $H, e$ )
- 3 Vermeide zu dünne Bäume:
  - a Wenn  $p = nil$ , dann fertig
  - b Wenn  $p$  unmarkiert: markiere  $p$ , fertig.
  - c Wenn  $p$  markiert: unmarkiere  $p$ , trenne  $p$  von seinem Vater  $pp$  ab und Insert( $H, p$ ). Iteriere mit  $p \leftarrow pp$ .

# Abschätzung für den Rang

## Theorem

*Sei  $p$  Knoten eines  $F$ -Heaps  $H$ . Ordnet man die Söhne von  $p$  in der zeitlichen Reihenfolge, in der sie an  $p$  durch Zusammenfügen angehängt wurden, so gilt: der  $i$ -te Sohn hat mindestens Rang  $i - 2$*

Beweis:  $p$  kann schon mehr Söhne gehabt haben und durch Abtrennung verloren haben. Als der  $i$ te Sohn  $p_i$  angehängt wurde, müssen  $p$  und  $p_i$  jeweils mindestens Rang  $i - 1$  gehabt haben.  $p_i$  kann maximal einen Sohn verloren haben (wegen Markierung), damit bleibt mindestens Rang  $i - 2$ .



# Abschätzung für den Rang

## Theorem

*Jeder Knoten  $p$  vom Rang  $k$  eines F-Heaps ist Wurzel eines Teilbaumes mit mindestens  $F_{k+1}$  Knoten. ( $F$ : Fibonacci-Folge)*

Beweis: Sei  $S_k$  Minimalzahl Nachfolger eines Knotens vom Rang  $k$  in einem F-Heap plus 1 (der Knoten selbst). Klar:  $S_0 = 1, S_1 = 2$ . Nach vorigem Theorem  $S_k \geq 2 + \sum_{i=0}^{k-2} S_i, k \geq 2$  ( $p$  und Knoten  $p_1$  jeweils 1). Für Fibonacci-Zahlen gilt (Induktion)  $F_k \geq 2 + \sum_{i=2}^k F_i, k \geq 2$  und somit (auch Induktion)  $S_k \geq F_{k+2}$ .

Fibonacci-Zahlen wachsen exponentiell ( $\mathcal{O}(\varphi^k)$ ) Folgerung: Maximaler Grad eines beliebigen Knotens im Fibonacci-Heap mit  $n$  Knoten ist  $\mathcal{O}(\log n)$ .

# Amortisierte Worst-case-Analyse Fibonacci Heap

$t(H)$ : Anzahl Bäume in der Wurzelliste von  $H$ ,  $m(H)$ : Anzahl markierte Knoten in  $H$  ausserhalb der Wurzelliste, Potentialfunktion  $\Phi(H) = t(H) + 2 \cdot m(H)$ . Zu Anfang  $\Phi(H) = 0$ . Potential immer nichtnegativ.

Amortisierte Kosten:

- **Insert**( $H, x$ ):  $t'(H) = t(H) + 1$ ,  $m'(H) = m(H)$ ,  
Potentialerhöhung 1, Amortisierte Kosten  $\Theta(1) + 1 = \Theta(1)$
- **Minimum**( $H$ ): Amortisierte Kosten = tatsächliche Kosten =  $\Theta(1)$
- **Union**( $H_1, H_2$ ): Amortisierte Kosten = tatsächliche Kosten =  $\Theta(1)$

# Amortisierte Kosten ExtractMin

- Anzahl der Bäume in der Wurzelliste  $t(H)$ .
- Tatsächliche Kosten der ExtractMin Operation:  $\mathcal{O}(\log n + t(H))$ .
- Nach dem Verschmelzen noch  $\mathcal{O}(\log n)$  Knoten.
- Anzahl der Markierungen kann beim Verschmelzen der Bäume maximal kleiner werden.
- Amortisierte Kosten von ExtractMin also maximal

$$\mathcal{O}(\log n + t(H)) + \mathcal{O}(\log n) - \mathcal{O}(t(H)) = \mathcal{O}(\log n).$$

# Amortisierte Kosten DecreaseKey

- Annahme: DecreaseKey führt zu  $c$  Abtrennungen eines Knotens von seinem Vaterknoten, tatsächliche Kosten  $\mathcal{O}(c)$
- $c$  Knoten kommen zur Wurzelliste hinzu
- Löschen von  $(c - 1)$  Markierungen, Hinzunahme maximal einer Markierung
- Amortisierte Kosten von DecreaseKey:

$$\mathcal{O}(c) + (t(H) + c) + 2 \cdot (m(H) - c + 2) - (t(H) + 2m(H)) = \mathcal{O}(1)$$

## 26. Flüsse in Netzen

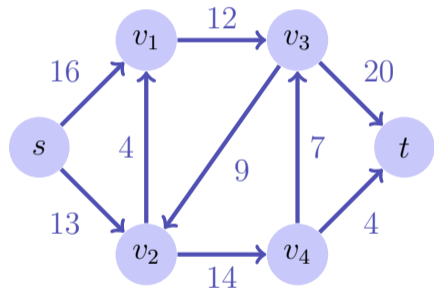
Flussnetzwerk, Maximaler Fluss, Schnitt, Restnetzwerk, Max-flow  
Min-cut Theorem, Ford-Fulkerson Methode, Edmonds-Karp  
Algorithmus, Maximales Bipartites Matching [Ottman/Widmayer,  
Kap. 9.7, 9.8.1], [Cormen et al, Kap. 26.1-26.3]

# Motivation

- Modelliere Fluss von Flüssigkeiten, Bauteile auf Fließbändern, Strom in elektrischen Netzwerken oder Information in Kommunikationsnetzwerken.
- Konnektivität von Kommunikationsnetzwerken, Bipartites Matching, Zirkulationen, Scheduling, Image Segmentation, Baseball Elimination...

# Flussnetzwerk

- **Flussnetzwerk**  $G = (V, E, c)$ : gerichteter Graph mit **Kapazitäten**
- Antiparallele Kanten verboten:  $(u, v) \in E \Rightarrow (v, u) \notin E$ .
- Fehlen einer Kante  $(u, v)$  auch modelliert durch  $c(u, v) = 0$ .
- **Quelle**  $s$  und **Senke**  $t$ : spezielle Knoten. Jeder Knoten  $v$  liegt auf einem Pfad zwischen  $s$  und  $t$ :  
 $s \rightsquigarrow v \rightsquigarrow t$

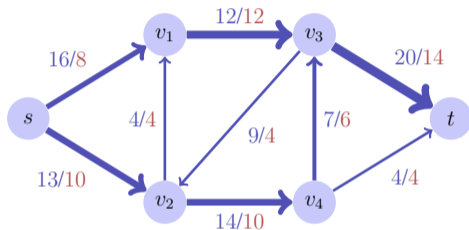


# Fluss

Ein *Fluss*  $f : V \times V \rightarrow \mathbb{R}$  erfüllt folgende Bedingungen:

- **Kapazitätsbeschränkung:**  
Für alle  $u, v \in V$ :  $f(u, v) \leq c(u, v)$ .
- **Schiefsymmetrie:**  
Für alle  $u, v \in V$ :  $f(u, v) = -f(v, u)$ .
- **Flusserhaltung:**  
Für alle  $u \in V \setminus \{s, t\}$ :

$$\sum_{v \in V} f(u, v) = 0.$$



**Wert  $w$  des Flusses:**

$$|f| = \sum_{v \in V} f(s, v).$$

Hier  $|f| = 18$ .



# Wie gross kann ein Fluss sein?

Begrenzende Faktoren: Schnitte

- *s* von *t* trennender Schnitt: Partitionierung von  $V$  in  $S$  und  $T$  mit  $s \in S, t \in T$ .
- *Kapazität* eines Schnittes:  $c(S, T) = \sum_{v \in S, v' \in T} c(v, v')$
- *Minimaler Schnitt*: Schnitt mit minimaler Kapazität.
- *Fluss über Schnitt*:  $f(S, T) = \sum_{v \in S, v' \in T} f(v, v')$

# Implizites Summieren

Notation: Seien  $U, U' \subseteq V$

$$f(U, U') := \sum_{\substack{u \in U \\ u' \in U'}} f(u, u'), \quad f(u, U') := f(\{u\}, U')$$

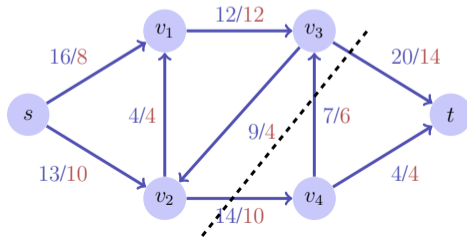
Somit

- $|f| = f(s, V)$
- $f(U, U) = 0$
- $f(U, U') = -f(U', U)$
- $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ , wenn  $X \cap Y = \emptyset$ .
- $f(R, V) = 0$  wenn  $R \cap \{s, t\} = \emptyset$ . [Flusserhaltung!]

# Wie gross kann ein Fluss sein?

Es gilt für jeden Fluss und jeden Schnitt, dass  $f(S, T) = |f|$ :

$$\begin{aligned} f(S, T) &= f(S, V) - \underbrace{f(S, S)}_0 = f(S, V) \\ &= f(s, V) + \underbrace{f(S - \{s\}, V)}_{\not\equiv t, \not\equiv s} = |f|. \end{aligned}$$

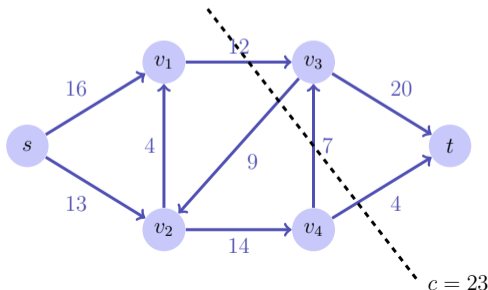


# Maximaler Fluss ?

Es gilt insbesondere für alle Schnitte  $(S, T)$  von  $V$ .

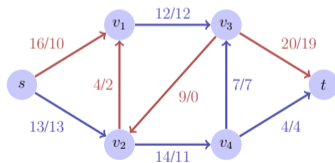
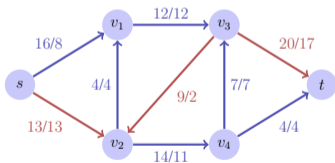
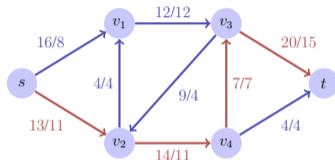
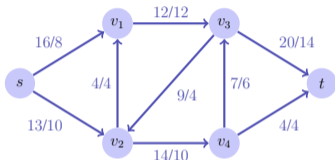
$$|f| \leq \sum_{v \in S, v' \in T} c(v, v') = c(S, T)$$

Werden sehen, dass Gleichheit gilt für  $\min_{S, T} c(S, T)$ .



# Maximaler Fluss ?

Naives Vorgehen:



Folgerung: Greedy Flussserhöhung löst das Problem nicht.

# Die Ford-Fulkerson Methode

- Starte mit  $f(u, v) = 0$  für alle  $u, v \in V$
- Bestimme Restnetzwerk\*  $G_f$  und Erweiterungspfad in  $G_f$
- Erhöhe Fluss über den Erweiterungspfad\*
- Wiederholung bis kein Erweiterungspfad mehr vorhanden.

$$G_f := (V, E_f, c_f)$$
$$c_f(u, v) := c(u, v) - f(u, v) \quad \forall u, v \in V$$
$$E_f := \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$$

\*Wird im Folgenden erklärt

# Flusserhöhung, negativ

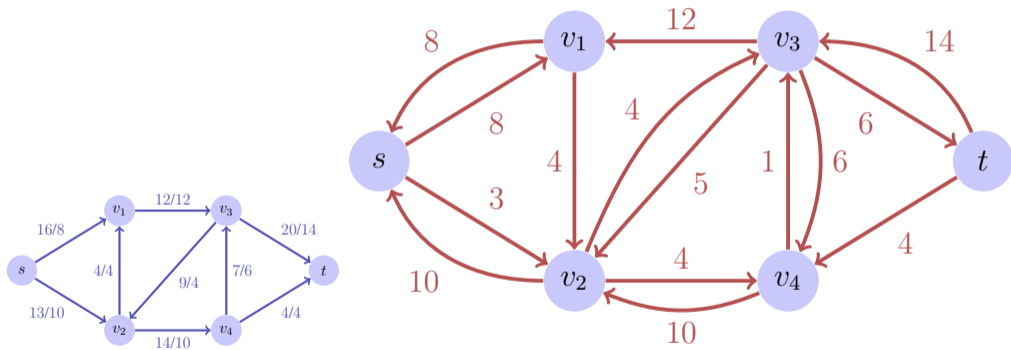
Sei ein Fluss  $f$  im Netzwerk gegeben.

Erkenntnis:

- Flusserhöhung in Richtung einer Kante möglich, wenn Fluss entlang der Kante erhöht werden kann, also wenn  $f(u, v) < c(u, v)$ .  
Restkapazität  $c_f(u, v) = c(u, v) - f(u, v) > 0$ .
- Flusserhöhung *entgegen* der Kantenrichtung möglich, wenn Fluss entlang der Kante verringert werden kann, also wenn  $f(u, v) > 0$ .  
Restkapazität  $c_f(v, u) = f(u, v) > 0$ .

# Restnetzwerk

*Restnetzwerk*  $G_f$  gegeben durch alle Kanten mit Restkapazität:



Restnetzwerke haben dieselben Eigenschaften wie Flussnetzwerke, ausser dass antiparallele Kapazitäten-Kanten zugelassen sind.



# Beobachtung

## Theorem

*Sei  $G = (V, E, c)$  ein Flussnetzwerk mit Quelle  $s$  und Senke  $t$  und  $f$  ein Fluss in  $G$ . Sei  $G_f$  das dazugehörige Restnetzwerk und sei  $f'$  ein Fluss in  $G_f$ . Dann definiert  $f \oplus f'$  mit*

$$(f \oplus f')(u, v) = f(u, v) + f'(u, v)$$

*einen Fluss in  $G$  mit Wert  $|f| + |f'|$ .*

# Beweis

$f \oplus f'$  ist ein Fluss in  $G$ :

## ■ Kapazitätsbeschränkung

$$(f \oplus f')(u, v) = f(u, v) + \underbrace{f'(u, v)}_{\leq c(u, v) - f(u, v)} \leq c(u, v)$$

## ■ Schiefsymmetrie

$$(f \oplus f')(u, v) = -f(v, u) + -f'(v, u) = -(f \oplus f')(v, u)$$

## ■ Flusserhaltung $u \in V - \{s, t\}$ :

$$\sum_{v \in V} (f \oplus f')(u, v) = \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) = 0$$

# Beweis

Wert von  $f \oplus f'$

$$\begin{aligned} |f \oplus f'| &= (f \oplus f')(s, V) \\ &= \sum_{u \in V} f(s, u) + f'(s, u) \\ &= f(s, V) + f'(s, V) \\ &= |f| + |f'| \end{aligned}$$



# Erweiterungspfade

*Erweiterungspfad*  $p$ : einfacher Pfad von  $s$  nach  $t$  im Restnetzwerk  $G_f$ .

*Restkapazität*  $c_f(p) = \min\{c_f(u, v) : (u, v) \text{ Kante in } p\}$

# Fluss in $G_f$

## Theorem

Die Funktion  $f_p : V \times V \rightarrow \mathbb{R}$ ,

$$f_p(u, v) = \begin{cases} c_f(p) & \text{wenn } (u, v) \text{ Kante in } p \\ -c_f(p) & \text{wenn } (v, u) \text{ Kante in } p \\ 0 & \text{sonst} \end{cases}$$

ist ein Fluss in  $G_f$  mit dem Wert  $|f_p| = c_f(p) > 0$ .

$f_p$  ist ein Fluss (leicht nachprüfbar). Es gibt genau einen Knoten  $u \in V$  mit  $(s, u) \in p$ . Somit  $|f_p| = \sum_{v \in V} f_p(s, v) = f_p(s, u) = c_f(p)$ .

# Folgerung

Strategie für den Algorithmus:

Mit einem Erweiterungspfad  $p$  in  $G_f$  definiert  $f \oplus f_p$  einen neuen Fluss mit Wert  $|f \oplus f_p| = |f| + |f_p| > |f|$ .

# Max-Flow Min-Cut Theorem

## Theorem

*Wenn  $f$  ein Fluss in einem Flussnetzwerk  $G = (V, E, c)$  mit Quelle  $s$  und Senke  $t$  ist, dann sind folgende Aussagen äquivalent:*

- 1  $f$  ist ein maximaler Fluss in  $G$*
- 2 Das Restnetzwerk  $G_f$  enthält keine Erweiterungspfade*
- 3 Es gilt  $|f| = c(S, T)$  für einen Schnitt  $(S, T)$  von  $G$ .*

# Beweis

- (3)  $\Rightarrow$  (1):

Es gilt  $|f| \leq c(S, T)$  für alle Schnitte  $S, T$ . Aus  $|f| = c(S, T)$  folgt also  $|f|$  maximal.

- (1)  $\Rightarrow$  (2):

$f$  maximaler Fluss in  $G$ . Annahme:  $G_f$  habe einen Erweiterungsfad. Dann gilt  $|f \oplus f_p| = |f| + |f_p| > |f|$ .  
Widerspruch.



## Beweis (2) $\Rightarrow$ (3)

Annahme:  $G_f$  habe keinen Erweiterungsfad

Definiere  $S = \{v \in V : \text{es existiert Pfad } s \rightsquigarrow v \text{ in } G_f\}$ .

$(S, T) := (S, V \setminus S)$  ist ein Schnitt:  $s \in S, t \in T$ .

Sei  $u \in S$  und  $v \in T$ . Dann  $c_f(u, v) = 0$ , also

$c_f(u, v) = c(u, v) - f(u, v) = 0$ . Somit  $f(u, v) = c(u, v)$ .

Somit

$$|f| = f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) = \sum_{u \in S} \sum_{v \in T} c(u, v) = C(S, T).$$



# Algorithmus Ford-Fulkerson( $G, s, t$ )

**Input:** Flussnetzwerk  $G = (V, E, c)$

**Output:** Maximaler Fluss  $f$ .

**for**  $(u, v) \in E$  **do**

└  $f(u, v) \leftarrow 0$

**while** Existiert Pfad  $p : s \rightsquigarrow t$  im Restnetzwerk  $G_f$  **do**

└  $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \in p\}$

└ **foreach**  $(u, v) \in p$  **do**

└└  $f(u, v) \leftarrow f(u, v) + c_f(p)$

└└  $f(v, u) \leftarrow f(v, u) - c_f(p)$

# Praktische Anmerkung

In einer Implementation des Ford-Fulkerson Algorithmus werden die negativen Flusskanten normalerweise nicht gespeichert, da ihr Wert sich stets als der negierter Wert der Gegenkante ergibt.

$$f(u, v) \leftarrow f(u, v) + c_f(p)$$

$$f(v, u) \leftarrow f(v, u) - c_f(p)$$

wird dann zu

**if**  $(u, v) \in E$  **then**

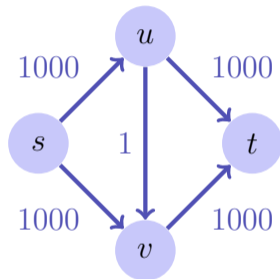
$$\quad | \quad f(u, v) \leftarrow f(u, v) + c_f(p)$$

**else**

$$\quad | \quad f(v, u) \leftarrow f(v, u) - c_f(p)$$

# Analyse

- Der Ford-Fulkerson Algorithmus muss für irrationale Kapazitäten nicht einmal terminieren! Für ganze oder rationale Zahlen terminiert der Algorithmus.
- Für ganzzahligen Fluss benötigt der Algorithmus maximal  $|f_{\max}|$  Durchläufe der While-Schleife (denn der Fluss erhöht sich mindestens um 1). Suche einzelner zunehmender Weg (z.B. Tiefensuche oder Breitensuche)  $\mathcal{O}(|E|)$ . Also  $\mathcal{O}(f_{\max}|E|)$ .



Bei schlecht gewählter Strategie benötigt der Algorithmus hier bis zu 2000 Iterationen.

# Edmonds-Karp Algorithmus

Wähle in der Ford-Fulkerson-Methode zum Finden eines Pfades in  $G_f$  jeweils einen Erweiterungspfad kürzester Länge (z.B. durch Breitensuche).

# Edmonds-Karp Algorithmus

## Theorem

*Wenn der Edmonds-Karp Algorithmus auf ein ganzzahliges Flussnetzwerk  $G = (V, E)$  mit Quelle  $s$  und Senke  $t$  angewendet wird, dann ist die Gesamtanzahl der durch den Algorithmus angewendete Flusserhöhungen in  $\mathcal{O}(|V| \cdot |E|)$ .*

*$\Rightarrow$  Gesamte asymptotische Laufzeit:  $\mathcal{O}(|V| \cdot |E|^2)$*

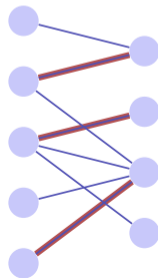
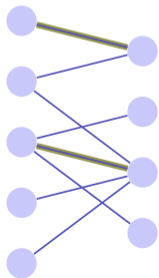
[Ohne Beweis]

# Anwendung: Maximales bipartites Matching

Gegeben: bipartiter ungerichteter Graph  $G = (V, E)$ .

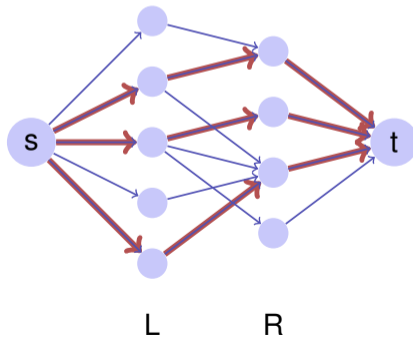
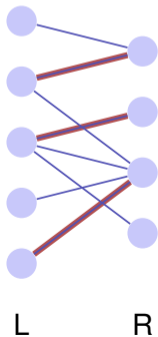
**Matching**  $M$ :  $M \subseteq E$  so dass  $|\{m \in M : v \in m\}| \leq 1$  für alle  $v \in V$ .

**Maximales Matching**  $M$ : Matching  $M$ , so dass  $|M| \geq |M'|$  für jedes Matching  $M'$ .



# Korrespondierendes Flussnetzwerk

Konstruiere zur einer Partition  $L, R$  eines bipartiten Graphen ein korrespondierendes Flussnetzwerk mit Quelle  $s$  und Senke  $t$ , mit gerichteten Kanten von  $s$  nach  $L$ , von  $L$  nach  $R$  und von  $R$  nach  $t$ . Jede Kante bekommt Kapazität 1.





# Ganzzahligkeitstheorem

## Theorem

*Wenn die Kapazitäten eines Flussnetzwerks nur ganzzahlige Werte annehmen, dann hat der durch Ford-Fulkerson erzeugte maximale Fluss die Eigenschaft, dass der Wert von  $f(u, v)$  für alle  $u, v \in V$  eine ganze Zahl ist.*

[ohne Beweis]

Folgerung: Ford Fulkerson erzeugt beim zum bipartiten Graph gehörenden Flussnetzwerk ein maximales Matching

$$M = \{(u, v) : f(u, v) = 1\}.$$

# 27. Parallel Programming I

Moore's Law und The Free Lunch, Hardware Architekturen, Parallele Ausführung, Klassifikation nach Flynn, Multi-Threading, Parallelität und Nebenläufigkeit, C++ Threads, Skalierbarkeit: Amdahl und Gustafson, Daten- und Taskparallelität, Scheduling

[Task-Scheduling: Cormen et al, Kap. 27] [Concurrency, Scheduling: Williams, Kap. 1.1 – 1.2]

# The Free Lunch

The free lunch is over<sup>55</sup>

---

<sup>55</sup>"The Free Lunch is Over", a fundamental turn toward concurrency in software, Herb Sutter, Dr. Dobb's Journal, 2005

# Moore's Law

Beobachtung von Gordon E. Moore:

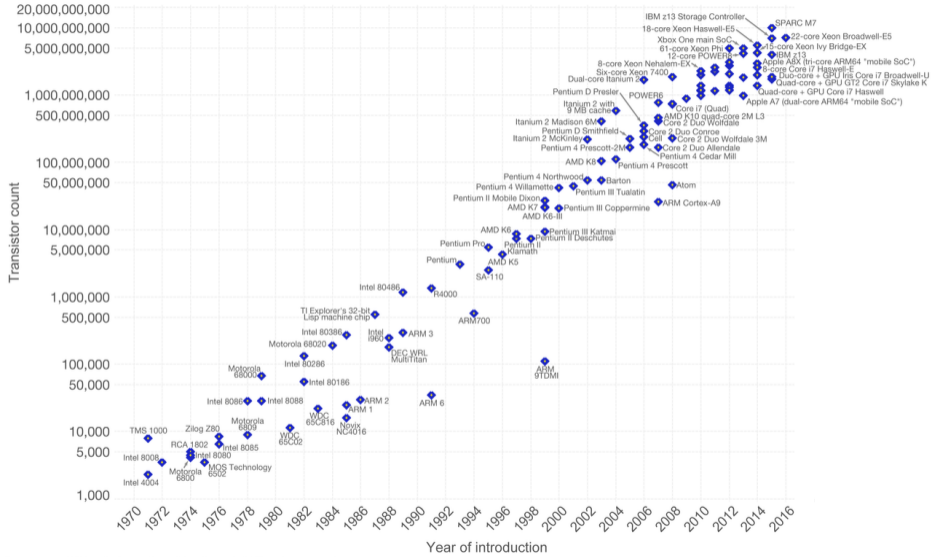
Die Anzahl Transistoren in integrierten Schaltkreisen verdoppelt sich ungefähr alle zwei Jahre.



Gordon E. Moore (1929)

# Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia ([https://en.wikipedia.org/wiki/Transistor\\_count](https://en.wikipedia.org/wiki/Transistor_count))  
The data visualization is available at [OurWorldInData.org](https://ourworldindata.org). There you find more visualizations and research on this topic.

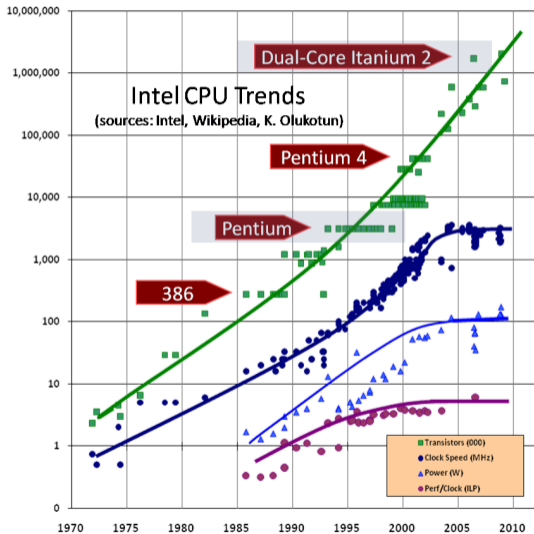
# Für eine lange Zeit...

- wurde die sequentielle Ausführung schneller ("Instruction Level Parallelism", "Pipelining", Höhere Frequenzen)
- mehr und kleinere Transistoren = mehr Performance
- Programmierer warteten auf die nächste schnellere Generation

# Heute

- steigt die Frequenz der Prozessoren kaum mehr an (Kühlproblematik)
- steigt die Instruction-Level Parallelität kaum mehr an
- ist die Ausführungsgeschwindigkeit in vielen Fällen dominiert von Speicherzugriffszeiten (Caches werden aber immer noch grösser und schneller)

# Trends





# Multicore

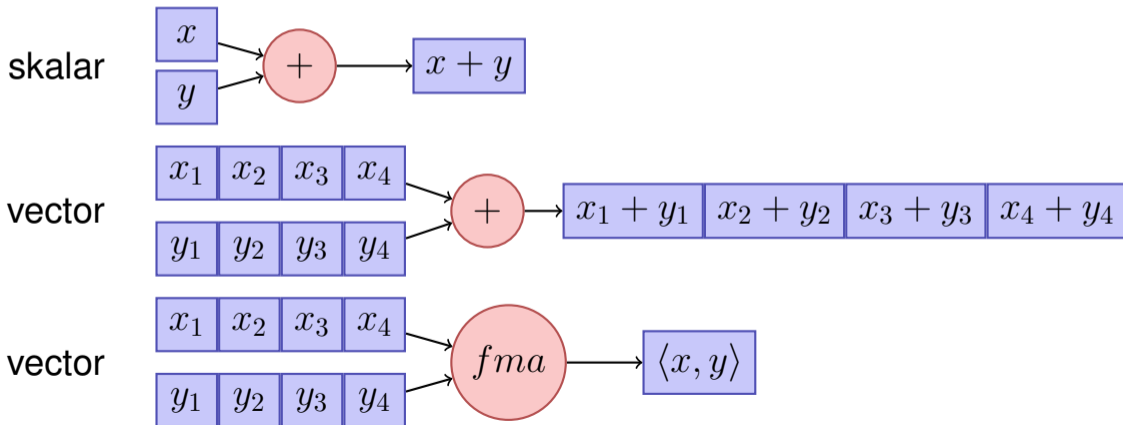
- Verwende die Transistoren für mehr Rechenkerne
- Parallelität in der Software
- Implikation: Programmierer müssen parallele Programme schreiben, um die neue Hardware vollständig ausnutzen zu können

# Formen der Parallelen Ausführung

- Vektorisierung
- Pipelining
- Instruction Level Parallelism
- Multicore / Multiprocessing
- Verteiltes Rechnen

# Vektorisierung

Parallele Ausführung derselben Operation auf Elementen eines Vektor(Register)s



# Pipelining in CPUs



Mehrere Stufen

- Jede Instruktion dauert 5 Zeiteinheiten (Zyklen)
- Im besten Fall: 1 Instruktion pro Zyklus, nicht immer möglich (“stalls”)

*Parallelität* (mehrere funktionale Einheiten) führt zu *schnellerer Ausführung*.

# ILP – Instruction Level Parallelism

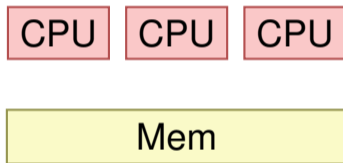
Moderne CPUs führen unabhängige Instruktionen intern auf mehreren Einheiten parallel aus

- Pipelining
- Superskalare CPUs (Mehrere Instruktionen pro Zyklus)
- Out-Of-Order Execution (Programmierer sieht die sequentielle Ausführung)
- Speculative Execution (Instruktionen werden spekulativ ausgeführt und unterbrochen, wenn die Bedingung zu deren Ausführung nicht erfüllt ist.)

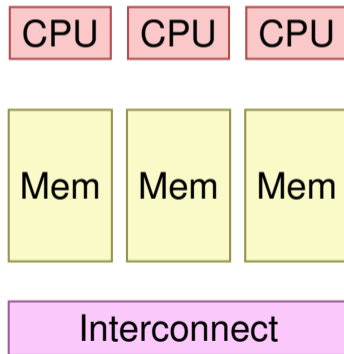
## **27.2 Hardware Architekturen**

# Gemeinsamer vs. verteilter Speicher

Gemeinsamer Speicher



Verteilter Speicher



# Shared vs. Distributed Memory Programming

- Kategorien des Programmierinterfaces
  - Kommunikation via Message Passing
  - Kommunikation via geteiltem Speicher
- Es ist möglich:
  - Systeme mit gemeinsamen Speicher als verteilte Systeme zu programmieren (z.B. mit Message Passing Interface MPI)
  - Systeme mit verteiltem Speicher als System mit gemeinsamen Speicher zu programmieren (z.B. Partitioned Global Address Space PGAS)

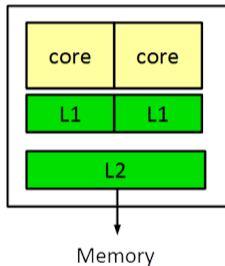


# Architekturen mit gemeinsamen Speicher

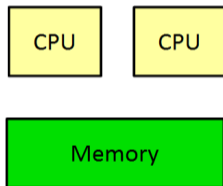
- Multicore (Chip Multiprocessor - CMP)
- Symmetric Multiprocessor Systems (SMP)
- Simultaneous Multithreading (SMT = Hyperthreading)
  - nur ein physischer Kern, Mehrere Instruktionsströme/Threads: mehrere virtuelle Kerne
  - Zwischen ILP (mehrere Units für einen Strom) und Multicore (mehrere Units für mehrere Ströme). Limitierte parallele Performance
- Non-Uniform Memory Access (NUMA)

Gleiches Programmierinterface!

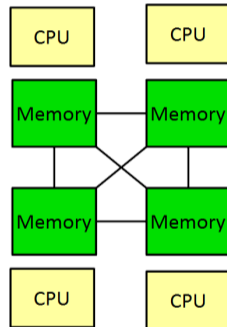
# Übersicht



CMP



SMP

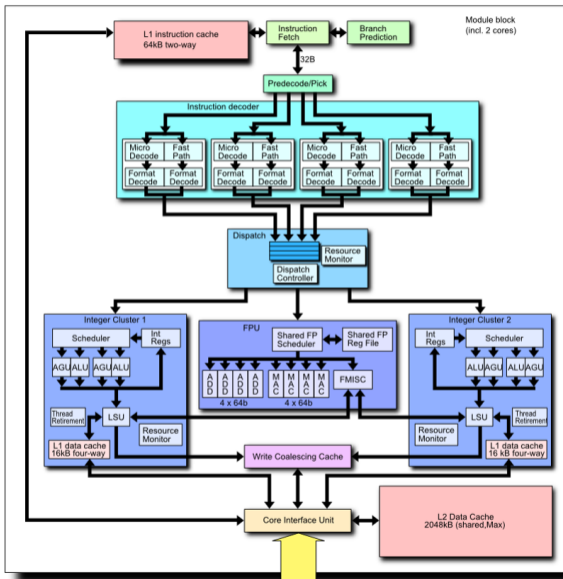


NUMA

# Ein Beispiel

AMD Bulldozer: Zwischen CMP und SMT

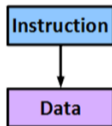
- 2x integer core
- 1x floating point core



# Klassifikation nach Flynn

## Single-Core

SISD

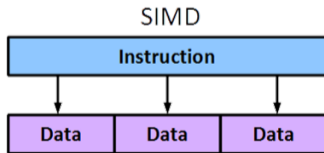
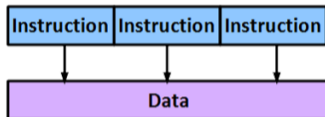


SI = Single Instruction  
MI = Multiple Instructions

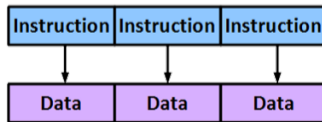
SD = Single Data  
MD = Multiple Data

## Fault-Tolerance

MISD



MIMD



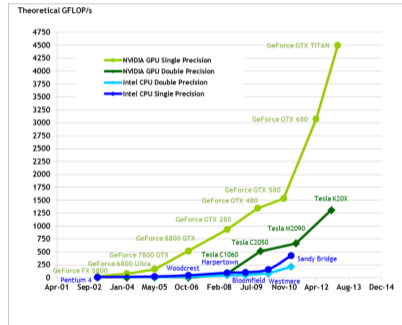
Vector Computing / GPU

Multi-Core

# Massiv Parallele Hardware

## [General Purpose] Graphical Processing Units ([GP]GPUs)

- Revolution im High Performance Computing
  - Calculation 4.5 TFlops vs. 500 GFlops
  - Memory Bandwidth 170 GB/s vs. 40 GB/s
- SIMD
  - Hohe Datenparallelität
  - Benötigt eigenes Programmiermodell.  
Z.B. CUDA / OpenCL



## **27.3 Multi-Threading, Parallelität und Nebenläufigkeit**

# Prozesse und Threads

- Prozess: Instanz eines Programmes
  - jeder Prozess hat seinen eigenen Kontext, sogar eigenen Adressraum
  - OS verwaltet Prozesse (Ressourcenkontrolle, Scheduling, Synchronisierung)
- Threads: Ausführungsfäden eines Programmes
  - Threads teilen sich einen Adressraum
  - Schneller Kontextwechsel zwischen Threads

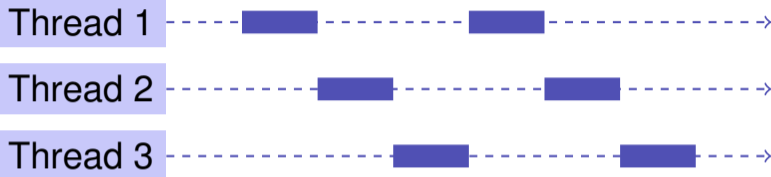
# Warum Multithreading?

- Verhinderung vom “Polling” auf Ressourcen (Files, Netzwerkzugriff, Tastatur)
- Interaktivität (z.B. Responsivität von GUI Programmen)
- Mehrere Applikationen / Clients gleichzeitig instanzierbar
- Parallelität (Performanz!)



# Multithreading konzeptuell

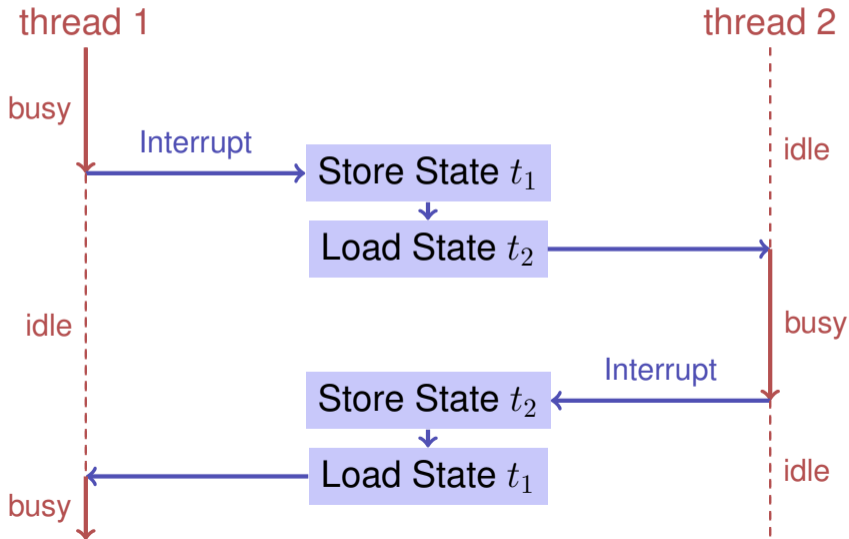
Single Core



Multi Core



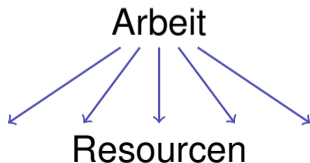
# Threadwechsel auf einem Core (Preemption)



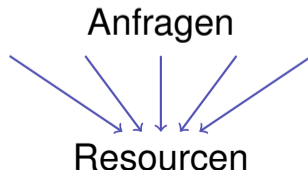
# Parallelität vs. Nebenläufigkeit (Concurrency)

- *Parallelität*: Verwende zusätzliche Ressourcen (z.B. CPUs), um ein Problem schneller zu lösen
- *Nebenläufigkeit*: Verwalte gemeinsam genutzte Ressourcen (z.B. Speicher) korrekt und effizient
- Begriffe überlappen offensichtlich. Bei parallelen Berechnungen besteht fast immer Synchronisierungsbedarf.

## Parallelität



## Nebenläufigkeit



# Thread-Sicherheit

Thread-Sicherheit bedeutet, dass in der nebenläufigen Anwendung eines Programmes dieses sich immer wie gefordert verhält.

Viele Optimierungen (Hardware, Compiler) sind darauf ausgerichtet, dass sich ein *sequentielles* Programm korrekt verhält.

Nebenläufige Programme benötigen für ihre Synchronisierungen auch eine Annotation, welche gewisse Optimierungen selektiv abschaltet

# Beispiel: Caches

- Speicherzugriff auf Register schneller als auf den gemeinsamen Speicher
- Prinzip der Lokalität
- Verwendung von Caches (transparent für den Programmierer)

Ob und wie weit die Cache-Kohärenz sichergestellt wird ist vom eingesetzten System abhängig.



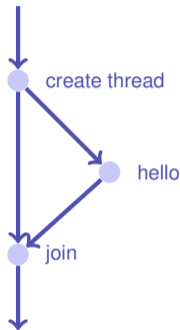
## 27.4 C++ Threads

# C++11 Threads

```
#include <iostream>
#include <thread>

void hello(){
    std::cout << "hello\n";
}

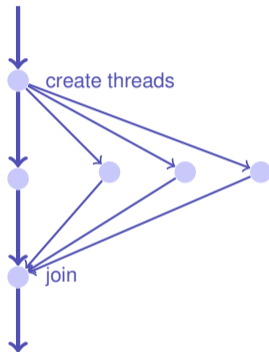
int main(){
    // create and launch thread t
    std::thread t(hello);
    // wait for termination of t
    t.join();
    return 0;
}
```



# C++11 Threads

```
void hello(int id){  
    std::cout << "hello from " << id << "\n";  
}
```

```
int main(){  
    std::vector<std::thread> tv(3);  
    int id = 0;  
    for (auto & t:tv)  
        t = std::thread(hello, ++id);  
    std::cout << "hello from main \n";  
    for (auto & t:tv)  
        t.join();  
    return 0;  
}
```





# Nichtdeterministische Ausführung!

Eine Ausführung:

hello from main  
hello from 2  
hello from 1  
hello from 0

Andere Ausführung:

hello from 1  
hello from main  
hello from 0  
hello from 2

Andere Ausführung:

hello from main  
hello from 0  
hello from hello from 1  
2

# Technisches Detail

Um einen Thread als Hintergrundthread weiterlaufen zu lassen:

```
void background();
```

```
void someFunction(){
```

```
    ...
```

```
    std::thread t(background);
```

```
    t.detach();
```

```
    ...
```

```
} // no problem here, thread is detached
```

# Mehr Technische Details

- Beim Erstellen von Threads werden auch Referenzparameter kopiert, ausser man gibt explizit `std::ref` bei der Konstruktion an.
- Funktoren oder Lambda-Expressions können auch auf einem Thread ausgeführt werden
- In einem Kontext mit Exceptions sollte das `join` auf einem Thread im `catch`-Block ausgeführt werden

Noch mehr Hintergründe im Kapitel 2 des Buches *C++ Concurrency in Action*, Anthony Williams, Manning 2012. Auch online bei der ETH Bibliothek erhältlich.

## **27.5 Skalierbarkeit: Amdahl und Gustafson**

# Skalierbarkeit

In der parallelen Programmierung:

- Geschwindigkeitssteigerung bei wachsender Anzahl  $p$  Prozessoren
- Was passiert, wenn  $p \rightarrow \infty$ ?
- Linear skalierendes Programm: Linearer Speedup

# Parallele Performanz

Gegeben fixierte Rechenarbeit  $W$  (Anzahl Rechenschritte)

Sequentielle Ausführungszeit sei  $T_1$

Parallele Ausführungszeit  $T_p$  auf  $p$  CPUs

- Perfektion:  $T_p = T_1/p$
- Performanzverlust:  $T_p > T_1/p$  (üblicher Fall)
- Hexerei:  $T_p < T_1/p$

# Paralleler Speedup

Paralleler Speedup  $S_p$  auf  $p$  CPUs:

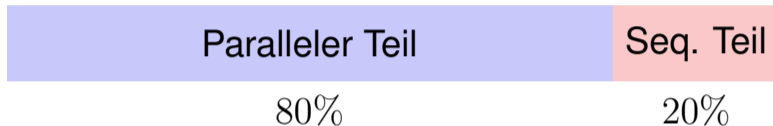
$$S_p = \frac{W/T_p}{W/T_1} = \frac{T_1}{T_p}.$$

- Perfektion: Linearer Speedup  $S_p = p$
- Verlust: sublinearer Speedup  $S_p < p$  (der übliche Fall)
- Hexerei: superlinearer Speedup  $S_p > p$

Effizienz:  $E_p = S_p/p$

# Erreichbarer Speedup?

Paralleles Programm



$$T_1 = 10$$

$$T_8 = \frac{10 \cdot 0.8}{8} + 10 \cdot 0.2 = 1 + 2 = 3$$

$$S_8 = \frac{T_1}{T_8} = \frac{10}{3} \approx 3.3 < 8 \quad (!)$$



# Amdahl's Law: Zutaten

Zu leistende Rechenarbeit  $W$  fällt in zwei Kategorien

- Parallelisierbarer Teil  $W_p$
- Nicht parallelisierbarer, sequentieller Teil  $W_s$

Annahme:  $W$  kann mit *einem* Prozessor in  $W$  Zeiteinheiten sequentiell erledigt werden ( $T_1 = W$ ):

$$T_1 = W_s + W_p$$

$$T_p \geq W_s + W_p/p$$

# Amdahl's Law

$$S_p = \frac{T_1}{T_p} \leq \frac{W_s + W_p}{W_s + \frac{W_p}{p}}$$

# Amdahl's Law

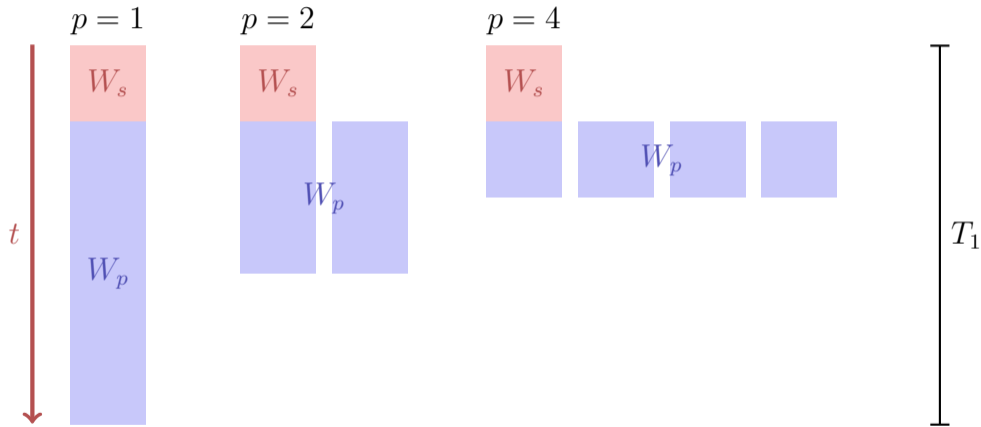
Mit seriellem, nicht parallelisierbarem Anteil  $\lambda$ :  $W_s = \lambda W$ ,  
 $W_p = (1 - \lambda)W$ :

$$S_p \leq \frac{1}{\lambda + \frac{1-\lambda}{p}}$$

Somit

$$S_\infty \leq \frac{1}{\lambda}$$

# Illustration Amdahl's Law



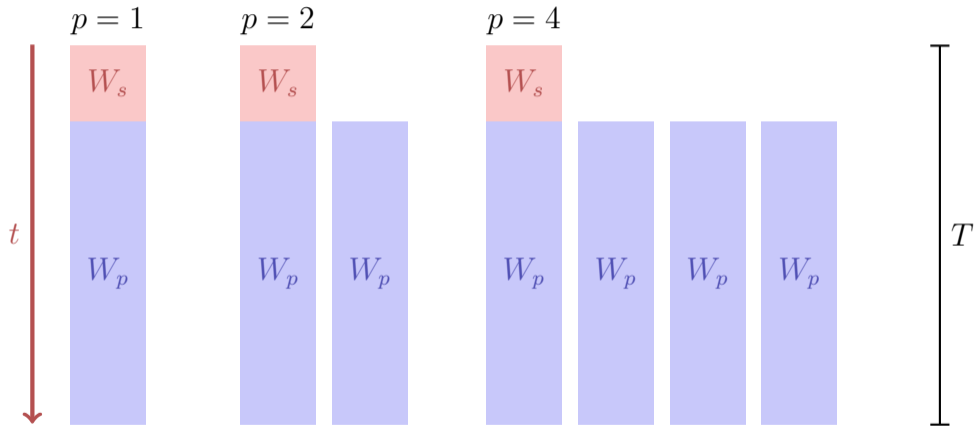
# Amdahl's Law ist keine gute Nachricht

Alle nicht parallelisierbaren Teile können Problem bereiten und stehen der Skalierbarkeit entgegen.

# Gustafson's Law

- Halte die Ausführungszeit fest.
- Variiere die Problemgrösse.
- Annahme: Der sequentielle Teil bleibt konstant, der parallele Teil wird grösser.

# Illustration Gustafson's Law



# Gustafson's Law

Arbeit, die mit einem Prozessor in der Zeit  $T$  erledigt werden kann:

$$W_s + W_p = T$$

Arbeit, die mit  $p$  Prozessoren in der Zeit  $T$  erledigt werden kann:

$$W_s + p \cdot W_p = \lambda \cdot T + p \cdot (1 - \lambda) \cdot T$$

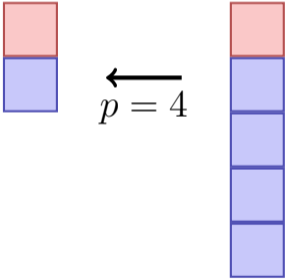
Speedup:

$$\begin{aligned} S_p &= \frac{W_s + p \cdot W_p}{W_s + W_p} = p \cdot (1 - \lambda) + \lambda \\ &= p - \lambda(p - 1) \end{aligned}$$

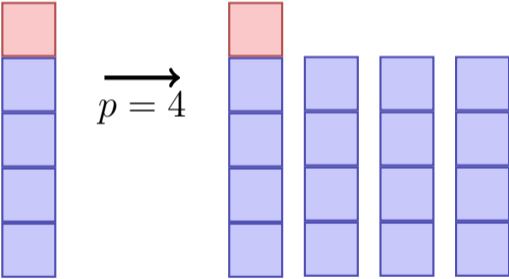


# Amdahl vs. Gustafson

Amdahl



Gustafson



# Amdahl vs. Gustafson

Die Gesetze von Amdahl und Gustafson sind Modelle der Laufzeitverbesserung bei Parallelisierung.

Amdahl geht von einem festen *relativen* sequentiellen Anteil der Arbeit aus, während Gustafson von einem festen *absoluten* sequentiellen Teil ausgeht (der als Bruchteil der Arbeit  $W_1$  ausgedrückt wird und bei Zunahme der Arbeit nicht wächst).

Die beiden Modelle widersprechen sich nicht, sondern beschreiben die Laufzeitverbesserung verschiedener Probleme und Algorithmen.

## **27.6 Task- und Datenparallelität**

# Paradigmen der Parallelen Programmierung

- *Task Parallel:* Programmierer legt parallele Tasks explizit fest.
- *Daten-Parallel:* Operationen gleichzeitig auf einer Menge von individuellen Datenobjekten.

# Beispiel Data Parallel (OMP)

```
double sum = 0, A[MAX];  
#pragma omp parallel for reduction (+:ave)  
for (int i = 0; i < MAX; ++i)  
    sum += A[i];  
return sum;
```

# Beispiel Task Parallel (C++11 Threads/Futures)

```
double sum(Iterator from, Iterator to)
{
    auto len = from - to;
    if (len > threshold){
        auto future = std::async(sum, from, from + len / 2);
        return sumS(from + len / 2, to) + future.get();
    }
    else
        return sumS(from, to);
}
```

# Partitionierung und Scheduling

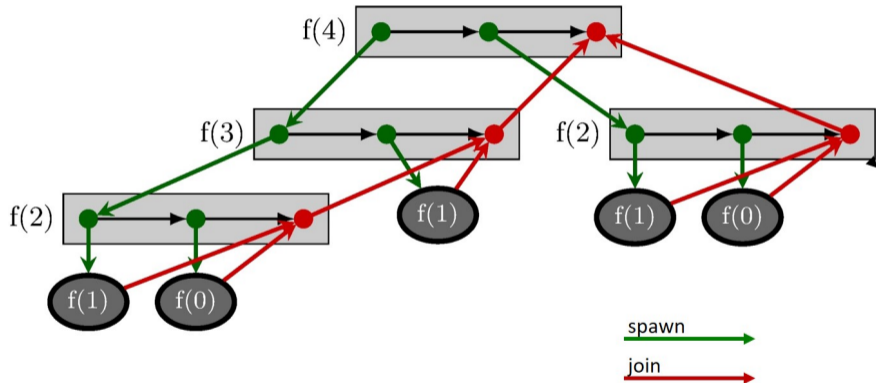
- Aufteilung der Arbeit in parallele Tasks (Programmierer oder System)
  - Ein Task ist eine Arbeitseinheit
  - Frage: welche Granularität?
- Scheduling (Laufzeitsystem)
  - Zuweisung der Tasks zu Prozessoren
  - Ziel: volle Ressourcennutzung bei wenig Overhead

# Beispiel: Fibonacci P-Fib

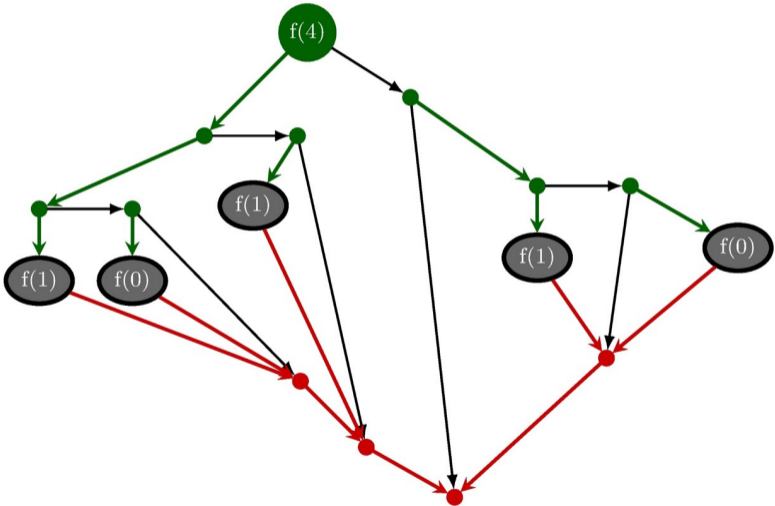
```
if  $n \leq 1$  then  
  | return  $n$   
else  
  |  $x \leftarrow$  spawn P-Fib( $n - 1$ )  
  |  $y \leftarrow$  spawn P-Fib( $n - 2$ )  
  | sync  
  | return  $x + y$ ;
```



# P-Fib Task Graph

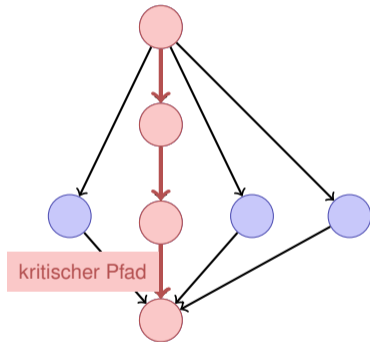


# P-Fib Task Graph



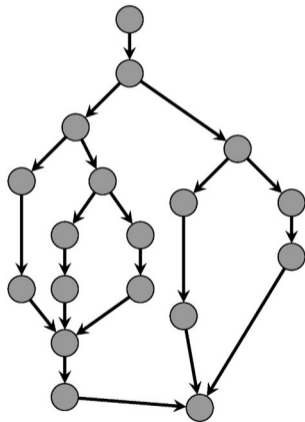
# Frage

- Jeder Knoten (Task) benötigt 1 Zeiteinheit.
- Pfeile bezeichnen Abhängigkeiten.
- Minimale Ausführungseinheit wenn Anzahl Prozessoren =  $\infty$ ?



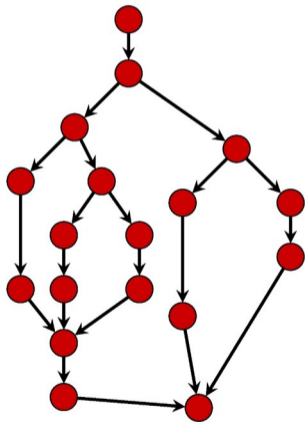
# Performanzmodell

- $p$  Prozessoren
- Dynamische Zuteilung
- $T_p$ : Ausführungszeit auf  $p$  Prozessoren



# Performanzmodell

- $T_p$ : Ausführungszeit auf  $p$  Prozessoren
- $T_1$ : *Arbeit*: Zeit für die gesamte Berechnung auf einem Prozessor
- $T_1/T_p$ : Speedup

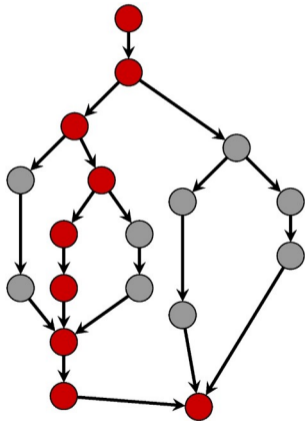


# Performanzmodell

- $T_\infty$ : **Zeitspanne**: Kritischer Pfad. Ausführungszeit auf  $\infty$  Prozessoren. Längster Pfad von der Wurzel zur Senke.
- $T_1/T_\infty$ : **Parallelität**: breiter ist besser
- Untere Grenzen

$$T_p \geq T_1/p \quad \text{Arbeitsgesetz}$$

$$T_p \geq T_\infty \quad \text{Zeitspannengesetz}$$



# Greedy Scheduler

Greedy Scheduler: teilt zu jeder Zeit so viele Tasks zu Prozessoren zu wie möglich.

## Theorem

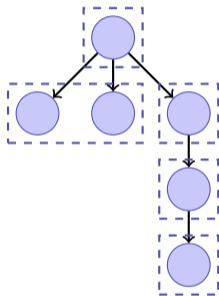
*Auf einem idealen Parallelrechner mit  $p$  Prozessoren führt ein Greedy-Scheduler eine mehrfädige Berechnung mit Arbeit  $T_1$  und Zeitspanne  $T_\infty$  in Zeit*

$$T_p \leq T_1/p + T_\infty$$

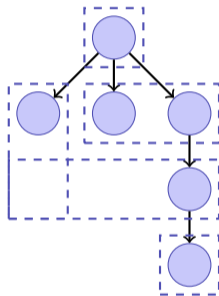
*aus.*

# Beispiel

Annahme  $p = 2$ .



$$T_p = 5$$



$$T_p = 4$$



# Beweis des Theorems

Annahme, dass alle Tasks gleich viel Arbeit aufweisen.

- Vollständiger Schritt:  $p$  Tasks stehen zur Berechnung bereit
- Unvollständiger Schritt: weniger als  $p$  Tasks bereit.

Annahme: Anzahl vollständige Schritte grösser als  $\lfloor T_1/p \rfloor$ . Ausgeführte Arbeit  $\geq \lfloor T_1/p \rfloor \cdot p + p = T_1 - T_1 \bmod p + p > T_1$ . Widerspruch. Also maximal  $\lfloor T_1/p \rfloor$  vollständige Schritte.

Betrachten nun den Graphen der ausstehenden Tasks. Jeder maximale (kritische) Pfad beginnt mit einem Knoten  $t$  mit  $\deg^-(t) = 0$ . Jeder unvollständige Schritt führt zu jedem Zeitpunkt alle vorhandenen Tasks  $t$  mit  $\deg^-(t) = 0$  aus und verringert also die Länge der Zeitspanne. Anzahl unvollständige Schritte also begrenzt durch  $T_\infty$ .

# Konsequenz

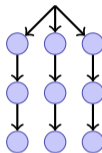
Wenn  $p \ll T_1/T_\infty$ , also  $T_\infty \ll T_1/p$ , dann  $T_p \approx T_1/p$ .

## Beispiel Fibonacci

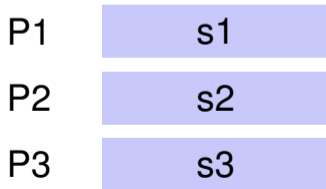
$T_1(n)/T_\infty(n) = \Theta(\phi^n/n)$ . Für moderate Grössen von  $n$  können schon viele Prozessoren mit linearem Speedup eingesetzt werden.

# Granularität: Wie viele Tasks?

- #Tasks = #Cores?
- Problem: wenn ein Core nicht voll ausgelastet werden kann
- Beispiel: 9 Einheiten Arbeit. 3 Cores. Scheduling von 3 sequentiellen Tasks.

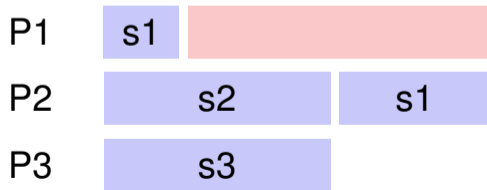


Exklusive Auslastung:



Ausführungszeit: 3 Einheiten

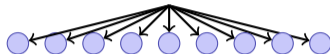
Fremder Thread "stört":



Ausführungszeit: 5 Einheiten

# Granularität: Wie viele Tasks?

- #Tasks = Maximum?
- Beispiel: 9 Einheiten Arbeit. 3 Cores.  
Scheduling von 9 sequentiellen Tasks.



Exklusive Auslastung:

P1	s1	s4	s7
P2	s2	s5	s8
P3	s3	s6	s9

Ausführungszeit:  $3 + \varepsilon$  Einheiten

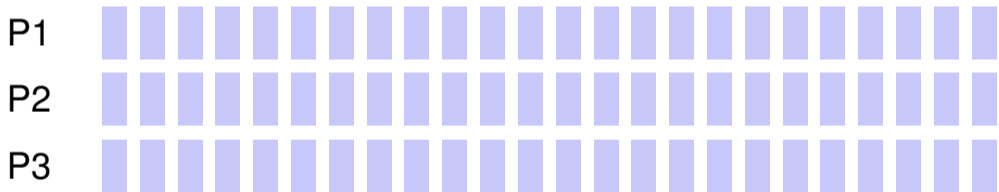
Fremder Thread "stört":

P1	s1	[Redacted]		
P2	s2	s4	s5	s8
P3	s3	s6	s7	s9

Ausführungszeit: 4 Einheiten.  
Volle Auslastung.

# Granularität: Wie viele Tasks?

- #Tasks = Maximum?
- Beispiel:  $10^6$  kleine Einheiten Arbeit.



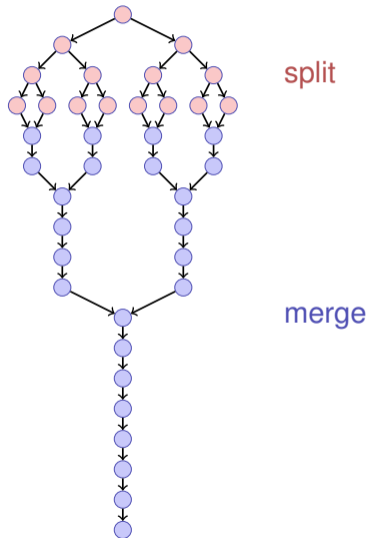
Ausführungszeit: dominiert vom Overhead.

# Granularität: Wie viele Tasks?

Antwort: so viele Tasks wie möglich mit sequentielltem Cut-off, welcher den Overhead vernachlässigen lässt.

# Beispiel: Parallelität von Mergesort

- Arbeit (sequentielle Laufzeit) von Mergesort  $T_1(n) = \Theta(n \log n)$ .
- Span  $T_\infty(n) = \Theta(n)$
- Parallelität  $T_1(n)/T_\infty(n) = \Theta(\log n)$   
(Maximal erreichbarer Speedup mit  $p = \infty$  Prozessoren)



# 28. Parallel Programming II

Gemeinsamer Speicher, Nebenläufigkeit, Exkurs: Lock Algorithmus (Peterson), Gegenseitiger Ausschluss Race Conditions [C++ Threads: Williams, Kap. 2.1-2.2], [C++ Race Conditions: Williams, Kap. 3.1] [C++ Mutexes: Williams, Kap. 3.2.1, 3.3.3]



## **28.1 Gemeinsamer Speicher, Nebenläufigkeit**

# Gemeinsam genutzte Ressourcen (Speicher)

- Bis hier: fork-join Algorithmen: Datenparallel oder Divide und Conquer
- Einfache Struktur (Datenunabhängigkeit der Threads) zum Vermeiden von Wettlaufsituationen (race conditions)
- Funktioniert nicht mehr, wenn Threads gemeinsamen Speicher nutzen müssen.

# Konsistenz des Zustands

Gemeinsamer Zustand: Hauptschwierigkeit beim nebenläufigen Programmieren.

Ansätze:

- Unveränderbarkeit, z.B. Konstanten
- Isolierte Veränderlichkeit, z.B. Thread-lokale Variablen, Stack.
- Gemeinsame veränderliche Daten, z.B. Referenzen auf gemeinsamen Speicher, globale Variablen

# Schütze den gemeinsamen Zustand

- Methode 1: Locks, Garantiere exklusiven Zugriff auf gemeinsame Daten.
- Methode 2: lock-freie Datenstrukturen, garantiert exklusiven Zugriff mit sehr viel feinerer Granularität.
- Methode 3: Transaktionsspeicher (hier nicht behandelt)

# Kanonisches Beispiel

```
class BankAccount {  
    int balance = 0;  
public:  
    int getBalance(){ return balance; }  
    void setBalance(int x) { balance = x; }  
    void withdraw(int amount) {  
        int b = getBalance();  
        setBalance(b - amount);  
    }  
    // deposit etc.  
};
```

(korrekt bei Einzelthreadausführung)

# Ungünstige Verschachtelung (Bad Interleaving)

Paralleler Aufruf von `withdraw(100)` auf demselben Konto

Thread 1

```
int b = getBalance();
```

Thread 2

```
int b = getBalance();
```

```
setBalance(b-amount);
```

```
setBalance(b-amount);
```



# Verlockende Fallen

FALSCH:

```
void withdraw(int amount) {  
    int b = getBalance();  
    if (b==getBalance())  
        setBalance(b - amount);  
}
```

Bad interleavings lassen sich fast **nie mit wiederholtem Lesen** lösen

# Verlockende Fallen

Auch FALSCH:

```
void withdraw(int amount) {  
    setBalance(getBalance() - amount);  
}
```

**Annahmen über Atomizität von Operationen sind fast immer falsch**



# Gegenseitiger Ausschluss (Mutual Exclusion)

Wir benötigen ein Konzept für den gegenseitigen Ausschluss  
*Nur ein Thread* darf zu einer Zeit die Operation withdraw *auf demselben Konto* ausführen.

Der Programmierer muss den gegenseitigen Ausschluss sicherstellen.

# Mehr verlockende Fallen

```
class BankAccount {
    int balance = 0;
    bool busy = false;
public:
    void withdraw(int amount) {
        while (busy); // spin wait
        busy = true;
        int b = getBalance();
        setBalance(b - amount);
        busy = false;
    }

    // deposit would spin on the same boolean
};
```

funktioniert nicht!

# Das Problem nur verschoben!

Thread 1

```
while (busy); //spin  
  
busy = true;  
  
int b = getBalance();  
  
setBalance(b - amount);
```

Thread 2

```
while (busy); //spin  
  
busy = true;  
  
int b = getBalance();  
setBalance(b - amount);
```

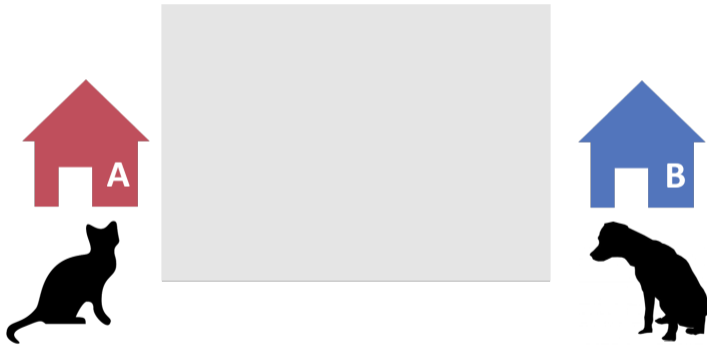
*t*

# Wie macht man das richtig?

- Wir benutzen ein *Lock* (eine Mutex) aus Bibliotheken
- Eine Mutex verwendet ihrerseits Hardwareprimitiven, *Read-Modify-Write* (RMW) Operationen, welche atomar lesen und abhängig vom Leseergebnis schreiben können.
- Ohne RMW Operationen ist der Algorithmus nichttrivial und benötigt zumindest atomaren Zugriff auf Variablen von primitivem Typ.

## **28.2 Exkurs: Lock Algorithmus**

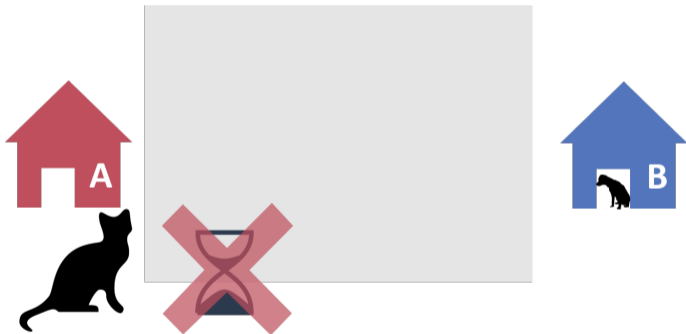
# Alice Katze und Bobs Dog



# Gefordert: Gegenseitiger Ausschluss



# Gefordert: Kein grundloses Aussperren





# Arten der Kommunikation

- Transient: Parteien kommunizieren zur selben Zeit

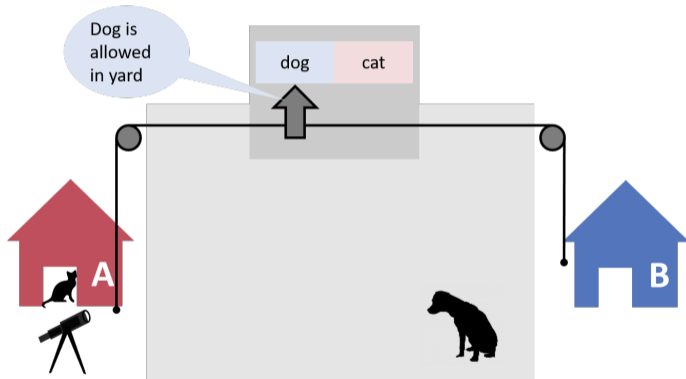


- Persistent: Parteien kommunizieren zu verschiedenen Zeiten

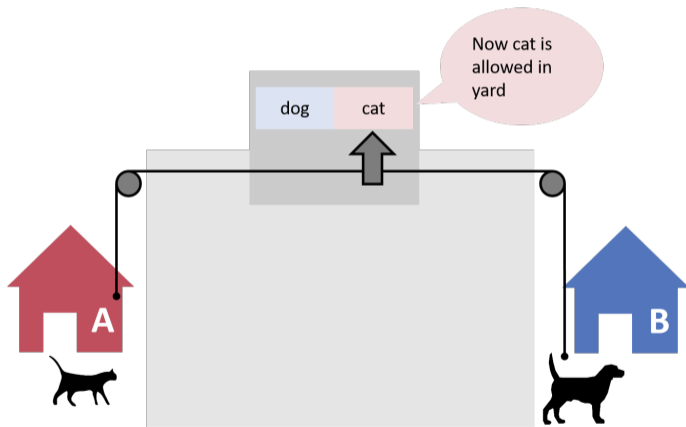


Gegenseitiger Ausschluss: Persistente Kommunikation

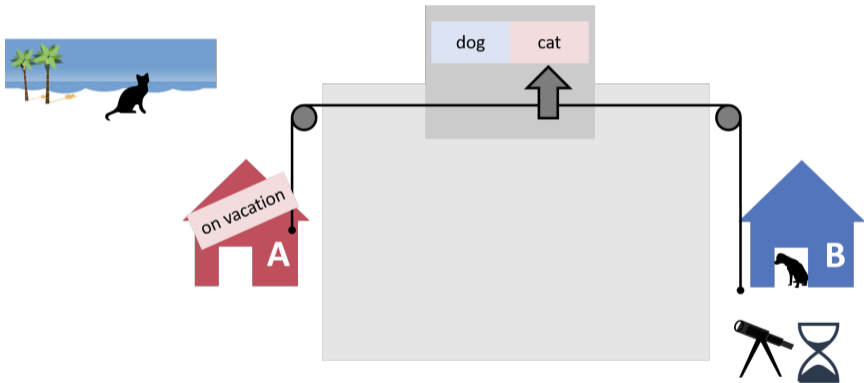
# Erste Idee



# Zugriffsprotokoll



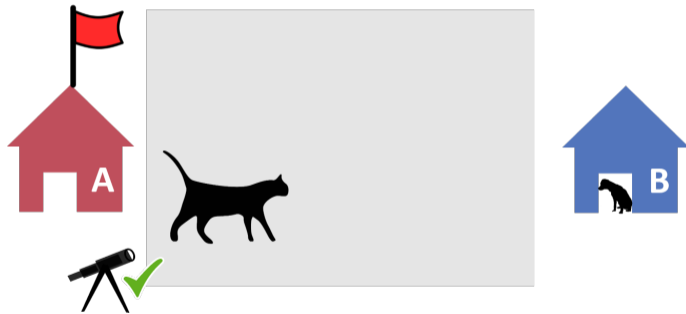
# Problem!



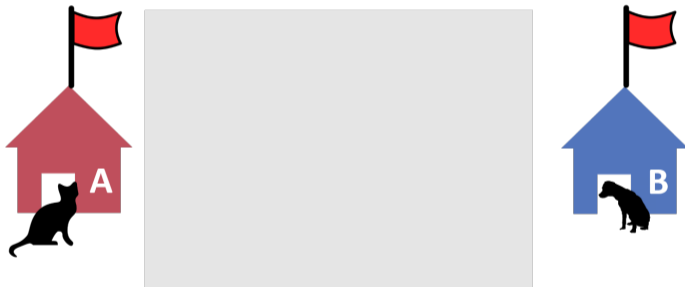
# Zweite Idee



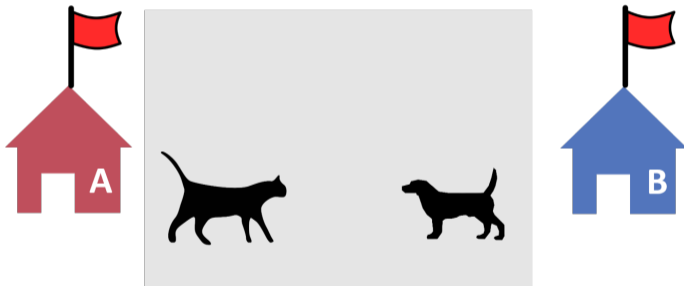
# Zugriffsprotokoll 2.1



# Anderes Szenario

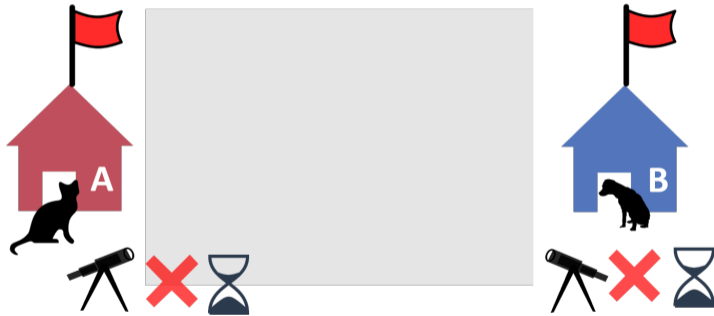


# Problem: Kein gegenseitiger Ausschluss

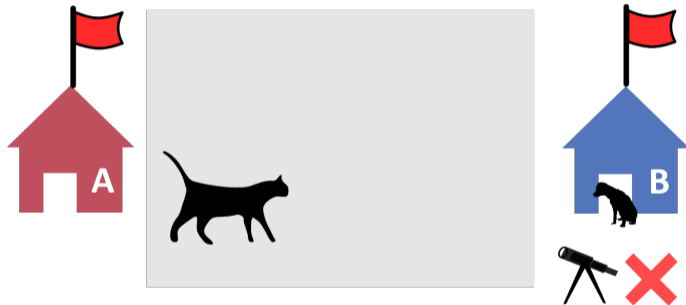




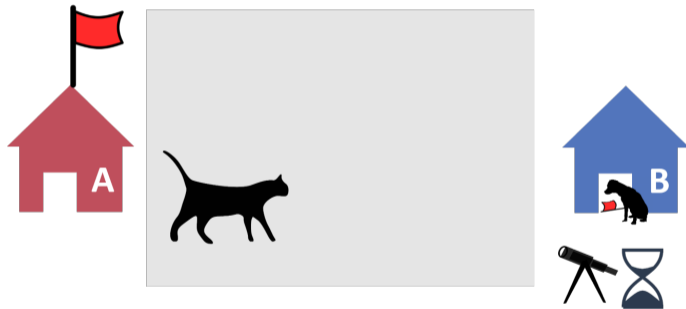
# Die Fahnen zweimal prüfen: Deadlock



# Zugriffsprotokoll 2.2



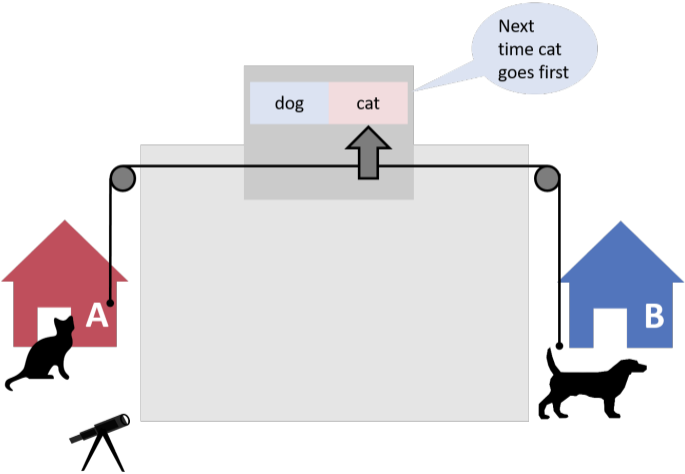
# Zugriffsprotokoll 2.2: beweisbar korrekt



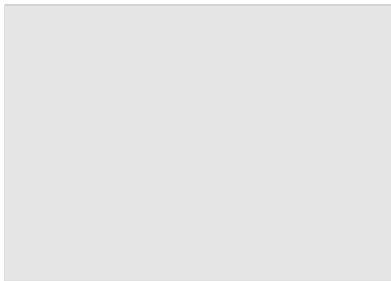
# Minor Problem: Starvation



# Lösung



# Das generelles Problem mit Locking bleibt



# Der Algorithmus von Peterson<sup>56</sup>

für zwei Prozesse ist beweisbar korrekt und frei von Starvation.

`non-critical section`

```
flag[me] = true // I am interested
victim = me // but you go first
// spin while we are both interested and you go first:
while (flag[you] && victim == me) {};
```

`critical section`

```
flag[me] = false
```

Der Code setzt voraus, dass der Zugriff auf `flag` / `victim` atomar, linearisiert oder sequentiell konsistent ist, eine Anforderung, welche – wie wir weiter unten sehen – für normale Variablen nicht unbedingt gegeben ist. Das Peterson-Lock wird auf moderner Hardware nicht eingesetzt.

---

<sup>56</sup>nicht prüfungsrelevant

## **28.3 Gegenseitiger Ausschluss**



# Kritische Abschnitte und Gegenseitiger Ausschluss

*Kritischer Abschnitt* (Critical Section)

Codestück, welches nur durch einen einzigen Thread zu einer Zeit ausgeführt werden darf.

*Gegenseitiger Ausschluss* (Mutual Exclusion)

Algorithmus zur Implementation eines kritischen Abschnitts

```
acquire_mutex(); // entry algorithm\  
... // critical section\  
release_mutex(); // exit algorithm
```

# Anforderung an eine Mutex.

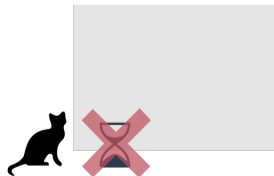
Korrektheit (Safety)

- Maximal ein Prozess in der kritischen Region



Fortschritt (Liveness)

- Das Betreten der kritischen Region darf nur endliche Zeit dauern, wenn kein Thread in der kritischen Region verweilt.



# Fast Korrekt

```
class BankAccount {
    int balance = 0;
    std::mutex m; // requires #include <mutex>
public:
    ...
    void withdraw(int amount) {
        m.lock();
        int b = getBalance();
        setBalance(b - amount);
        m.unlock();
    }
};
```

Was, wenn eine Exception auftritt?

# RAII Ansatz

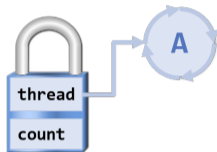
```
class BankAccount {
    int balance = 0;
    std::mutex m;
public:
    ...
    void withdraw(int amount) {
        std::lock_guard<std::mutex> guard(m);
        int b = getBalance();
        setBalance(b - amount);
    } // Destruction of guard leads to unlocking m
};
```

Was ist mit getBalance / setBalance?

# Reentrante Locks

## Reentrantes Lock (rekursives Lock)

- merkt sich den betroffenen Thread;
- hat einen Zähler
  - Aufruf von lock: Zähler wird inkrementiert
  - Aufruf von unlock: Zähler wird dekrementiert. Wenn Zähler = 0, wird das Lock freigegeben



# Konto mit reentrantem Lock

```
class BankAccount {
    int balance = 0;
    std::recursive_mutex m;
    using guard = std::lock_guard<std::recursive_mutex>;
public:
    int getBalance(){ guard g(m); return balance;
    }
    void setBalance(int x) { guard g(m); balance = x;
    }
    void withdraw(int amount) { guard g(m);
        int b = getBalance();
        setBalance(b - amount);
    }
};
```

## 28.4 Race Conditions

# Wettlaufsituation (Race Condition)

- Eine *Wettlaufsituation* (Race Condition) tritt auf, wenn das Resultat einer Berechnung vom Scheduling abhängt.
- Wir unterscheiden *bad interleavings* und *data races*
- Bad Interleavings können auch unter Verwendung einer Mutex noch auftreten.



# Beispiel: Stack

Stack mit korrekt synchronisiertem Zugriff:

```
template <typename T>
class stack{
    ...
    std::recursive_mutex m;
    using guard = std::lock_guard<std::recursive_mutex>;
public:
    bool isEmpty(){ guard g(m); ... }
    void push(T value){ guard g(m); ... }
    T pop(){ guard g(m); ...}
};
```

# Peek

Peek Funktion vergessen. Dann so?

```
template <typename T>
T peek (stack<T> &s){
    T value = s.pop();
    s.push(value);
    return value;
}
```

nicht Thread-sicher!

Code trotz fragwürdigem Stil in sequentieller Welt korrekt. Nicht so in nebenläufiger Programmierung!

# Bad Interleaving!

Initial leerer Stack  $s$ , nur von Threads 1 und 2 gemeinsam genutzt.

Thread 1 legt einen Wert auf den Stack und prüft, dass der Stack nichtleer ist. Thread 2 liest mit `peek()` den obersten Wert.

Thread 1

```
s.push(5);
```

```
assert(!s.isEmpty());
```

Thread 2

```
int value = s.pop();
```

```
s.push(value);
```

```
return value;
```

$t$



# Die Lösung

Peek muss mit demselben Lock geschützt werden, wie die anderen Zugriffsmethoden.

# Bad Interleavings

Race Conditions in Form eines Bad Interleavings können also auch auf hoher Abstraktionsstufe noch auftreten.

Betrachten nachfolgend andere Form der Wettlaufsituation: Data Race.

# Wie ist es damit?

```
class counter{
    int count = 0;
    std::recursive_mutex m;
    using guard = std::lock_guard<std::recursive_mutex>;
public:
    int increase(){
        guard g(m); return ++count;
    }
    int get(){
        return count;
    }
}
```

nicht Thread-sicher!

# Warum falsch?

Es sieht so aus, als könne hier nichts schiefgehen, da der Update von count in einem “winzigen Schritt” geschieht.

Der Code ist trotzdem falsch und von Implementationsdetails der Programmiersprache und unterliegenden Hardware abhängig.

Das vorliegende Problem nennt man *Data-Race*

Moral: *Vermeide Data-Races, selbst wenn jede denkbare Form von Verschachtelung richtig aussieht. Mache keine Annahmen über die Anordnung von Speicheroperationen.*

# Etwas formaler

*Data Race* (low-level Race-Conditions) Fehlerhaftes Programmverhalten verursacht durch ungenügend synchronisierten Zugriff zu einer gemeinsam genutzten Resource, z.B. gleichzeitiges Lesen/Schreiben oder Schreiben/Schreiben zum gleichen Speicherbereich.

*Bad Interleaving* (High Level Race Condition) Fehlerhaftes Programmverhalten verursacht durch eine unglückliche Ausführungsreihenfolge eines Algorithmus mit mehreren Threads, selbst dann wenn die gemeinsam genutzten Ressourcen anderweitig gut synchronisiert sind.



# Genau hingeschaut

```
class C {  
    int x = 0;  
    int y = 0;  
public:  
    void f() {  
        (A) x = 1;  
        (B) y = 1;  
    }  
    void g() {  
        (C) int a = y;  
        (D) int b = x;  
        assert(b >= a);  
    }  
}
```

Kann das  
schiefgehen

Es gibt keine Verschachtelung zweier f und g aufrufender Threads die die Bedingung in der Assertion falsch macht:

- A B C D ✓
- A C B D ✓
- A C D B ✓
- C A B D ✓
- C C D B ✓
- C D A B ✓

**Es kann trotzdem schiefgehen!**

# Ein Grund: Memory Reordering

*Daumenregel:* Compiler und Hardware dürfen die Ausführung des Codes so ändern, dass die *Semantik einer sequentiellen Ausführung* nicht geändert wird.

```
void f() {  
    x = 1;  
    y = x+1;  
    z = x+1;  
}
```

$\longleftrightarrow$   
sequentiell äquivalent

```
void f() {  
    x = 1;  
    z = x+1;  
    y = x+1;  
}
```

# Die Software-Perspektive

Moderne Compiler geben keine Garantie, dass die globale Anordnung aller Speicherzugriffe der Ordnung im Quellcode entsprechen

- Manche Speicherzugriffe werden sogar komplett wegoptimiert
- Grosses Potential für Optimierungen – und Fehler in der nebenläufigen Welt, wenn man falsche Annahmen macht

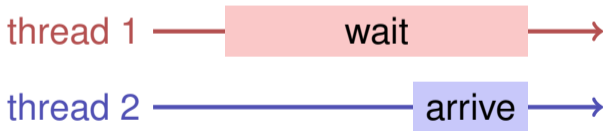
# Beispiel: Selbstgemachtes Rendezvous

```
int x; // shared
```

```
void wait(){  
    x = 1;  
    while(x == 1);  
}
```

```
void arrive(){  
    x = 2;  
}
```

Angenommen Thread 1 ruft wait auf, später ruft Thread 2 arrive auf. Was passiert?



# Kompilation

## Source

```
int x; // shared
```

```
void wait(){  
    x = 1;  
    while(x == 1);  
}
```

```
void arrive(){  
    x = 2;  
}
```

## Ohne Optimierung

```
wait:  
movl $0x1, x  
test: ←  
mov x, %eax  
cmp $0x1, %eax  
je test ← if equal
```

```
arrive:  
movl $0x2, x
```

## Mit Optimierung

```
wait:  
movl $0x1, x  
test: ← always  
jmp test
```

```
arrive  
movl $0x2, x
```

# Hardware Perspektive

Moderne Prozessoren erzwingen nicht die globale Anordnung aller Instruktionen aus Gründen der Performanz:

- Die meisten Prozessoren haben einen Pipeline und können Teile von Instruktionen simultan oder in anderer Reihenfolge ausführen.
- Jeder Prozessor(kern) hat einen lokalen Cache, der Effekt des Speicherns im gemeinsamen Speicher kann bei anderen Prozessoren u.U. erst zu einem späteren Zeitpunkt sichtbar werden.

# Speicherhierarchie

Registers

schnell, kleine Latenz, hohe Kosten, geringe Kapazität

L1 Cache

L2 Cache

...

System Memory

langsam, hohe Latenz, geringe Kosten, hohe Kapazität

# Eine Analogie

Anna

$C \leftarrow A \cdot C$   
n times

$C =$ 

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

 $A =$ 

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

$V =$ 

0	1	0	2	0	0	0	0
---	---	---	---	---	---	---	---

 $A =$ 

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

 $C =$ 

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

$h = 5$

$B =$ 

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

 $Z =$ 

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

global data

Beat

$Z = V$   
 $Z = A^h \cdot Z$

$Z =$ 

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

$A =$ 

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

local data

Zoe

$Z =$ 

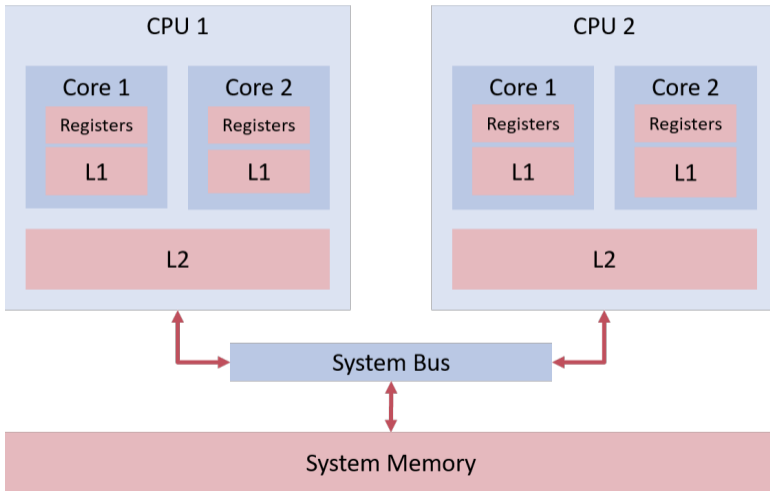
0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

wait until  $Z \neq 0$ .

then  $V = B \cdot Z$



# Schematisch



# Speichermodelle

Wann und ob Effekte von Speicheroperationen für Threads sichtbar werden, hängt also von Hardware, Laufzeitsystem und der Programmiersprache ab.

Ein *Speichermodell* (z.B. das von C++) gibt Minimalgarantien für den Effekt von Speicheroperationen.

- Lässt Möglichkeiten zur Optimierung offen
- Enthält Anleitungen zum Schreiben Thread-sicherer Programme

C++ gibt zum Beispiel *Garantien, wenn Synchronisation mit einer Mutex verwendet* wird.

# Repariert

```
class C {
    int x = 0;
    int y = 0;
    std::mutex m;
public:
    void f() {
        m.lock(); x = 1; m.unlock();
        m.lock(); y = 1; m.unlock();
    }
    void g() {
        m.lock(); int a = y; m.unlock();
        m.lock(); int b = x; m.unlock();
        assert(b >= a); // cannot fail
    }
};
```

# Atomic

Hier auch möglich:

```
class C {
    std::atomic_int x{0}; // requires #include <atomic>
    std::atomic_int y{0};
public:
    void f() {
        x = 1;
        y = 1;
    }
    void g() {
        int a = y;
        int b = x;
        assert(b >= a); // cannot fail
    }
};
```

# 29. Parallel Programming III

Verklemmung (Deadlock) und Verhungern (Starvation)

Producer-Consumer, Konzept des Monitors, Condition Variables

[Deadlocks : Williams, Kap. 3.2.4-3.2.5] [Condition Variables:

Williams, Kap. 4.1]

# Verklemmung (Deadlock) Motivation

```
class BankAccount {
    int balance = 0;
    std::recursive_mutex m;
    using guard = std::lock_guard<std::recursive_mutex>;
public:
    ...
    void withdraw(int amount) { guard g(m); ... }
    void deposit(int amount){ guard g(m); ... }

    void transfer(int amount, BankAccount& to){
        guard g(m);
        withdraw(amount);
        to.deposit(amount);
    }
};
```

Problem?

# Verklemmung (Deadlock) Motivation

Betrachte BankAccount Instanzen  $x$  und  $y$

Thread 1: `x.transfer(1,y);`

Thread 2: `y.transfer(1,x);`

acquire lock for  $x$  ← 

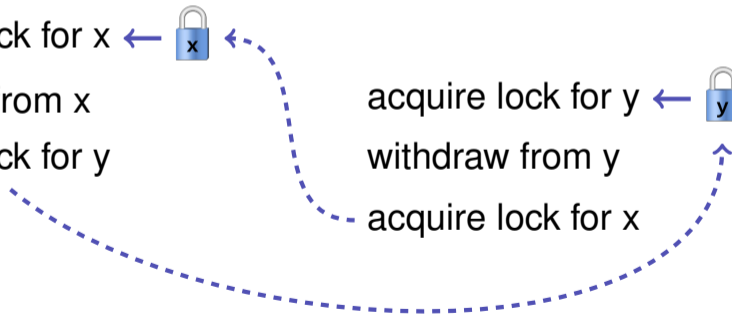
withdraw from  $x$

acquire lock for  $y$

acquire lock for  $y$  ← 

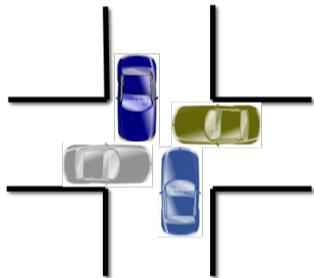
withdraw from  $y$

acquire lock for  $x$





# Deadlock

*Deadlock:* zwei oder mehr Prozesse sind gegenseitig blockiert, weil jeder Prozess auf einen anderen Prozess warten muss, um fortzufahren.



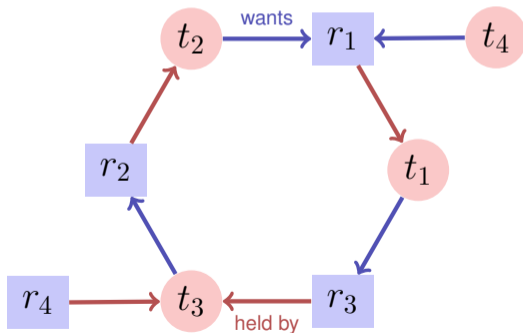


# Threads und Ressourcen

- Grafisch: Threads  $t$  und Ressourcen (Locks)  $r$
- Thread  $t$  versucht Resource  $a$  zu bekommen:   $t \rightarrow a$
- Resource  $b$  wird von Thread  $q$  gehalten:   $s \leftarrow b$

# Deadlock – Erkennung

Ein Deadlock für Threads  $t_1, \dots, t_n$  tritt auf, wenn der gerichtete Graph, der die Beziehung der  $n$  threads und Ressourcen  $r_1, \dots, r_m$  beschreibt, einen Kreis enthält.



# Techniken

- *Deadlock Erkennung* findet die Zyklen im Abhängigkeitsgraph. Deadlock kann normalerweise nicht geheilt werden: Freigeben von Locks resultiert in inkonsistentem Zustand.
- *Deadlock Vermeidung* impliziert, dass Zyklen nicht auftreten können
  - Grobere Granularität “one lock for all”
  - Zwei-Phasen-Locking mit Retry-Mechanismus
  - Lock-Hierarchien
  - ...
  - *Anordnen der Ressourcen*

# Zurück zum Beispiel

```
class BankAccount {
    int id; // account number, also used for locking order
    std::recursive_mutex m; ...
public:
    ...
    void transfer(int amount, BankAccount& to){
        if (id < to.id){
            guard g(m); guard h(to.m);
            withdraw(amount); to.deposit(amount);
        } else {
            guard g(to.m); guard h(m);
            withdraw(amount); to.deposit(amount);
        }
    }
};
```

# C++11 Stil

```
class BankAccount {
    ...
    std::recursive_mutex m;
    using guard = std::lock_guard<std::recursive_mutex>;
public:
    ...
    void transfer(int amount, BankAccount& to){
        std::lock(m,to.m); // lock order done by C++
        // tell the guards that the lock is already taken:
        guard g(m,std::adopt_lock); guard h(to.m,std::adopt_lock);
        withdraw(amount);
        to.deposit(amount);
    }
};
```

# Übrigens...

```
class BankAccount {
    int balance = 0;
    std::recursive_mutex m;
    using guard = std::lock_guard<std::recursive_mutex>;
public:
    ...
    void withdraw(int amount) { guard g(m); ... }
    void deposit(int amount){ guard g(m); ... }

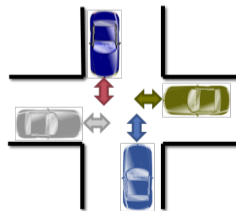
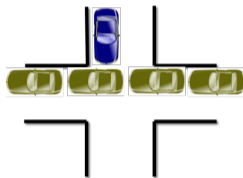
    void transfer(int amount, BankAccount& to){
        withdraw(amount);
        to.deposit(amount);
    }
};
```

Das hätte auch funktioniert. Allerdings verschwindet dann kurz das Geld, was inakzeptabel ist (kurzzeitige Inkonsistenz!)

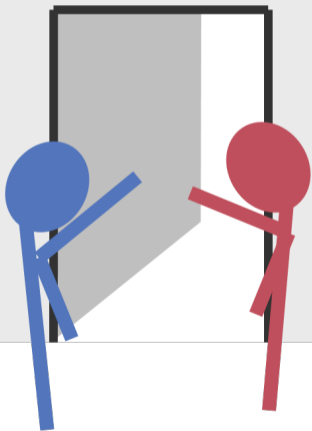
# Starvation und Livelock

*Starvation:* der wiederholte, erfolglose Versuch eine zwischenzeitlich freigegebene Resource zu erhalten, um die Ausführung fortzusetzen.

*Livelock:* konkurrierende Prozesse erkennen einen potentiellen Deadlock, machen aber keinen Fortschritt beim Auflösen des Problems.



# Politelock





# Produzenten-Konsumenten Problem

Zwei (oder mehr) Prozesse, Produzenten und Konsumenten von Daten, sollen mit Hilfe einer Datenstruktur entkoppelt werden.

Fundamentale Datenstruktur für den Bau von Software-Pipelines!



# Sequentielle Implementation (unbeschränkter Buffer)

```
class BufferS {
    std::queue<int> buf;
public:
    void put(int x){
        buf.push(x);
    }

    int get(){
        while (buf.empty()){ } // wait until data arrive
        int x = buf.front();
        buf.pop();
        return x;
    }
};
```

nicht Thread-sicher

# Wie wärs damit?

```
class Buffer {
    std::recursive_mutex m;
    using guard = std::lock_guard<std::recursive_mutex>;
    std::queue<int> buf;
public:
    void put(int x){ guard g(m);
        buf.push(x);
    }
    int get(){ guard g(m);
        while (buf.empty()){
            int x = buf.front();
            buf.pop();
            return x;
        }
    };
};
```

Deadlock

# Ok, so?

```
void put(int x){
    guard g(m);
    buf.push(x);
}
int get(){
    m.lock();
    while (buf.empty()){
        m.unlock();
        m.lock();
    }
    int x = buf.front();
    buf.pop();
    m.unlock();
    return x;
}
```

Ok, das geht, verschwendet aber CPU Zeit!

# Besser?

```
void put(int x){
    guard g(m);
    buf.push(x);
}
int get(){
    m.lock();
    while (buf.empty()){
        m.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
        m.lock();
    }
    int x = buf.front(); buf.pop();
    m.unlock();
    return x;
}
```

Ok, etwas besser. Limitiert aber die Reaktivität!

# Moral

Wir wollen das Warten auf eine Bedingung nicht selbst implementieren müssen.

Dafür gibt es bereits einen Mechanismus: *Bedingungsvariablen* (*condition variables*).

Das zugrunde liegende Konzept nennt man *Monitor*.

# Monitor

*Monitor* Abstrakte Datenstruktur, die mit einer Menge Operationen ausgestattet ist, die im gegenseitigen Ausschluss arbeiten und synchronisiert werden können.

Erfunden von C.A.R. Hoare und Per Brinch Hansen (cf. Monitors – An Operating System Structuring Concept, C.A.R. Hoare 1974)

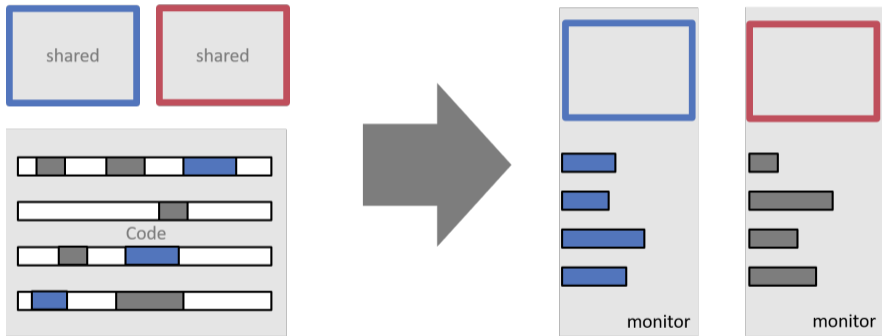


C.A.R. Hoare,  
\*1934



Per Brinch Hansen  
(1938-2007)

# Monitors vs. Locks





# Monitor und Bedingungen

Ein Monitor stellt, zusätzlich zum gegenseitigen Ausschluss, folgenden Mechanismus bereit:

*Warten auf Bedingungen:* Ist eine Bedingung nicht erfüllt, dann

- Gib das Lock auf
- Warte auf die Erfüllung der Bedingung
- Prüfe die Erfüllung der Bedingung wenn ein Signal gesendet wird

*Signalisieren:* Thread, der die Bedingung wahr machen könnte:

- Sende Signal zu potentiell wartenden Threads

# Bedingungsvariablen

```
#include <mutex>
#include <condition_variable>
...

class Buffer {
    std::queue<int> buf;

    std::mutex m;
    // need unique_lock guard for conditions
    using guard = std::unique_lock<std::mutex>;
    std::condition_variable cond;
public:
    ...
};
```

# Bedingungsvariablen

```
class Buffer {  
    ...  
public:  
    void put(int x){  
        guard g(m);  
        buf.push(x);  
        cond.notify_one();  
    }  
    int get(){  
        guard g(m);  
        cond.wait(g, [&]{return !buf.empty();});  
        int x = buf.front(); buf.pop();  
        return x;  
    }  
};
```

# Technische Details

- Ein Thread, der mit `cond.wait` wartet, läuft höchstens sehr kurz auf einem Core. Danach belastet er das System nicht mehr und “schläft”.
- Der Notify (oder Signal-) Mechanismus weckt schlafende Threads auf, welche nachfolgend ihre Bedingung prüfen.
  - `cond.notify_one` signalisiert *einen* wartenden Threads.
  - `cond.notify_all` signalisiert *alle* wartende Threads. Benötigt, wenn wartende Threads potentiell auf *verschiedene* Bedingungen warten.

# Technische Details

- In vielen anderen Sprachen gibt es denselben Mechanismus. Das Prüfen von Bedingungen (in einem Loop!) muss der Programmierer dort oft noch selbst implementieren.

## Java Beispiel

```
synchronized long get() {  
    long x;  
    while (isEmpty())  
        try {  
            wait ();  
        } catch (InterruptedException e) { }  
    x = doGet();  
    return x;  
}
```

```
synchronized put(long x){  
    doPut(x);  
    notify ();  
}
```

# Übrigens: mit bounded Buffer..

```
class Buffer {  
    ...  
    CircularBuffer<int,128> buf; // from lecture 6  
public:  
    void put(int x){ guard g(m);  
        cond.wait(g, [&]{return !buf.full();});  
        buf.put(x);  
        cond.notify_all();  
    }  
    int get(){ guard g(m);  
        cond.wait(g, [&]{return !buf.empty();});  
        cond.notify_all();  
        return buf.get();  
    }  
};
```

# 30. Parallel Programming IV

Futures, Read-Modify-Write Instruktionen, Atomare Variablen, Idee der lockfreien Programmierung

[C++ Futures: Williams, Kap. 4.2.1-4.2.3] [C++ Atomic: Williams, Kap. 5.2.1-5.2.4, 5.2.7] [C++ Lockfree: Williams, Kap. 7.1.-7.2.1]

# Futures: Motivation

Threads waren bisher Funktionen ohne Resultat:

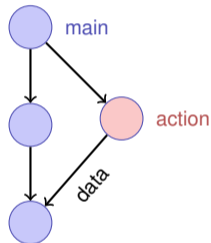
```
void action(some parameters){  
    ...  
}  
  
std::thread t(action, parameters);  
...  
t.join();  
// potentially read result written via ref-parameters
```



# Futures: Motivation

Wir wollen nun etwas in dieser Art:

```
T action(some parameters){  
  ...  
  return value;  
}  
  
std::thread t(action, parameters);  
...  
value = get_value_from_thread();
```



# Wir können das schon!

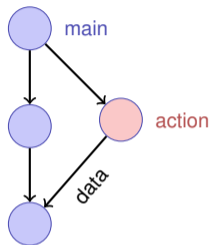
- Wir verwenden das Producer/Consumer Pattern (implementiert mit Bedingungsvariablen)
- Starten einen Thread mit Referenz auf den Buffer
- Wenn wir das Resultat brauchen, holen wir es vom Buffer
- Synchronisation ist ja bereits implementiert

# Zur Erinnerung

```
template <typename T>
class Buffer {
    std::queue<T> buf;
    std::mutex m;
    std::condition_variable cond;
public:
    void put(T x){ std::unique_lock<std::mutex> g(m);
        buf.push(x);
        cond.notify_one();
    }
    T get(){ std::unique_lock<std::mutex> g(m);
        cond.wait(g, [&]{return (!buf.empty());});
        T x = buf.front(); buf.pop(); return x;
    }
};
```

# Anwendung

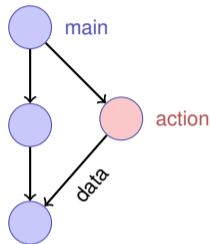
```
void action(Buffer<int>& c){  
    // some long lasting operation ...  
    c.put(42);  
}  
  
int main(){  
    Buffer<int> c;  
    std::thread t(action, std::ref(c));  
    t.detach(); // no join required for free running thread  
    // can do some more work here in parallel  
    int val = c.get();  
    // use result  
    return 0;  
}
```



# Mit C++11 Bordmitteln

```
int action(){
    // some long lasting operation
    return 42;
}

int main(){
    std::future<int> f = std::async(action);
    // can do some work here in parallel
    int val = f.get();
    // use result
    return 0;
}
```



## **30.2 Read-Modify-Write**

# Beispiel: Atomare Operationen in Hardware

## CMPXCHG

## Compare and Exchange

Compares the value in the AL, AX, EAX, or RAX register with the value in a register or a memory location (first operand). If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the rFLAGS register to 1. Otherwise, it copies the value in the first operand to the AL, AX, EAX, or RAX register and clears the ZF flag to 0.

The OF, SF, AF, DF, and CF flags are set to reflect the results of the compare.

When the first operand is a register, the instruction performs a read-modify-write on the register. When the first operand is a memory location, the instruction performs a read-modify-write on the memory location. The second operand must be the same size as the first operand.

The forms of the instruction are described in the following sections. For details about the LOCK prefix, see the LOCK prefix section.

### Mnemonic

CMPXCHG reg

CMPXCHG reg

CMPXCHG reg

CMPXCHG reg/mem64, reg64 0F B1 /r

### Related Instructions

CMPXCHG8B, CMPXCHG16B

CMPXCHG mem, reg  
«compares the value in Register A with the value in a memory location. If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the flag registers to 1. Otherwise it copies the value in the first operand to A register and clears ZF flag to 0»

Compares the value in the first register or memory location with the value in the second register or memory location. If equal, copy the second operand to the first operand. Otherwise, copy the first operand to RAX.

## 1.2.5 Lock Prefix

The LOCK prefix causes certain kinds of memory read-modify-write instructions to occur atomically. The mechanism for doing so is implementation-dependent (for example, the mechanism may involve

8

«The lock prefix causes certain kinds of memory read-modify-write instructions to occur atomically»

Instruction Formats



24594—Rev. 3.14—September 2007

AMD64 Technology

bus signaling or packet messaging between the processor and a memory controller). The prefix is intended to give the processor exclusive use of shared memory in a multiprocessor system.

The LOCK prefix can only be used with forms of the following instructions that write a memory operand: ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XADD, XCHG, and XOR. An invalid-opcode exception occurs if the LOCK prefix is used with any other instruction.

AMD64 Architecture  
Programmer's Manual

# Read-Modify-Write

Konzept von **Read-Modify-Write**: Der Effekt von Lesen, Verändern und Zurückschreiben, wird zu einem Zeitpunkt sichtbar (geschieht atomar).



# Pseudo-Code für CAS – Compare-And-Swap

```
bool CAS(int& variable, int& expected, int desired){  
    if (variable == expected){  
        variable = desired;  
        return true;  
    }  
    else{  
        expected = variable;  
        return false;  
    }  
}
```

atomic

# Verwendungsbeispiel CAS in C++11

Wir bauen unser eigenes (Spin-)Lock:

```
class Spinlock{
    std::atomic<bool> taken {false};
public:
    void lock(){
        bool old = false;
        while (!taken.compare_exchange_strong(old=false, true)){}
    }
    void unlock(){
        bool old = true;
        assert(taken.compare_exchange_strong(old, false));
    }
};
```

## 30.3 Lock-Freie Programmierung

Ideen

# Lock-freie Programmierung

Datenstruktur heisst

- *lock-frei*: zu jeder Zeit macht mindestens ein Thread in beschränkter Zeit Fortschritt, selbst dann, wenn viele Algorithmen nebenläufig ausgeführt werden. Impliziert systemweiten Fortschritt aber nicht Starvationfreiheit.
- *wait-free*: jeder Thread macht zu jeder Zeit in beschränkter Zeit Fortschritt, selbst dann wenn andere Algorithmen nebenläufig ausgeführt werden.

# Fortschrittsbedingungen

	Lock-frei	Blockierend
Jeder macht Fortschritt	Wait-frei	Starvation-frei
Mindestens einer macht Fortschritt	Lock-frei	Deadlock-frei

# Implikation

- Programmieren mit Locks: jeder Thread kann andere Threads beliebig blockieren.
- Lockfreie Programmierung: der Ausfall oder das Aufhängen eines Threads kann nicht bewirken, dass andere Threads blockiert werden

# Wie funktioniert lock-freie Programmierung?

Beobachtung:

- RMW-Operationen sind in Hardware *Wait-Free* implementiert.
- Jeder Thread sieht das Resultat eines CAS oder TAS in begrenzter Zeit.

Idee der lock-freien Programmierung: lese Zustand der Datenstruktur und verändere die Datenstruktur *atomar* dann und nur dann, wenn der gelesene Zustand unverändert bleibt.

# Beispiel: lock-freier Stack

Nachfolgend vereinfachte Variante eines Stacks

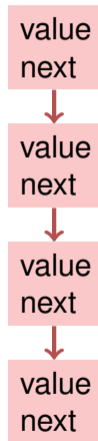
- pop prüft nicht, ob der Stack leer ist
- pop gibt nichts zurück



# (Node)

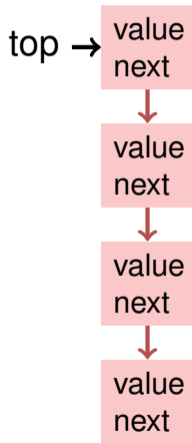
Nodes:

```
struct Node {  
    T value;  
  
    Node<T>* next;  
    Node(T v, Node<T>* nxt): value(v), next(nxt) {}  
};
```



# (Blockierende Version)

```
template <typename T>
class Stack {
    Node<T> *top=nullptr;
    std::mutex m;
public:
    void push(T val){ guard g(m);
        top = new Node<T>(val, top);
    }
    void pop(){ guard g(m);
        Node<T>* old_top = top;
        top = top->next;
        delete old_top;
    }
};
```



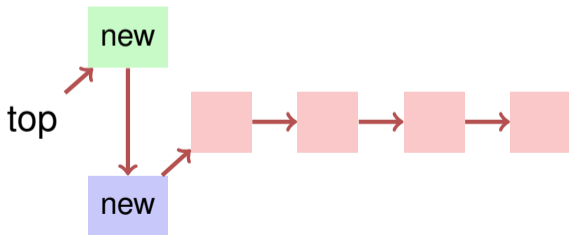
# Lock-Frei

```
template <typename T>
class Stack {
    std::atomic<Node<T>*> top {nullptr};
public:
    void push(T val){
        Node<T>* new_node = new Node<T> (val, top);
        while (!top.compare_exchange_weak(new_node->next, new_node));
    }
    void pop(){
        Node<T>* old_top = top;
        while (!top.compare_exchange_weak(old_top, old_top->next));
        delete old_top;
    }
};
```

# Push

```
void push(T val){  
    Node<T>* new_node = new Node<T> (val, top);  
    while (!top.compare_exchange_weak(new_node->next, new_node));  
}
```

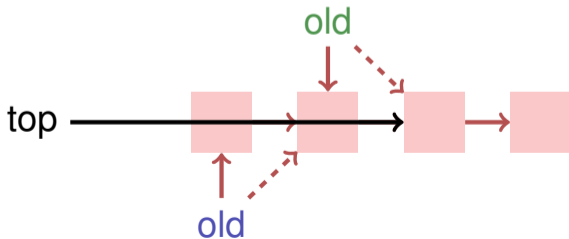
2 Threads:



# Pop

```
void pop(){  
    Node<T>* old_top = top;  
    while (!top.compare_exchange_weak(old_top, old_top->next));  
    delete old_top;  
}
```

2 Threads:



# Lockfreie Programmierung – Grenzen

- Lockfreie Programmierung ist kompliziert.
- Wenn mehr als ein Wert nebenläufig angepasst werden muss (Beispiel: Queue), wird es schwieriger. Damit Algorithmen lock-frei bleiben, müssen Threads sich “gegenseitig helfen”.
- Bei Speicherwiederverwendung kann das *ABA Problem* auftreten. Die Lösung dieses Problems ist aufwändig.