

# Algorithmen und Komplexität

Stephan Schulz

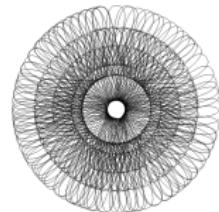
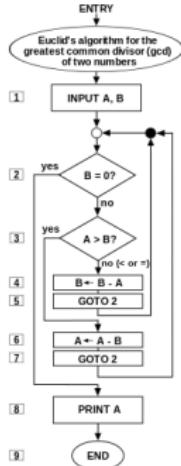
[stephan.schulz@dhw-stuttgart.de](mailto:stephan.schulz@dhw-stuttgart.de)

Monika Kochanowski

[monika.kochanowski@dhw-stuttgart.de](mailto:monika.kochanowski@dhw-stuttgart.de)

Janko Dietzsch

[janko.dietzsch@dhw-stuttgart.de](mailto:janko.dietzsch@dhw-stuttgart.de)



mit Beiträgen von Jan Hladík und Falko Kötter  
[jan.hladik@dhw-stuttgart.de](mailto:jan.hladik@dhw-stuttgart.de) [koetter@lehre.dhbw-stuttgart.de](mailto:koetter@lehre.dhbw-stuttgart.de)

# Inhaltsverzeichnis I

1

Einführung

2

Komplexität

●  $\mathcal{O}$ -Notation

● Einschub: Logarithmen

● Komplexität anschaulich

● Dynamisches Programmieren

● Rekurrenzen und Divide&Conquer

3

Arrays

4

Listen

5

Sortieren

● Einfache Sortierverfahren

● Sortieren mit Divide-and-Conquer

● Heaps and Heapsort

● Sortieren – Abschluss

6

Schlüsselmengen/Wert-Verwaltung

- Binäre Suchbäume
- Balancierte Bäume und AVL-Bäume
- Hashing

7

Graph-Algorithmen

- Minimale Spannbäume & Algorithmus von Prim
- Kürzeste Wege/Dijkstra

8

Zusammenfassung

9

Einzelvorlesungen

- Vorlesung 1
- Vorlesung 2
- Vorlesung 3
- Vorlesung 4
- Vorlesung 5
- Vorlesung 6

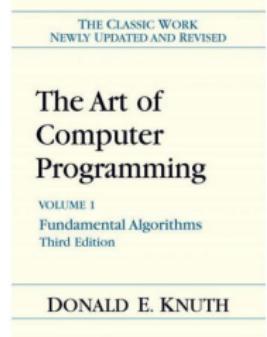
- Vorlesung 7
- Vorlesung 8
- Vorlesung 9
- Vorlesung 10
- Vorlesung 11
- Vorlesung 12
- Vorlesung 13
- Vorlesung 14
- Vorlesung 15
- Vorlesung 16
- Vorlesung 17
- Vorlesung 18
- Vorlesung 19
- Vorlesung 20
- Vorlesung 21
- Vorlesung 22
- Graphalgorithmen

# Semesterübersicht

- ▶ Was sind Algorithmen?
- ▶ Wie kann man die Komplexität von Algorithmen beschreiben?
  - ▶ Platzbedarf
  - ▶ Zeitbedarf
- ▶ Mathematische Werkzeuge zur Komplexitätsanalyse
  - ▶ Z.B. Rekurrenzrelationen
- ▶ Klassifikation von Algorithmen
  - ▶ Z.B. Brute Force, Greedy, Divide&Conquer, Dynamic Programming
  - ▶ Ansätze zur Algorithmenentwicklung
- ▶ Algorithmen und Datenstrukturen
  - ▶ Arrays
  - ▶ Listen
  - ▶ Suchbäume
  - ▶ Heaps
  - ▶ Hashes
  - ▶ Graphen

# Literatur

- ▶ Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms*
  - ▶ 3. Auflage 2009, moderner Klassiker (der Titel lügt)
- ▶ Robert Sedgewick, Kevin Wayne: *Algorithms*
  - ▶ 4. Auflage 2011, moderner Klassiker
  - ▶ Ähnlich auch als *Algorithms in C/Java/C++*
- ▶ Donald Knuth: *The Art of Computer Programming*
  - ▶ Die Bibel, seit 1962, Band 1 1968 (3. Auflage 1997), Band 4a 2011, Band 5 geplant für 2020 2025 (!)
- ▶ Niklaus Wirth: *Algorithmen und Datenstrukturen*
  - ▶ Deutschsprachiger Klassiker (1. Auflage 1975), 5. Auflage 2013



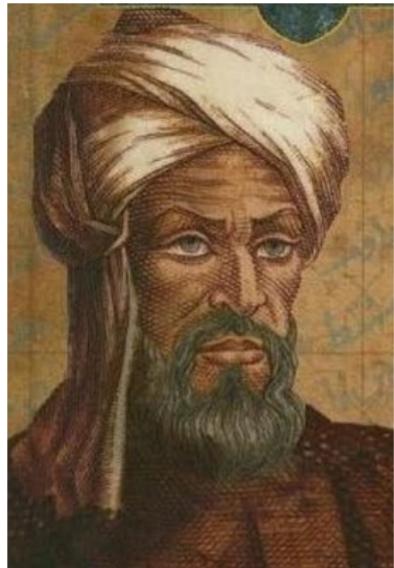
# Informelle Definition: Algorithmus

- ▶ Ein Algorithmus ist ein Verfahren zur Lösung eines Problems oder einer Problemklasse
- ▶ Ein Algorithmus...
  - ▶ ... überführt eine Eingabe in eine Ausgabe
  - ▶ ... besteht aus endlich vielen Einzelschritten
  - ▶ ... ist ohne tiefes Verständnis durchführbar
  - ▶ Jeder Einzelschritt ist wohldefiniert, ausführbar, und terminiert nach endlicher Zeit
  - ▶ Gelegentliche Forderung: Der Algorithmus terminiert (problematisch)
- ▶ Formalisierung: z.B. Turing-Maschine



Alan Turing  
(1912–1954)

# Der Begriff *Algorithmus*



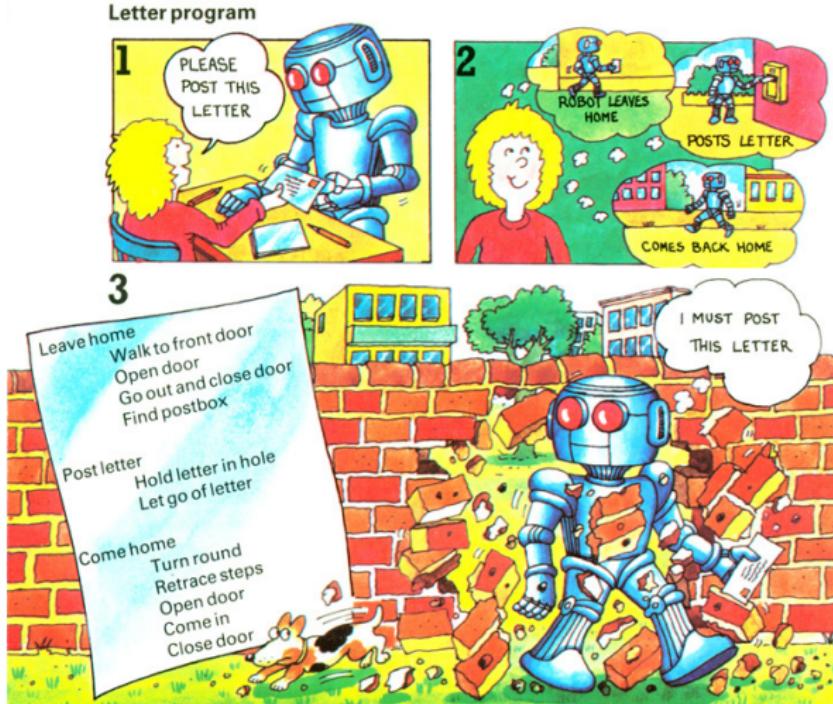
Muhammad ibn Musa al-Khwarizmi

- ▶ Mathematiker, Astronom, Geograph
- ▶ geboren ca. 780 nahe des Aralsees (heute Usbekistan)
- ▶ gestorben ca. 850 in Bagdad
- ▶ Latinisiert als *Algaurizin*, *Algorismi*, *Algoritmi*
- ▶ zahlreiche Werke
  - ▶ *Algoritmi de numero Indorum*
  - ▶ Rechenverfahren
  - ▶ Algebra

# Einige Klassen von Algorithmen

- ▶ Suchalgorithmen
  - ▶ Schlüssel in Datenbank
  - ▶ Pfad in Umgebung/Landkarte/Graph
  - ▶ Beweis in Ableitungsraum
- ▶ Sortierverfahren
  - ▶ Total
  - ▶ Topologisch
- ▶ Optimierungsverfahren
  - ▶ Spielpläne
  - ▶ Kostenminimierung
- ▶ Komprimierung
  - ▶ Lossy
  - ▶ Lossless
- ▶ Mathematische Algorithmen
  - ▶ Faktorisierung
  - ▶ Größter gemeinsamer Teiler
  - ▶ Gauß-Verfahren
- ▶ ...

# Warum Algorithmen wichtig (und schwierig) sind



*The computer is incredibly fast, accurate, and stupid. Man is unbelievably slow, inaccurate, and brilliant. The marriage of the two is a challenge and opportunity beyond imagination.*

Stuart G. Welsh

Introduction to Computer Programming -  
Basic for Beginners (1982)

# Ein gelöstes Problem?

- ▶ Moderne Computer haben so viele Ressourcen wie noch nie
  - ▶ Gigabytes an Hauptspeicher
  - ▶ Terabytes an Massenspeicher
  - ▶ Viele Milliarden Operationen pro Sekunde
- ▶ Viele aktuelle Programmiersprachen haben Datenstrukturen und Algorithmen eingebaut
  - ▶ Java: Array, Map, List, Set, Collections.Sort, ...
  - ▶ C++ STL: vector, list, queue, stack, sort, ...
- ▶ Aber was ist mit ... ?
  - ▶ Skalierbarkeit (z. B. Webseiten mit Millionen Nutzern)
  - ▶ Embedded Systems (z. B. im Auto)
  - ▶ Der Frage, welche Datenstrukturen man wann nutzen sollte
  - ▶ Neuen komplexen Anwendungsfälle (z. B. in der KI)
  - ▶ Individuellen Fragestellungen, die nicht im Standard sind?

# Beispiel: Euklids GGT-Algorithmus

- ▶ Problem: Finde den größten gemeinsamen Teiler (GGT) (*greatest common divisor, GCD*) für zwei natürliche Zahlen  $a$  und  $b$ .
- ▶ Also: Eingabe  $a, b \in \mathbb{N}$
- ▶ Ausgabe:  $c \in \mathbb{N}$  mit folgenden Eigenschaften:
  - ▶  $c$  teilt  $a$  ohne Rest
  - ▶  $c$  teilt  $b$  ohne Rest
  - ▶  $c$  ist die größte natürliche Zahl mit diesen Eigenschaften
- ▶ Beispiele:
  - ▶  $\text{ggt}(12, 9) = 3$
  - ▶  $\text{ggt}(30, 15) = 15$
  - ▶  $\text{ggt}(25, 11) = 1$

# Beispiel: Euklids GGT-Algorithmus



Euklid von  
Alexandria (ca. 3  
Jh. v. Chr.),  
*Elemente*

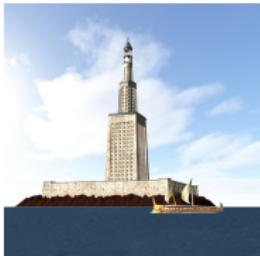
- Berechnung des GGT in Euklids Worten:  
Εἰ δὲ οὐ μετρεῖ ὁ ΓΔ τὸν ΑΒ, τῶν ΑΒ, ΓΔ  
ἀνθυφαιρουμένου ἀεὶ τοῦ ἐλάσσονος ἀπὸ τοῦ  
μείζονος λειφθήσεται τις ἀριθμός, ὃς μετρήσει τὸν  
πρὸ ἔαυτοῦ. μονὰς μὲν γὰρ οὐ λειφθήσεται: εἰ δὲ μῆ,  
*Wenn CD aber AB nicht misst, und man nimmt bei  
AB, CD abwechselnd immer das kleinere vom  
größeren weg, dann muss (schließlich) eine Zahl  
übrig bleiben, die die vorangehende misst.*

Elemente, Buch VII, Behauptung 2

# Beispiel: Euklids GGT-Algorithmus

Euklids Algorithmus moderner:

- ▶ Gegeben: Zwei natürliche Zahlen  $a$  und  $b$
- ▶ Wenn  $a = b$ : Ende, der GGT ist  $a$
- ▶ Ansonsten:
  - ▶ Sei  $c$  die absolute Differenz von  $a$  und  $b$ .
  - ▶ Bestimme den GGT von  $c$  und dem kleineren der beiden Werte  $a$  und  $b$



Der *Pharos*  
von Alexandria,  
Bild: Emad  
Victor  
Shenouda

# Übung: Euklids GGT-Algorithmus

- ▶ Algorithmus
  - ▶ Gegeben: Zwei natürliche Zahlen  $a$  und  $b$
  - ▶ Wenn  $a = b$ : Ende, der GGT ist  $a$
  - ▶ Ansonsten: Sei  $c$  die absolute Differenz von  $a$  und  $b$ .
  - ▶ Bestimme den GGT von  $c$  und dem kleineren der beiden Werte  $a$  und  $b$
- ▶ Aufgabe: Bestimmen Sie mit Euklids Algorithmus die folgenden GGTs. Notieren Sie die Zwischenergebnisse.
  - ▶  $\text{ggt}(16, 2)$
  - ▶  $\text{ggt}(36, 45)$
  - ▶  $\text{ggt}(17, 2)$
  - ▶  $\text{ggt}(121, 55)$
  - ▶  $\text{ggt}(2, 0)$

# Spezifikation von Algorithmen

Algorithmen können auf verschiedene Arten beschrieben werden:

- ▶ Informeller Text
- ▶ Semi-Formaler Text
- ▶ Pseudo-Code
- ▶ Konkretes Programm in einer Programmiersprache
- ▶ Flussdiagramm
- ▶ ...

# Euklid als (semi-formaler) Text

- ▶ Algorithmus: Größter gemeinsamer Teiler
  - ▶ Eingabe: Zwei natürliche Zahlen  $a, b$
  - ▶ Ausgabe: Größter gemeinsamer Teiler von  $a$  und  $b$
- 1 Wenn  $a$  gleich 0, dann ist das Ergebnis  $b$ . Ende.
  - 2 Wenn  $b$  gleich 0, dann ist das Ergebnis  $a$ . Ende.
  - 3 Wenn  $a$  größer als  $b$  ist, dann setze  $a$  gleich  $a - b$ .  
Mache mit Schritt 3 weiter.
  - 4 Wenn  $b$  größer als  $a$  ist, dann setze  $b$  gleich  $b - a$ .  
Mache mit Schritt 3 weiter.
  - 5 Ansonsten:  $a$  ist gleich  $b$ , und ist der gesuchte GGT. Ende.

# Euklid als (Pseudo-)Code

```
def euclid_gcd(a, b):
```

```
    """
```

*Compute the Greatest Common Divisor of two numbers, using  
Euclid's naive algorithm.*

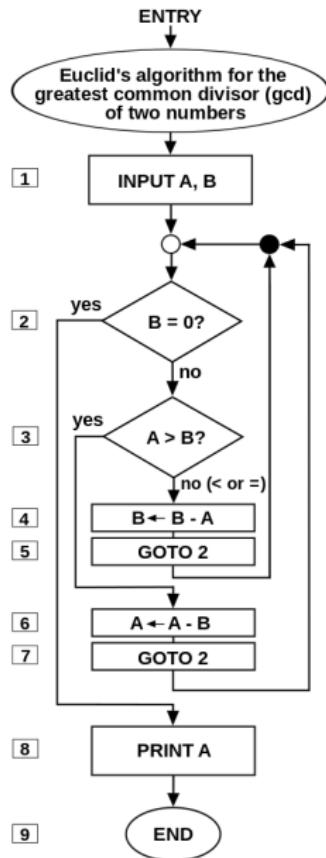
```
    """
```

```
    if a==0:  
        return b  
    if b==0:  
        return a  
    while a!=b:  
        if a>b:  
            a=a-b  
        else:  
            b=b-a  
    return a
```

# Euklid Rekursiv

```
def euclid_gcdr(a, b):
    """
    Compute the Greatest Common Divisor of two numbers, using
    Euclid's naive algorithm.
    """
    if a==0:
        return b
    if b==0:
        return a
    if a>b:
        return euclid_gcdr(a-b,b)
    else:
        return euclid_gcdr(b,b-a)
```

# Panta Rhei



- Flussdiagramm: Graphische Visualisierung des Algorithmus
- Wer findet den Fehler?

Was passiert bei  
 $A = 0, B \neq 0$ ?

# Übung: Euklids Worst Case

- Wie oft durchläuft der Euklidsche Algorithmus im schlimmsten Fall für ein gegebenes  $n = a + b$  die Schleife? Begründen Sie Ihr Ergebnis!

# Das Geheimnis des Modulus

- Der modulo-Operator ermittelt den **Divisionsrest** bei der ganzzahligen Division:

- Sei z.B.  $z = nq + r$  mit  $n, q, r \in \mathbb{N}$  und  $r < q$
- Dann ist

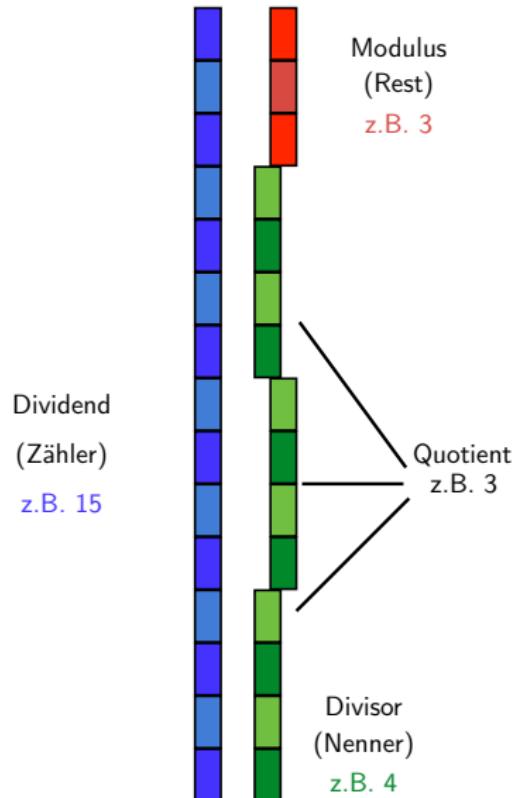
$$z/n = q \text{ mit Rest } r$$

- oder auch:

$$z/n = q \text{ und } z\%n = r$$

- Alternative Schreibweisen:

$$z \text{ div } n = q \text{ und } z \text{ mod } n = r$$



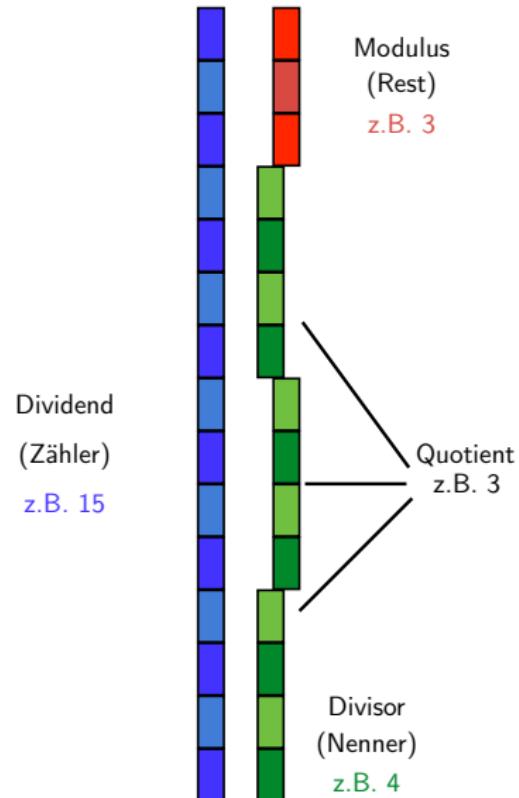
# Modulus Teil 2

## ► Eigenschaften:

- Der Divisionsrest **modulo  $n$**  liegt zwischen 0 und  $n - 1$
- Aufeinanderfolgende Zahlen **kleiner  $n$**  haben aufeinanderfolgende Divisionsreste
- Aufeinanderfolgende Zahlen haben **meistens** aufeinanderfolgende Divisionsreste (Ausnahme: Die größere ist glatt durch  $n$  teilbar)

## ► Verwendung:

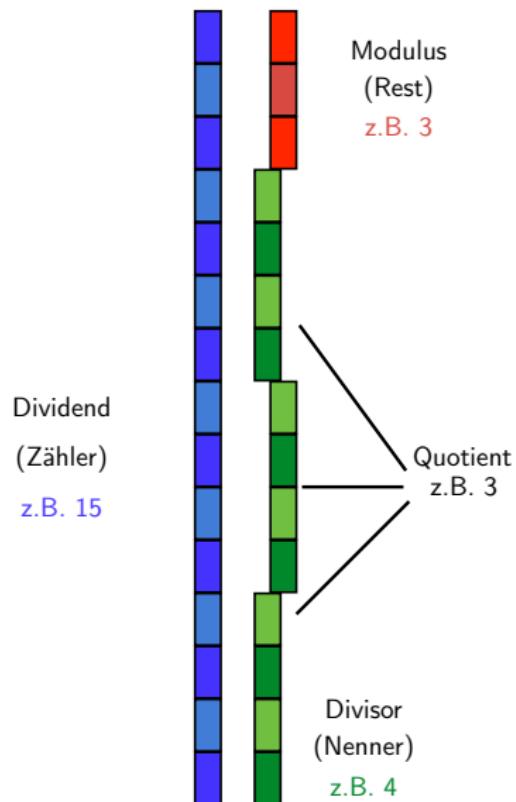
- Kryptographie (RSA)
- Hashing
- „Faire“ Verteilung auf  $n$  Töpfe



# Modulus Teil 3

## Beispiele

Divident	Divisor	Quotient	Modulus
0	3	0	0
1	3	0	1
2	3	0	2
3	3	1	0
4	3	1	1
5	3	1	2
6	3	2	0
25	17	1	8
26	17	1	9
27	17	1	10
34	17	2	0
35	17	2	1
37	1	37	0



# Übung: Modulus

$11 \bmod 15 =$	$19 \bmod 23 =$	$52 \bmod 2 =$
$82 \bmod 12 =$	$54 \bmod 29 =$	$66 \bmod 10 =$
$44 \bmod 26 =$	$12 \bmod 16 =$	$23 \bmod 15 =$
$96 \bmod 20 =$	$26 \bmod 15 =$	$87 \bmod 17 =$
$93 \bmod 26 =$	$64 \bmod 14 =$	$68 \bmod 20 =$
$99 \bmod 14 =$	$15 \bmod 25 =$	$36 \bmod 23 =$
$34 \bmod 19 =$	$28 \bmod 27 =$	$46 \bmod 14 =$
$71 \bmod 24 =$	$84 \bmod 24 =$	$62 \bmod 20 =$
$76 \bmod 27 =$	$21 \bmod 20 =$	$38 \bmod 17 =$
$96 \bmod 23 =$	$36 \bmod 14 =$	$44 \bmod 13 =$
$35 \bmod 25 =$	$72 \bmod 29 =$	$32 \bmod 7 =$

# Übung: Modulus

11 mod 15	=	11	19 mod 23	=	19	52 mod 2	=	0
82 mod 12	=	10	54 mod 29	=	25	66 mod 10	=	6
44 mod 26	=	18	12 mod 16	=	12	23 mod 15	=	8
96 mod 20	=	16	26 mod 15	=	11	87 mod 17	=	2
93 mod 26	=	15	64 mod 14	=	8	68 mod 20	=	8
99 mod 14	=	1	15 mod 25	=	15	36 mod 23	=	13
34 mod 19	=	15	28 mod 27	=	1	46 mod 14	=	4
71 mod 24	=	23	84 mod 24	=	12	62 mod 20	=	2
76 mod 27	=	22	21 mod 20	=	1	38 mod 17	=	4
96 mod 23	=	4	36 mod 14	=	8	44 mod 13	=	5
35 mod 25	=	10	72 mod 29	=	14	32 mod 7	=	4

Geschafft!

Ende Vorlesung 1

# GGT-Algorithmus von Euklid

- ▶ Algorithmus: Größter gemeinsamer Teiler
  - ▶ Eingabe: Zwei natürliche Zahlen  $a, b$
  - ▶ Ausgabe: Größter gemeinsamer Teiler von  $a$  und  $b$
- 1 Wenn  $a$  gleich 0, dann ist das Ergebnis  $b$ . Ende.
  - 2 Wenn  $b$  gleich 0, dann ist das Ergebnis  $a$ . Ende.
  - 3 Wenn  $a$  größer als  $b$  ist, dann setze  $a$  gleich  $a - b$ .  
Mache mit Schritt 3 weiter.
  - 4 Wenn  $b$  größer als  $a$  ist, dann setze  $b$  gleich  $b - a$ .  
Mache mit Schritt 3 weiter.
  - 5 Ansonsten:  $a$  ist gleich  $b$ , und ist der gesuchte GGT. Ende.

# Analyse: Euklids GGT-Algorithmus

Sei o.B.d.A  $a$  größer als  $b$  und sei  $g$  der  $\text{ggt}(a, b)$

- ▶ Dann gilt:  $a = m \cdot g$  und  $b = n \cdot g$  für  $m, n \in \mathbb{N}$  und  $m > n$
- ▶ Nach einem Schritt ist also  $a = (m - n)g$  und  $b = n \cdot g$ 
  - ▶  $g$  teilt immer noch  $a$  und  $b$  (Korrektheit!)
  - ▶ Wenn  $m$  groß gegen  $n$  ist, dann durchläuft der Algorithmus viele Schritte, bis  $a \leq b$  gilt

**Geht das auch schneller?**

Sei o.B.d.A  $a$  größer als  $b$  und sei  $g$  der  $\text{ggT}(a, b)$

- ▶ Dann gilt:  $a = m \cdot g$  und  $b = n \cdot g$  für  $m, n \in \mathbb{N}$  und  $m > n$
- ▶ Nach einem Schritt ist also  $a = (m - n)g$  und  $b = n \cdot g$ 
  - ▶  $g$  teilt immer noch  $a$  und  $b$  (Korrektheit!)
  - ▶ Wenn  $m$  groß gegen  $n$  ist, dann durchläuft der Algorithmus viele Schritte, bis  $a \leq b$  gilt
- ▶ Beobachtung: Es wird so lange immer wieder  $b$  von  $a$  abgezogen, bis  $a \leq b$  gilt!
  - ▶ Sei im folgenden  $a'$  der Originalwert von  $a$
  - ▶ Wenn wir  $b$   $i$ -mal von  $a'$  abziehen, so gilt also:  $a' = ib + a$
  - ▶ In anderen Worten:  $a$  ist der Divisionsrest von  $a'/b$ !

**Wir können also die wiederholten Subtraktionen durch eine Division mit Restberechnung ersetzen!**

# Euklid Schneller in (Pseudo)-Code

```
def euclid_gcd2(a, b):
    """ Compute the Greatest Common Divisor of two numbers,
       using an improved version od Euclid's algorithm.
    """
    while a!=b:
        if a==0:
            return b
        if b==0:
            return a
        if a>b:
            a=a%b
        else:
            b=b%a
    return a
```

# Übung: Euklid Schneller

- Aufgabe: Bestimmen Sie mit dem verbesserten Algorithmus die folgenden GGTs. Notieren Sie die Zwischenergebnisse.

- $\text{ggt}(16, 2)$
- $\text{ggt}(36, 45)$
- $\text{ggt}(17, 2)$
- $\text{ggt}(121, 55)$
- $\text{ggt}(89, 55)$

```
def euclid_gcd2(a, b):  
    while a!=b:  
        if a==0:  
            return b  
        if b==0:  
            return a  
        if a>b:  
            a=a%b  
        else:  
            b=b%a  
    return a
```

# Gruppenübung: Analyse von Euklid

- ▶ Wie viele Schritte/Schleifendurchläufe braucht der Euklidsche Algorithmus ungefähr?
- ▶ Fall 1: Naiver Euklid (mit Subtraktion  $a - b$ ):
  - ▶ Schlimmster Fall:  $b = 1$ , dann  $\approx a$  Schritte
- ▶ Fall 2: Verbesserter Algorithmus mit *Modulus*  $a \bmod b$ 
  - ▶ Ansatz: Betrachte z.B.  $\max(a, b)$  oder  $a + b$  vor und nach einem Schritt
  - ▶ Wir nehmen wieder o.B.d.A. an, dass  $a > b$ .
    - ▶ Ist  $a \geq 2b$ :  $a \bmod b < b \leq a/2$
    - ▶ Ist  $a < 2b$ :  $a \bmod b = a - b \leq a - a/2 \leq a/2$
    - ▶ Also:  $a$  schrumpft auf höchstens die Hälfte
    - ▶ Außerdem: Nach dem Schritt ist  $b$  größergleich dem neuen  $a$
    - ▶ Also: Im nächsten Schritt schrumpft  $b$  auf höchstens die Hälfte
    - ▶ Zusammen: Nach zwei Schritten ist  $a + b$  um mindestens den Faktor 2 geschrumpft!
  - ▶ Wie oft kann eine Ganzzahl sich halbieren, bis 1 herauskommt? Anzahl der Schleifendurchläufe ist beschränkt durch  $2 \log_2(a + b)$

# Übung: Datenstrukturen

- Zur Lösung eines gegebenen Problems kann es verschieden effiziente Algorithmen geben
  - Oft wesentlich: Geschickte Organisation der Daten durch geeignete **Datenstrukturen**
- Übung: Finden Sie die zu den 5 Namen in Spalte 1 gehörende Ziffernfolge in [Liste 1](#). Stoppen Sie Ihre Zeit! Übung: Finden Sie die zu den 5 Namen in Spalte 2 gehörende Ziffernfolge in [Liste 2](#). Stoppen Sie Ihre Zeit! Übung: Finden Sie die zu den 5 Namen in Spalte 3 gehörende Ziffernfolge in [Liste 3](#). Stoppen Sie Ihre Zeit!

<b>Spalte 1</b>	<b>Spalte 2</b>	<b>Spalte 3</b>
Stone, Jo	Mcdonald, Jeffrey	Sims, Helen
Pierce, Jaime	Palmer, Katie	Obrien, Kim
Nunez, Glenda	Pierce, Jaime	Curry, Courtney
Hawkins, Mona	Schmidt, Tami	Brewer, Marcella
Massey, Harold	French, Erica	Thornton, Dwight

# Datenstrukturen?

- ▶ Liste 1: Unsortiertes Array
  - ▶ Lineare Suche
- ▶ Liste 2: Sortiertes Array
  - ▶ Binäre Suche (oder auch auch „gefühlte [Interpolationssuche](#)“)
- ▶ Liste 3: Sortiertes Array mit pre-Hashing
  - ▶ Ditto, aber mit besserem Einstiegspunkt

Wahl der geeigneten Datenstruktur ermöglicht bessere/effizientere Suche!



*„Bad programmers worry about the code. Good programmers worry about data structures and their relationships.“ — Linus Torvalds, 2006*

## **Komplexität**

# Komplexität von Algorithmen

- ▶ Fragestellung: Wie **teuer** ist ein Algorithmus?
- ▶ Konkreter:
  - ▶ Wie viele (elementare) Schritte braucht er für eine gegebene Eingabe?
  - ▶ Wie viel Speicherplatz braucht er für eine gegebene Eingabe?
- ▶ Allgemeiner:
  - ▶ Wie viele elementare Schritte braucht ein Algorithmus für Eingaben einer bestimmten Länge?
    - ▶ ... im Durchschnitt?
    - ▶ ... schlimmstenfalls?
  - ▶ Wie viel Speicher braucht ein Algorithmus für Eingaben einer bestimmten Länge?
    - ▶ ... im Durchschnitt?
    - ▶ ... schlimmstenfalls?

# Effizienz und Auswahl von Algorithmen

- ▶ Was ist der **beste** Algorithmus für ein gegebenes Problem?
  - ▶ Verschiedene Algorithmen können sehr verschiedene Komplexität haben (Beispiel: Euklid!)
- ▶ Kriterien:
  - ▶ Performanz auf erwarteten Eingabedaten!
  - ▶ Performanz im Worst-Case
  - ▶ Anforderungen der Umgebung (Echtzeit? Begrenzter Speicher?)
  - ▶ (Nachweisbare) Korrektheit!

**Eleganz und Einfachheit sind schwer zu quantifizieren, aber ebenfalls wichtig!**

# Komplexitätsfragen

Die Zeit, die ein Computer für eine Operation braucht, hängt ab von

- ▶ Art der Operation (Addition ist einfacher als Logarithmus)
- ▶ Speicherort der Operanden (Register, Cache, Hauptspeicher, Swap)
- ▶ Länge der Operanden (8/16/32 Bit)
- ▶ Taktrate des Prozessors
- ▶ Programmiersprache / Compiler

Diese Parameter hängen ihrerseits ab von

- ▶ Prozessormodell
- ▶ Größe von Cache und Hauptspeicher
- ▶ Laufzeitbedingungen
  - ▶ Wie viele andere Prozesse?
  - ▶ Wie viel freier Speicher?

**Exakte Berechnung ist extrem aufwendig und  
umgebungsabhängig**

# Komplexität abstrakt

Um von den genannten Problemen zu abstrahieren, d.h. Komplexitätsberechnungen **praktikabel** und **umgebungsunabhängig** zu gestalten, definiert man:

- ▶ eine Zuweisung braucht **1 Zeiteinheit**
- ▶ eine arithmetische Operation braucht **1 ZE**
  - ▶ moderne Prozessoren arbeiten mit 64-Bit-Integers
  - ▶ Vereinfachung ist legitim für Zahlen von -9 bis +9 Trillionen
  - ▶ Ausnahmen für extrem große Zahlen ( $\sim$  Kryptographie)
- ▶ ein Vergleich (`if`, `while`, `for`) braucht **1 ZE**

Ende Vorlesung 2

# Komplexität und Eingabe (1)

Bei (fast) allen Algorithmen hängt die Laufzeit von der Größe der Eingabe ab

- ▶ Suchen/Sortieren: Anzahl der Elemente
- ▶ Matrix-Multiplikation: Dimensionen der Matrizen
- ▶ Graph-Operationen: Anzahl der Knoten/Kanten

~ Komplexität kann sinnvoll nur als **Funktion** angegeben werden,  
die von der **Größe der Eingabe** abhängt.

# Komplexität und Eingabe (1)

Größe wird meistens abstrakt beschrieben:

- ▶ eine Zahl benötigt **1 Größeneinheit**
- ▶ ein Buchstabe benötigt **1 GE**
- ▶ Elemente von komplexen Strukturen (Knoten/Kanten/...) benötigen **1 GE**

Für spezielle Algorithmen interessieren uns auch speziellere Größenmaße

- ▶ Z.B. nur Anzahl der Knoten in einem Graphen
- ▶ Z.B. lineare Größe einer quadratischen Matrix
- ▶ Z.B. numerischer Wert von Eingabedaten (Euklid!)

# Beispiel: Komplexität

## Beispiel: Matrix-Multiplikation

**Eingabe** zwei  $n \times n$ -Matrizen  $a, b$

**Ausgabe** Matrizenprodukt ( $n \times n$ -Matrix)

```
def matrix_mult(a,b):
    for x in range(n):
        for y in range(n):
            sum=0
            for z in range(n):
                sum=sum+a[x,z]*b[z,y]
            c[x,y]=sum
    return c
```

Anmerkung: `range(n)` läuft von  
0 ... ( $n - 1$ )

- ▶ Schleife z:  $n \cdot 4$
- ▶ Schleife y:  
 $n \cdot (3 + \textcolor{blue}{n} \cdot 4) = 3 \cdot n + 4 \cdot n^2$
- ▶ Schleife x:  
 $n \cdot (1 + \textcolor{blue}{3 \cdot n + 4 \cdot n^2}) =$   
 $n + 3 \cdot n^2 + 4 \cdot n^3$
- ▶ Funktion `matrix_mult()`:  
 $4n^3 + 3n^2 + n + 1$

# Übung: Komplexität

Ein Dozent verteilt  $n$  Klausuren an  $n$  Studenten. Er verwendet die folgenden Verfahren:

- 1 Er geht zum ersten Studenten, vergleicht dessen Namen mit denen auf jeder Klausur, und gibt dem Studenten seine Klausur, sobald er sie gefunden hat. Anschließend macht er beim nächsten Studenten weiter.
- 2 Der Dozent nimmt die erste Klausur, liest den Namen auf der Klausur und gibt die Klausur dem Studenten, der sich meldet.

Berechnen Sie, wie groß der Aufwand der Verteilung in Abhängigkeit von  $n$  bei jedem Verfahren ist. Machen Sie dabei die folgenden Annahmen:

- Der Vergleich von zwei Namen dauert eine ZE.
- In einem Stapel der Größe  $k$  ist die gesuchte Klausur an Position  $\lceil k/2 \rceil$  (und  $k$  wird zunehmend kleiner)
- Das Übergeben der Klausur an den entsprechenden Studenten (Variante 2) hat konstanten Aufwand von einer ZE.

# Exkurs: Summenformel

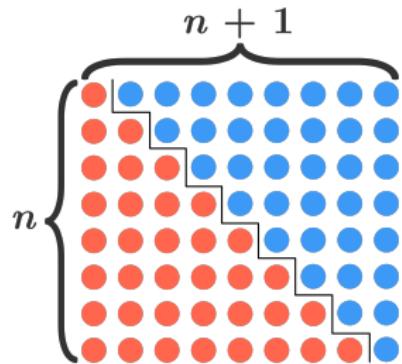
Wenn eine Laufzeit berechnet wird, braucht man oft die Summe der Zahlen von 1 bis  $n$ .

$$S_n = \sum_{i=1}^n i = \frac{n(n+1)}{2}$$

Herleitung:

$$S_n = 1 + 2 + 3 + \dots + n$$

$$S_n = n + n - 1 + n - 2 + \dots + 1$$



from brilliant.org

$$2S_n = (1 + n) + (2 + n - 1) + (3 + n - 2) + \dots + (n + 1)$$

$$= \underbrace{(n + 1) + (n + 1) + (n + 1) + \dots + (n + 1)}_{n \text{ times}}$$

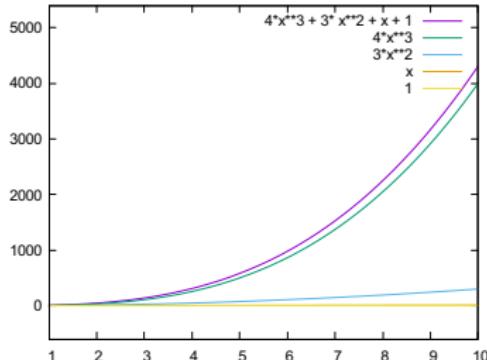
$$= n(n + 1)$$

$$S_n = \frac{n(n + 1)}{2}$$

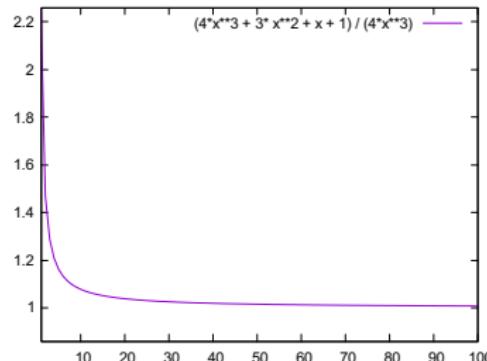
# Komplexität für große Eingaben

Der Term  $4n^3 + 3n^2 + n + 1$  ist unhandlich.

Wie verhält sich der Term für große Werte von  $n$ ?

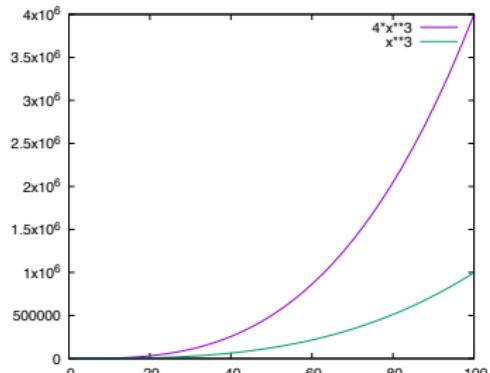


- für  $n > 5$  ist der Unterschied zwischen  $4n^3 + 3n^2 + n + 1$  und  $4 \cdot n^3$  irrelevant



- der **Abstand** wird zwar größer, ...
- ... aber das **Verhältnis** konvergiert gegen 1
- bei Polynomen ist nur der **größte Exponent** interessant  $\sim 4n^3$

# Weitere Vereinfachungen



$4n^3$  und  $n^3$  verhalten sich „ähnlich“:

- ▶ Verdoppelung von  $n \rightsquigarrow$  Verachtfachung des Funktionswertes
- ▶  $4n^3/n^3$  konvergiert gegen einen konstanten Wert (nämlich 4)

- ▶ Außerdem: konstante Faktoren oft abhängig von Implementierungs-/Sprach-/Compiler-Details
  - ▶  $a = 2 * 3 + 4; \rightsquigarrow 3$  Schritte
  - ▶  $a = 2 * 3; a = a + 4; \rightsquigarrow 4$  Schritte
- ▶  $\rightsquigarrow$  Vernachlässigung von konstanten Faktoren
- ▶ **Nicht unumstritten!**
  - ▶ Sedgewick entwickelt eigene Notation, die konstante Faktoren berücksichtigt
  - ▶ Verbesserung eines Algorithmus um Faktor 10 ist spürbar

# $\mathcal{O}$ -Notation

## $\mathcal{O}$ -Notation

Für eine Funktion  $f$  bezeichnet  $\mathcal{O}(f)$  die Menge aller Funktionen  $g$  mit

$$\exists k \in \mathbb{N} \quad \exists c \in \mathbb{R}^{\geq 0} \quad \forall n > k : g(n) \leq c \cdot f(n)$$

- ▶ Ab einer bestimmten Grenze ( $k$ ) für  $n$  ist  $g(n)$  kleiner-gleich  $c \cdot f(n)$  für einen konstanten Faktor  $c$ .
- ▶  $\mathcal{O}(f)$  ist die Menge aller Funktionen, die **langfristig nicht wesentlich schneller wachsen** als  $f$
- ▶ Statt  $g \in \mathcal{O}(f)$  sagt man oft „ $g$  ist  $\mathcal{O}(f)$ “.  
„Der Aufwand des Matrix-Multiplikations-Algorithmus ist  $\mathcal{O}(n^3)$ .“

## Beispiele: $\mathcal{O}$ -Notation

- ▶  $n^2$  ist  $\mathcal{O}(n^3)$
- ▶  $3 \cdot n^3$  ist  $\mathcal{O}(n^3)$
- ▶  $4n^3 + 3n^2 + n + 1$  ist  $\mathcal{O}(n^3)$
- ▶  $n \cdot \sqrt{n}$  ist  $\mathcal{O}(n^2)$

## Vorsicht: $\mathcal{O}$ -Notation

In der Literatur wird  $g \in \mathcal{O}(f)$  oft geschrieben als  $g = \mathcal{O}(f)$ .  
Dieses = ist nicht symmetrisch:  $n = \mathcal{O}(n^2)$ , aber  $n^2 \neq \mathcal{O}(n)$ .  
Besser:  $g \in \mathcal{O}(f)$  oder  $g$  ist  $\mathcal{O}(f)$ .

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

Für jede Funktion $f$	$f \in \mathcal{O}(f)$	
$g \in \mathcal{O}(f)$	$\Rightarrow c \cdot g \in \mathcal{O}(f)$	Konstanter Faktor
$g \in \mathcal{O}(f) \wedge h \in \mathcal{O}(f)$	$\Rightarrow g + h \in \mathcal{O}(f)$	Summe
$g \in \mathcal{O}(f) \wedge h \in \mathcal{O}(g)$	$\Rightarrow h \in \mathcal{O}(f)$	Transitivität
$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \in \mathbb{R}$	$\Rightarrow g \in \mathcal{O}(f)$	Grenzwert

# Übung: Einfache $\mathcal{O}$ -Bestimmung

Zeigen Sie:

- ▶  $3x^2 + 2x \in O(x^3)$
- ▶  $3x^2 + 2x \in O(x^2)$
- ▶  $\sin(x)^2 \in O(1)$

Lösung 1.+3. Teilaufgabe

Ende Vorlesung 3

# Grenzwertbetrachtung

- Grenzwertregel:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \in \mathbb{R} \Rightarrow g \in \mathcal{O}(f)$$

- Anschaulich:

- Wenn der Grenzwert existiert, dann steigt  $g$  langfristig höchstens um einen konstanten Faktor schneller als  $f$
- Spezialfall: Wenn der Grenzwert 0 ist, dann steigt  $f$  um mehr als einen konstanten Faktor schneller als  $g$

- Beispiel:

- $f(n) = 3n + 2, g(n) = 5n$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \frac{3}{5} \Rightarrow f \in \mathcal{O}(g)$
- $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \frac{5}{3} \Rightarrow g \in \mathcal{O}(f)$

# Ein nützliches Resultat der Analysis

## Regel von l'Hôpital

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}$$

... falls folgende Bedingungen gelten:

- $\lim_{x \rightarrow \infty} f(x)$  und  $\lim_{x \rightarrow \infty} g(x)$  sind beide 0 oder beide  $+\/-\infty$
- $\lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}$  existiert

## Beispiel

$$\lim_{x \rightarrow \infty} \frac{10 \cdot x + 5}{x^2} = \lim_{x \rightarrow \infty} \frac{10}{2 \cdot x} = 0$$

# L'Hôpital - Ableitungsregeln (1)

$f(x) = k \rightarrow f'(x) = 0$	$f(x) = \ln x \rightarrow f'(x) = \frac{1}{x}$
$f(x) = x \rightarrow f'(x) = 1$	$f(x) = \log_a x \rightarrow f'(x) = \frac{1}{x \cdot \ln a}$
$f(x) = x^k \rightarrow f'(x) = kx^{k-1}$	$f(x) = e^x \rightarrow f'(x) = e^x$
$f(x) = \frac{1}{x} \rightarrow f'(x) = -\frac{1}{x^2}$	$f(x) = a^x \rightarrow f'(x) = a^x \ln a$
$f(x) = \sqrt{x} \rightarrow f'(x) = \frac{1}{2\sqrt{x}}$	$f(x) = kg(x) \rightarrow f'(x) = kg'(x)$
$f(x) = \sin(x) \rightarrow f'(x) = \cos(x)$	$f(x) = \cos(x) \rightarrow f'(x) = -\sin(x)$

## L'Hôpital - Ableitungsregeln (2)

$$f(x) = g(x) + h(x) \rightarrow f'(x) = g'(x) + h'(x)$$

$$f(x) = g(x) \cdot h(x) \rightarrow f'(x) = g'(x)h(x) + g(x)h'(x)$$

$$f(x) = \frac{g(x)}{h(x)} \rightarrow f'(x) = \frac{g'(x)h(x) - g(x)h'(x)}{(h(x))^2}$$

$$f(x) = g(h(x)) \rightarrow f'(x) = g'(h(x))h'(x)$$

Hilfreich, aber in Prüfungen nicht verfügbar:  
**Wolfram Alpha**

<https://www.wolframalpha.com/>

# Übung: $\mathcal{O}$ -Bestimmung

- Finden Sie das kleinste  $p \in \mathbb{N}$  mit  $n \cdot \log n \in \mathcal{O}(n^p)$
- Finden Sie das kleinste  $p \in \mathbb{N}$  mit  $n \cdot (\log n)^2 \in \mathcal{O}(n^p)$
- Finden Sie das kleinste  $p \in \mathbb{N}$  mit  $2^n \in \mathcal{O}(n^p)$
- Ordnen Sie die folgenden Funktionen nach Komplexität

$$n^2 \quad \sqrt{n} \quad n \cdot 2^n \quad \log(\log(n)) \quad 2^n \quad n^{10} \quad 1,1^n \quad n^n \quad \log n$$

Anmerkung:

- $\log_b n = \frac{\ln n}{\ln b} = \frac{1}{\ln b} \cdot \ln n = c \cdot \ln n$
- $\leadsto$  die Basis der Logarithmus-Funktion bewirkt nur eine Änderung um einen konstanten Faktor.
- $\leadsto$  die Basis ist für die  $\mathcal{O}$ -Betrachtung vernachlässigbar
- Es gilt:  $(\log_b x)' = \frac{1}{x \cdot \log_e b} = \frac{1}{x \cdot \ln b}$

# Ergänzung

Frage: Sei  $f(n) = n^n$ ,  $g(n) = n \cdot 2^n$ . Gilt  $f \in O(g)$  oder  $g \in O(f)$ ?

- Antwort ist nicht offensichtlich!
- Idee: Betrachte die Ableitungen:
  - $f'(n) = n^n((\ln n) + 1) = (\ln n)n^n + n^n = (\ln n)f(n) + f(n)$
  - $g'(n) = ((\ln 2)n + 1)2^n = (\ln 2)n2^n + 2^n = (\ln 2)g(n) + \frac{g(n)}{n}$
- Also:
  - $g'$  steigt nicht wesentlich schneller als  $g$ :  $g' \in O(g)$
  - Aber:  $f'$  steigt wesentlich schneller als  $f$ :  $f' \notin O(f)$
- Alternativ:

$$\lim_{n \rightarrow \infty} \frac{n \cdot 2^n}{n^n} = \lim_{n \rightarrow \infty} \frac{n \cdot 2 \cdot 2^{(n-1)}}{n \cdot n^{(n-1)}} = \lim_{n \rightarrow \infty} \frac{2 \cdot 2^{(n-1)}}{n^{(n-1)}} = \lim_{n \rightarrow \infty} \frac{2 \cdot 2^n}{n^n} = 0$$

Also:  $g \in O(f)$  und  $f \notin O(g)$ .  $n^n$  wächst echt schneller als  $n \cdot 2^n$

## **Logarithmen**

# Logarithmus: Grundideen

## Definition: Logarithmus

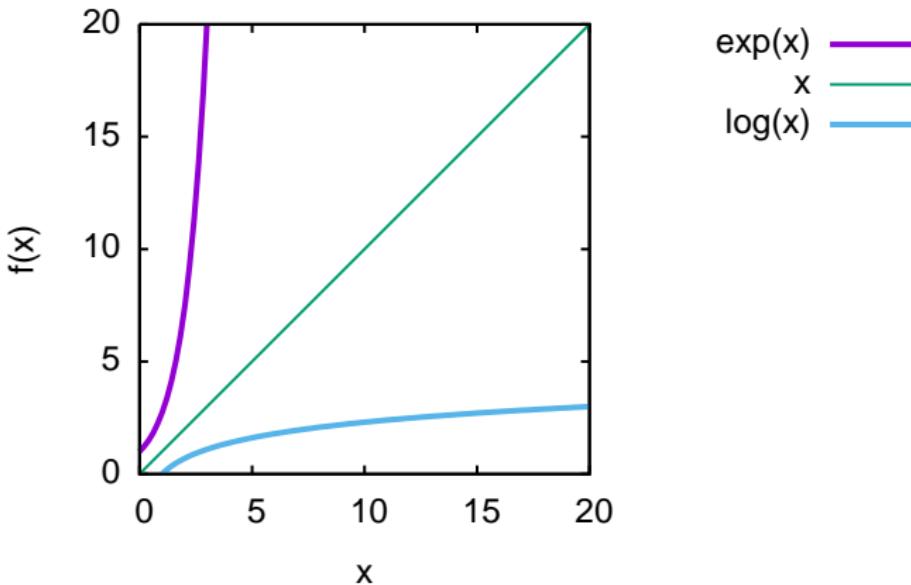
Seien  $x, b \in \mathbb{R}^{\geq 0}$  positive reelle Zahlen, sei  $b \neq 1$  und sei  $x = b^y$ . Dann heißt  $y$  der **Logarithmus von  $x$  zur Basis  $b$** .

- ▶ Wir schreiben:  $y = \log_b(x)$ .
- ▶ Der Logarithmus ist die Umkehrfunktion zur Potenzierung
  - ▶ Also:  $b^{\log_b(x)} = \log_b(b^x) = x$
  - ▶ Die Graphen von  $b^x$  und  $\log_b(x)$  gehen durch Spiegelung an der Diagonalen ineinander über
- ▶ Die Klammern lassen wir oft weg:
  - ▶  $y = \log_b x$



John Napier, 1550-1617,  
Erfinder des Begriffs  
Logarithmus

# Logarithmus illustriert



# Logarithmus: Basen (1)

- ▶ Wichtige Basen  $b$ 
  - ▶ 1 ist als Basis verboten (warum?)
  - ▶ 2 ist die für Informatiker wichtigste Basis (warum?)
    - ▶ Für  $\log_2$  schreiben wir auch Id (*logarithmus dualis*)
    - ▶ Bei uns oft implizit mit log gemeint (aber siehe auch nächste Folie)
  - ▶  $e \approx 2.71828 \dots$  hat die Eigenschaft, dass  $\frac{d}{dx} e^x = (e^x)' = e^x$  gilt
    - ▶ Also: Beim Ableiten kommt „das Selbe“ raus!
    - ▶ Die Funktion  $\log_e x$  heißt auch *natürlicher Logarithmus* und wird  $\ln(x)$  geschrieben (*logarithmus naturalis*)
    - ▶ Es gilt:  $\ln(x)' = 1/x$
  - ▶ 10 ist auf den meisten Taschenrechnern der Default

ALL YOUR  
BASE ARE  
BELONG TO  
US!

# Logarithmus: Basen (2)

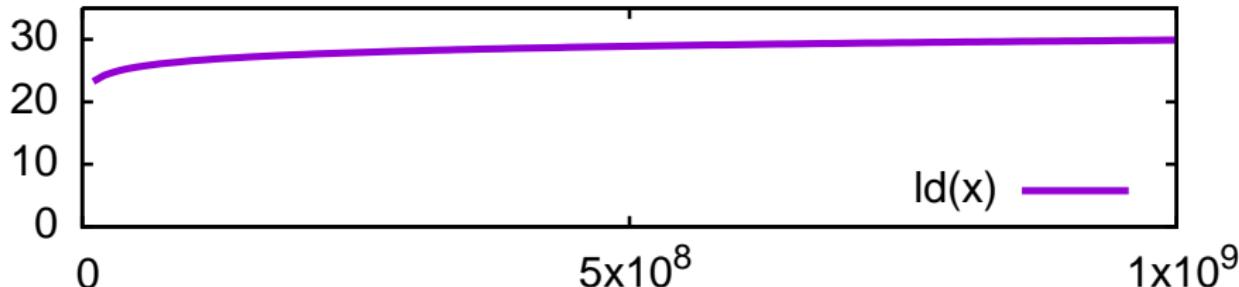
- Wir können verschiedene Basen ineinander überführen

$$\log_a x = \frac{\log_b x}{\log_b a}$$

- Also:  $\log_a x = \frac{1}{\log_b a} \log_b x = c \log_b x$
- Damit:  $\mathcal{O}(\log_a x) = \mathcal{O}(c \log_b x) = \mathcal{O}(\log_b x)$   
Die Basis ist für  $\mathcal{O}$  irrelevant!
- Mit der Basentransformation und der Produktregel ist das Ableiten der Logarithmus-Funktion zu verschiedenen Basen einfach:

$$\frac{d}{dx} \log_a x = \frac{d}{dx} \frac{\ln x}{\ln a} = \frac{d}{dx} \frac{1}{\ln a} \ln x = \frac{1}{\ln a \cdot x}$$

# Logarithmen: Wachstum



- „Für große  $n$  ist  $\log n$  annähernd konstant,”
  - $\log_2 1024 = 10$  und  $\log_2 2048 = 11$
  - $\log_2 1048576 = 20$  und  $\log_2 2097152 = 21$

Algorithmen mit logarithmischer Laufzeit sind in der Regel sehr effizient und skalieren auf sehr große Eingaben!

# Logarithmen: Anwendungen

- ▶ Wie oft kann ich eine Zahl  $x$  halbieren, bis sie 1 oder kleiner wird?
  - ▶ Antwort:  $\log_2 x$
- ▶ Wie oft kann ich eine Menge  $M$  in zwei fast gleichmächtige Teilmengen (+/- 1 Element) teilen, bis jede höchstens ein Element hat?
  - ▶ Antwort:  $\log_2 |M|$
  - ▶ Beispiel: Binäre Suche!
- ▶ Wie hoch ist ein vollständiger Binärbaum mit  $n = 2^k - 1$  Knoten?  
Mit anderen Worten: Wie lang ist der längste Ast eines solchen Baums?
  - ▶ Antwort:  $\log_2(n + 1) \approx \log_2 n$
  - ▶ Beispiel: Heap

# Logarithmen: Ausgewählte Rechenregeln

- Traditionell wichtig:

$$\log_b(x \cdot y) = \log_b x + \log_b y$$

- Begründung:  $b^x \cdot b^y = b^{x+y}$
- Anwendung: Kopfrechnen mit großen Zahlen
- Dafür: *Logarithmentafeln*
- Mini-Übung: Berechnen Sie  $2 \cdot 26$  und  $81/3$  mit Hilfe der Logarithmentafel
- $\log_b x^r = r \cdot \log_b x$

- Iterierter Anwendung der vorherigen Regel

TAFEL I.							
DE BRIGGSE LOGARITHMEN							
DER GETALLEN							
VAN 1 TOT 10000.							
N.	L.	N.	L.	N.	L.	N.	L.
1	0,00 000	26	1,41 497	51	1,70 757	76	1,88 081
2	0,30 103-	27	1,43 130	52	1,71 600	77	1,88 649
3	0,47 712	28	1,44 716-	53	1,72 428-	78	1,89 209
4	0,60 206-	29	1,46 240-	54	1,73 039	79	1,89 763-
5	0,69 897	30	1,47 712	55	1,74 036	80	1,90 309-
6	0,77 815	31	1,49 136	56	1,74 819-	81	1,90 849-
7	0,84 510-	32	1,50 515-	57	1,75 587	82	1,91 381-
8	0,90 309-	33	1,51 851	58	1,76 343-	83	1,91 908-
9	0,95 424	34	1,53 148-	59	1,77 085	84	1,92 428-
10	1,00 000	35	1,54 407-	60	1,77 815	85	1,92 942-

Logarithmentafeln gibt es (vermutlich) seit 1588 (zeitgleich schickte Phillip II die spanische Armada gegen Elisabeth I)

# Log und Exp: Ausgewählte Rechenregeln

- Eulersche Zahl:  $e = \sum_{k=0}^{\infty} \frac{1}{k!} \approx 2.71828\dots$
- $\ln e = \log_e e = 1 = e^0$  und  $e^1 = e$
- Umkehrfunktion:  $\ln e^x = x = e^{\ln x}$  und  $\log_b b^x = x = b^{\log_b x}$
- Brüche:  $\frac{c^n}{d^n} = \left(\frac{c}{d}\right)^n$  und damit
$$\lim_{n \rightarrow \infty} \frac{c^n}{d^n} = \begin{cases} 0 & \text{if } c < d \\ 1 & \text{if } c = d \\ \infty & \text{if } c > d \end{cases}$$
- Ableitung:  $(e^x)' = e^x$  und  $(b^x)' = b^x \ln b$
- Ableitung:  $(\ln x)' = \frac{1}{x}$  und  $(\log_b x)' = \frac{1}{x \ln b}$



Leonard Euler  
(1707-1783)

# Logarithmen: Übung

► Kopfberechnen Sie:

- $\log_2 1$
- $\log_2 1000$  (in etwa)
- $\log_2 2000000$  (in etwa)

► Bestimmen Sie

- $\log_8 2$
- $\log_2 \frac{1}{8}$
- $\log_8 x$  zur Basis 2

## Definition: Tiefe eines Baums

Die Tiefe eines Baumes ist die Anzahl der Knoten auf seinem längsten Ast.

Formal:

- Die Tiefe des leeren Baums ist 0.
- Die Tiefe eines nichtleeren Baums ist  $1 + \max(\text{Tiefe der Kinder der Wurzel})$ .
- Zeigen Sie: Ein Binärbaum mit  $n$  Knoten hat mindestens Tiefe  $\lceil \log_2(n + 1) \rceil$

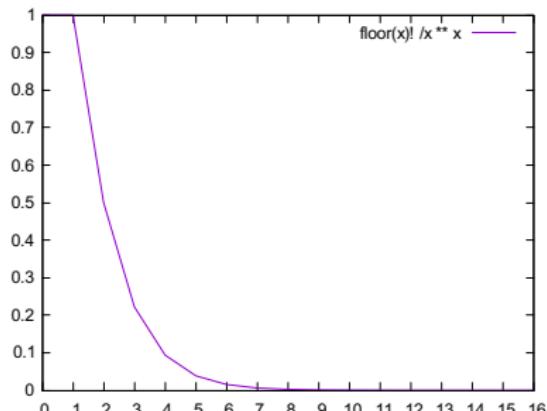
## **Komplexität (2)**

# Noch ein nützliches Resultat der Analysis

## Stirling-Formel

$$\lim_{n \rightarrow \infty} \frac{n!}{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n} = 1$$

$n!$      $\in \mathcal{O}(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n)$   
       $\in \mathcal{O}(c \cdot \frac{\sqrt{n}}{e^n} \cdot n^n)$   
       $\in \mathcal{O}(n^n)$   
       $\in \mathcal{O}(e^{n \log n})$   
 $n!$      $\notin \mathcal{O}(e^{c \cdot n})$  für irgendein  $c$

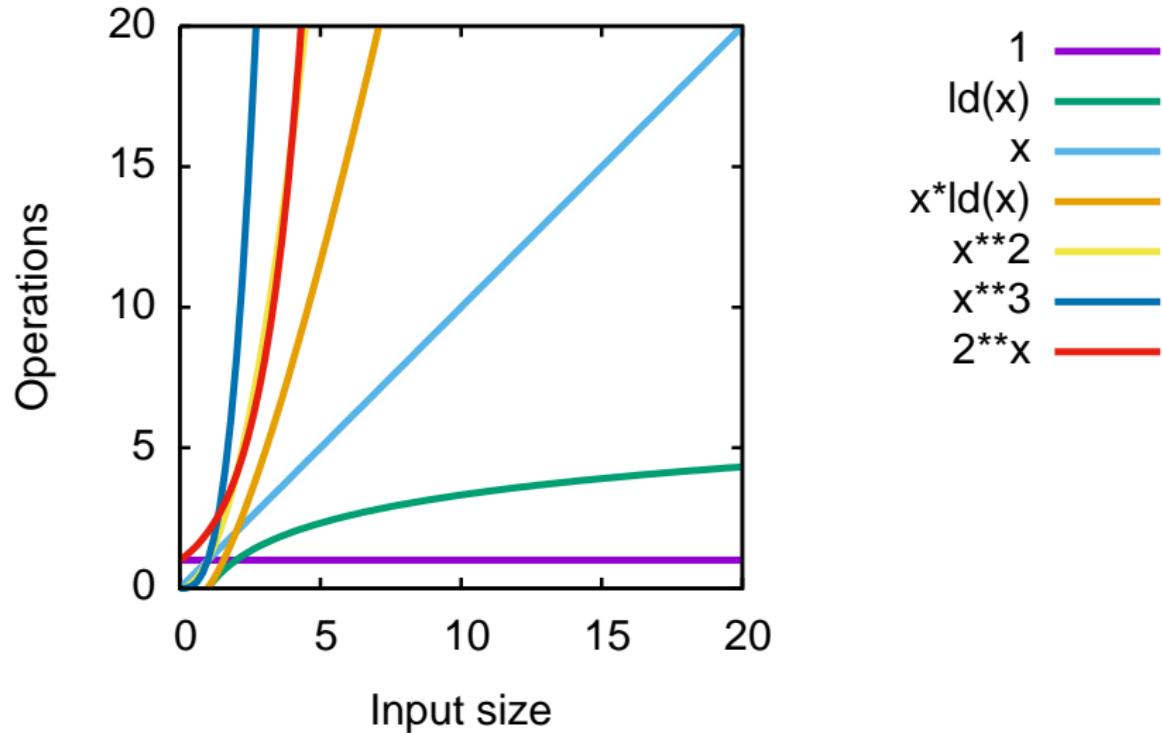


Ende Vorlesung 5

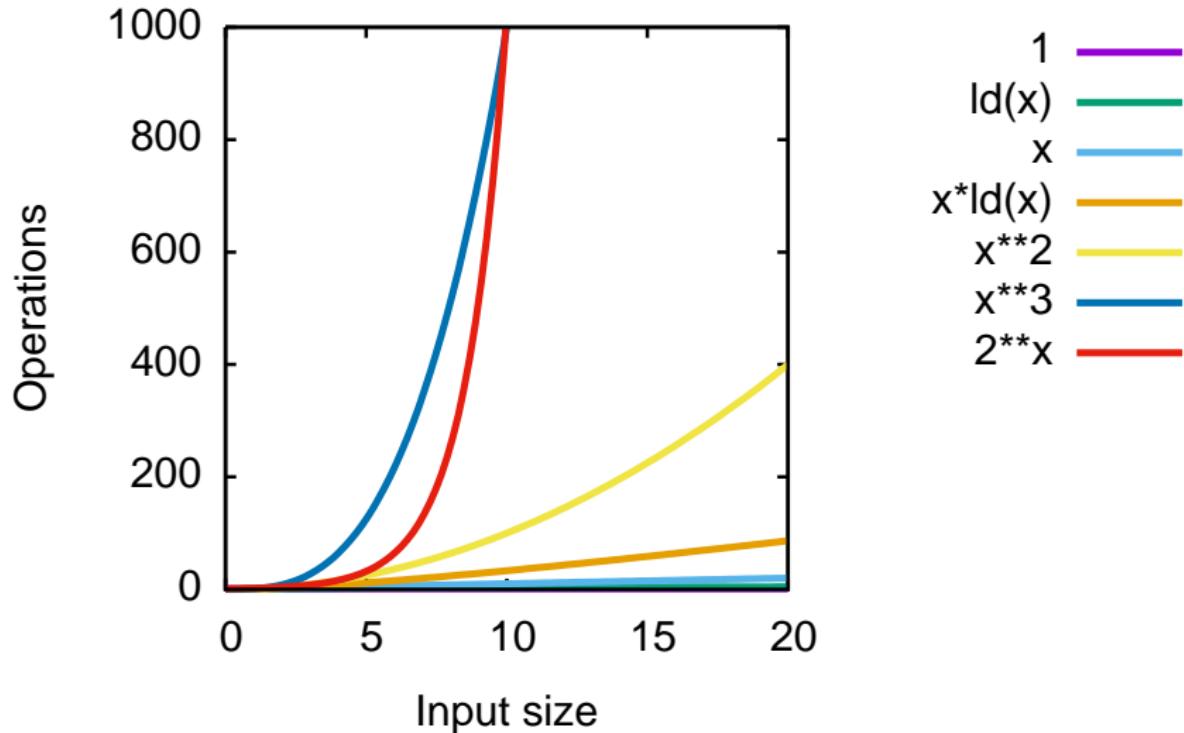
# Komplexitätsklassen verschiedener Funktionen

Ordnung	Bezeichnung	Typische Operation	Beispiel
$\mathcal{O}(1)$	konstant	elementare Operation	Zuweisung
$\mathcal{O}(\log n)$	logarithmisch	divide and conquer (ein Teil gebraucht)	binäre Suche
$\mathcal{O}(n)$	linear	alle Elemente testen	lineare Suche
$\mathcal{O}(n \log n)$	„linearithmisch“ „super-linear“	divide and conquer (alle Teile gebraucht)	effizientes Sortieren
$\mathcal{O}(n^2)$	quadratisch	jedes Element mit jedem vergleichen	naives Sortieren
$\mathcal{O}(n^3)$	kubisch	jedes Tripel	Matrix-Multiplikation
$\mathcal{O}(2^n)$	exponentiell	alle Teilmengen	Brute-Force-Optimierung
$\mathcal{O}(n!)$	„faktoriell“	alle Permutationen	Travelling Salesman Brute-Force-Sortieren
$\mathcal{O}(n^n)$		alle Folgen der Länge $n$	
$\mathcal{O}(2^{2^n})$	doppelt exponentiell	binärer Baum mit exponentieller Tiefe	

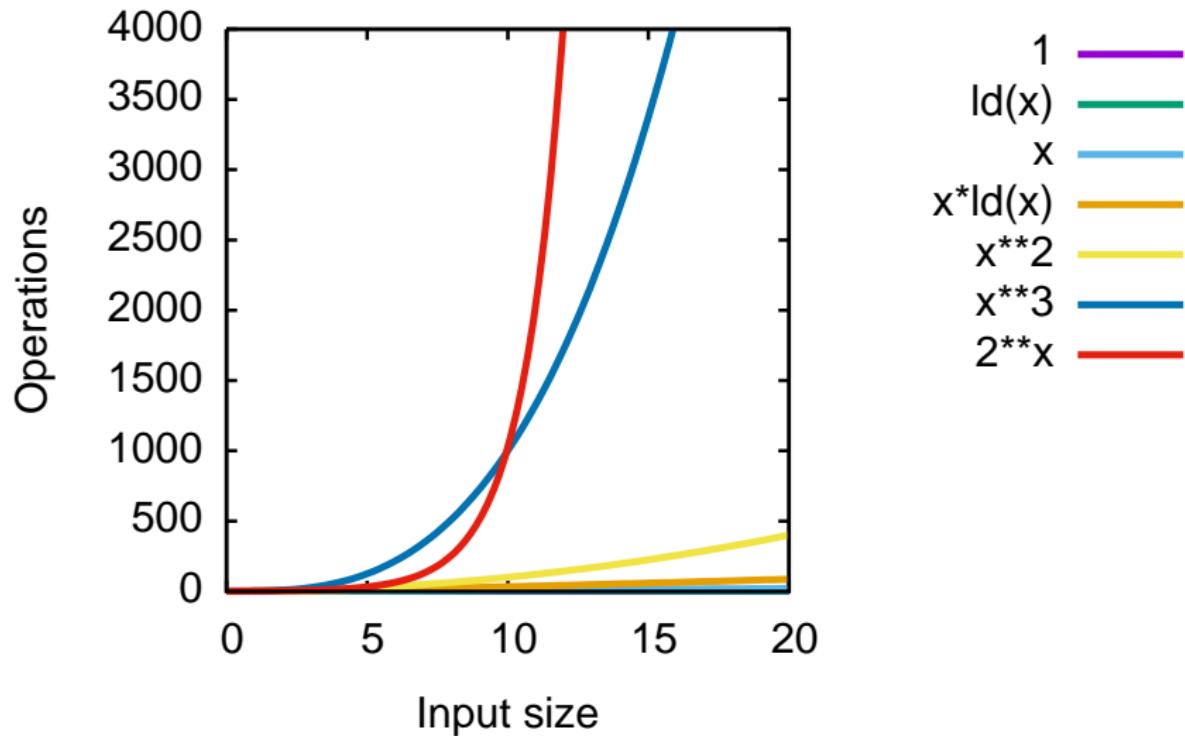
# Komplexität anschaulich



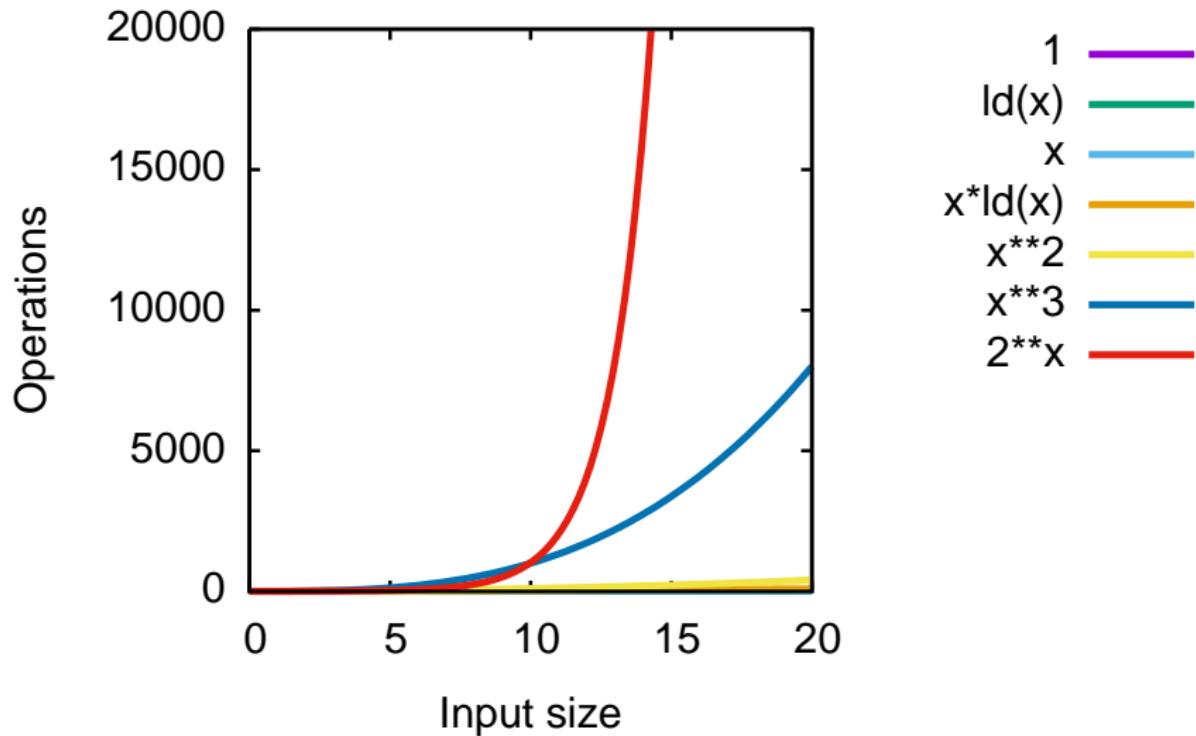
# Komplexität anschaulich



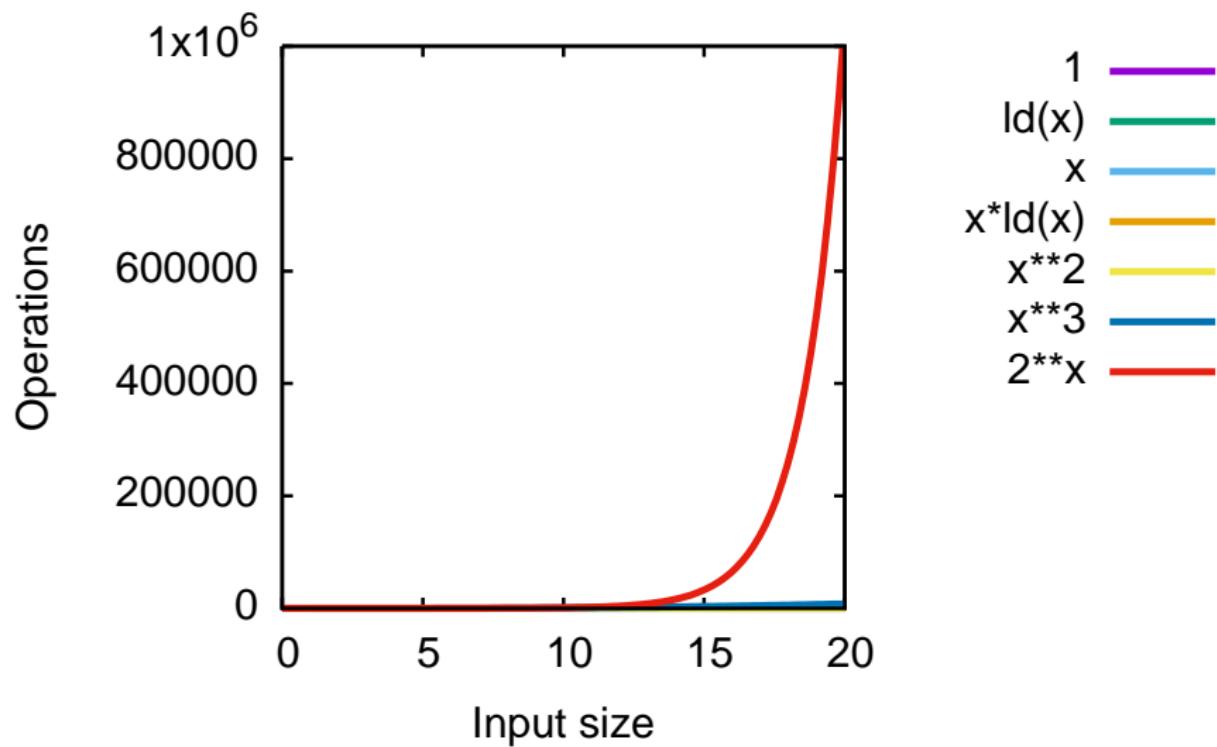
# Komplexität anschaulich



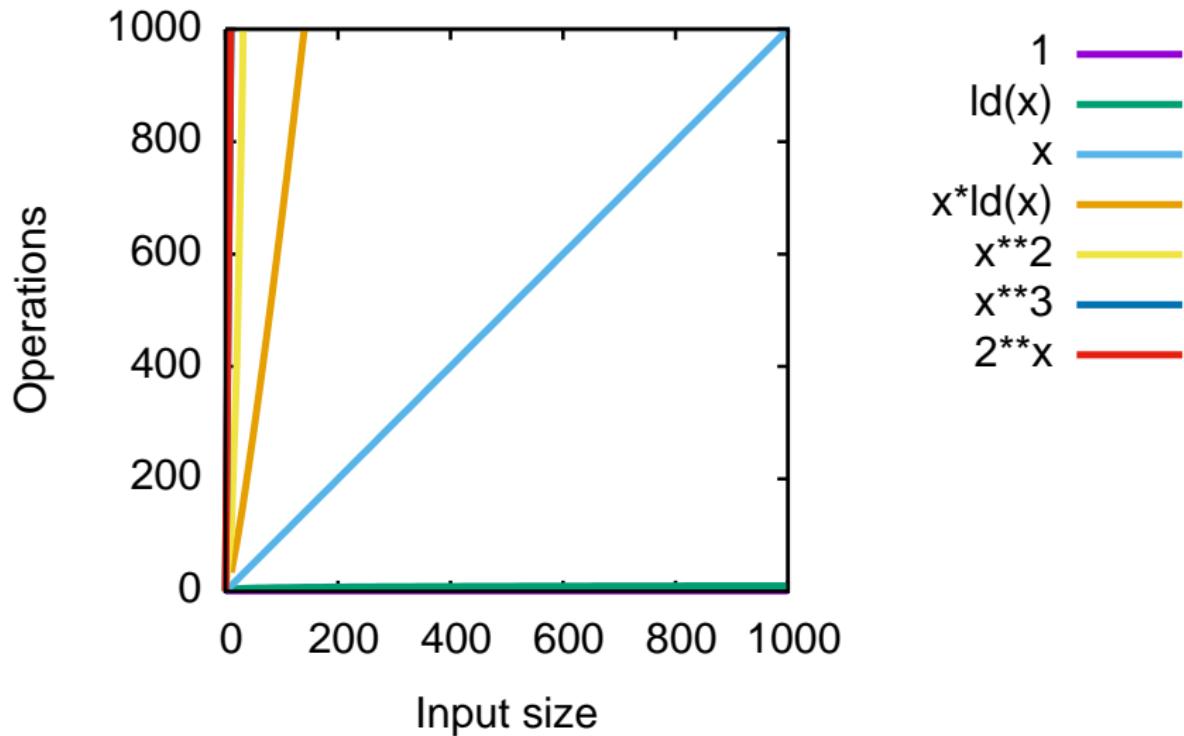
# Komplexität anschaulich



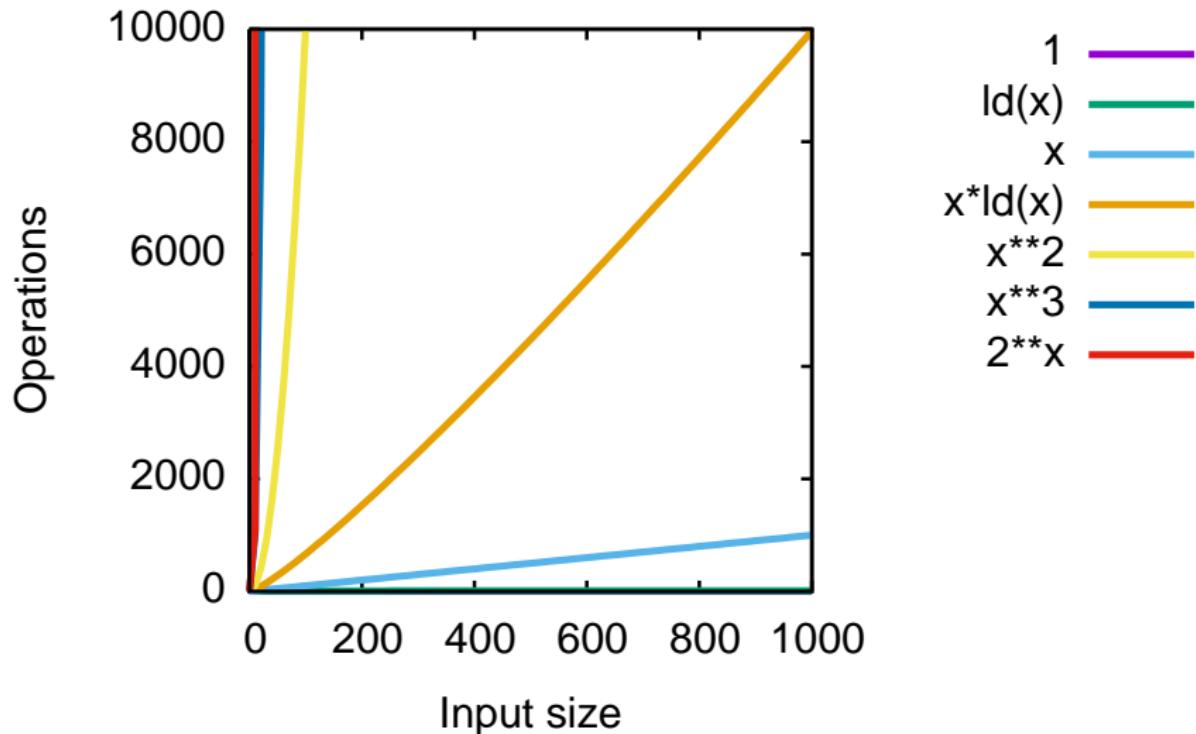
# Komplexität anschaulich



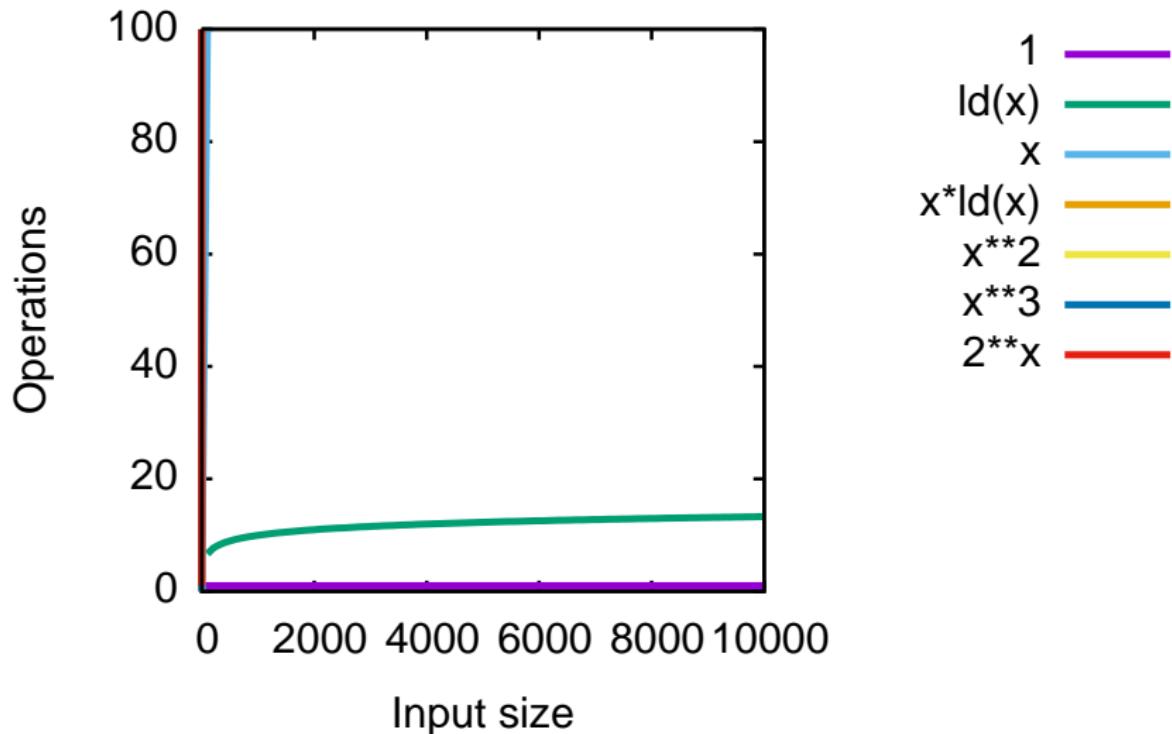
# Komplexität anschaulich



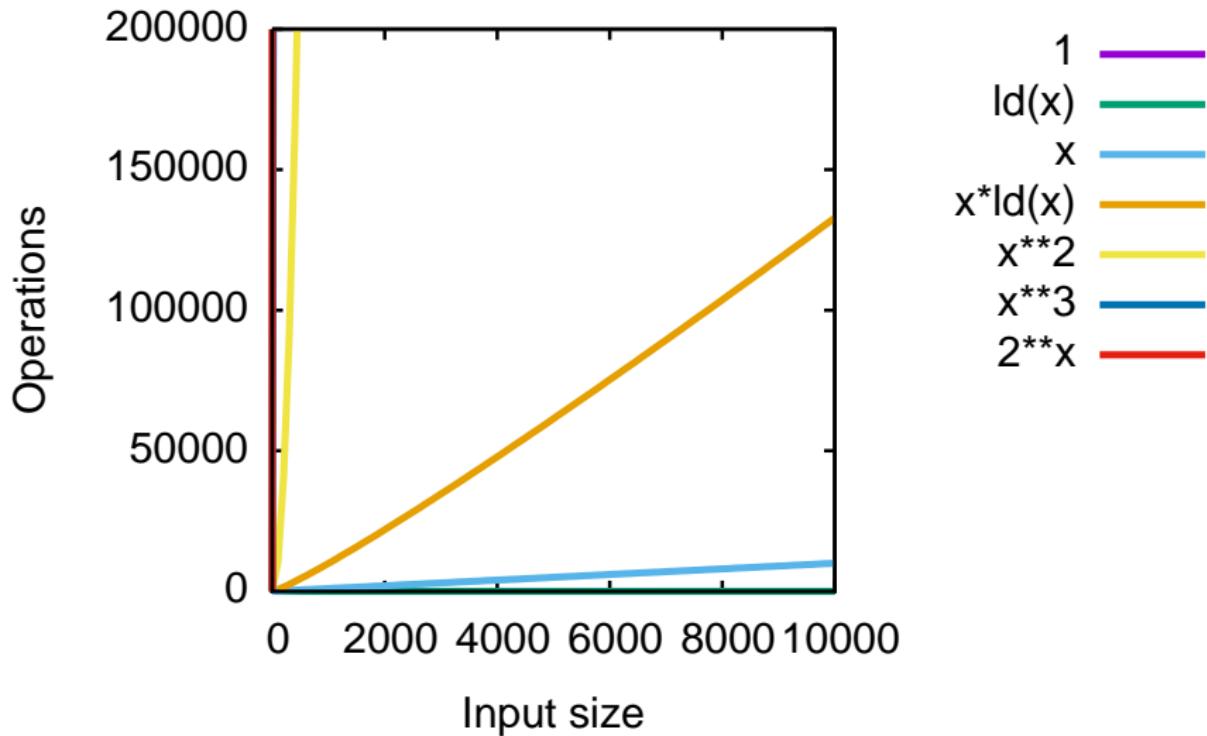
# Komplexität anschaulich



# Komplexität anschaulich



# Komplexität anschaulich



# Was geht?

- Moderne Prozessoren schaffen ca. 10 Milliarden Instruktionen pro Sekunde
- Welche Eingabegröße kann vom Prozessor verarbeitet werden?
  - Annahme: 1 Byte = 1 GE, 1 Instruktion = 1 ZE

Zeitlimit	1s	1m	1h	1d
Komplexität				
$\text{Id}(n)$	Overflow	Overflow	Overflow	Overflow
$n$	10 GB	600 GB	36 TB	864 TB
$n \cdot \text{Id}(n)$	350 MB	17 GB	900 GB	1.95 TB
$n^2$	100 KB	700 KB	6 MB	29 MB
$n^3$	2.1 KB	8.4 KB	33 KB	95 KB
$2^n$	33 B	39 B	45 B	50 B
$2^{2^n}$	5 B	5,3 B	5,5 B	5,6 B

# Mr. Universe vs. Log-Man



Geschätzte  $10^{87}$  Elementarteilchen  
im beobachtbaren Universum

1000000000000000000000000  
0000000000000000000000000  
0000000000000000000000000  
0000000000000000000000000



$$\log_2(10^{87}) \approx 289$$

Ein logarithmischer Algorithmus  
könnte das Universum  
 $\approx 35$  Millionen Mal  
pro Sekunde durchsuchen!

## Fibonacci-Zahlen

- ▶  $f(1) = 1$
- ▶  $f(2) = 1$
- ▶  $f(n) = f(n - 1) + f(n - 2)$

Erste 10 Zahlen: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

- 1 Schreiben Sie zwei Funktionen (Pseudocode oder Sprache ihrer Wahl), von denen eine  $f(n)$  rekursiv , die andere die selbe Funktion iterativ berechnet.
- 2 Bestimmen Sie dann für jede Funktion die Komplexität ( $\mathcal{O}$ -Notation).

# Optimierung des rekursiven Fib-Algorithmus

- Die naive rekursive Funktion ist ineffizient
  - Beispiel: Berechnung von  $\text{fib}(10)$
  - $1 \times \text{fib}(9), 2 \times \text{fib}(8), 3 \times \text{fib}(7), 5 \times \text{fib}(6), 8 \times \text{fib}(5), \dots$
- Rekursives Programmieren ist eleganter als iteratives
- Oft ist es nicht (leicht) möglich, eine Funktion iterativ zu berechnen (z.B. Ackermann-Funktion)
- Wie kann man den rekursiven Algorithmus auf  $\mathcal{O}(n)$  bringen?

Ansatz: [Zwischenspeichern](#) von

Teilergebnissen

- speichere jedes berechnete Ergebnis  $f(n)$
- berechne nur Ergebnisse, die noch nicht gespeichert sind
- $f$  ist  $\mathcal{O}(n)$

```
def fibd(n) :  
    if n==1:  
        return 1  
    elif n==2:  
        return 1  
    elif fib[n]==0:  
        fib[n]=fibd(n-1)+fibd(n-2)  
    return fib[n]
```

# Übung: Fibonacci dynamisch

```
def fibd(n) :  
    if n==1:  
        return 1  
    elif n==2:  
        return 1  
    elif fib[n]==0:  
        fib[n]=fibd(n-1)+fibd(n-2)  
    return fib[n]
```

- ▶ Führen Sie den Algorithmus für  $n = 7$  aus
  - ▶ Sie können davon ausgehen, dass das Array `fib` geignet groß und in allen Elementen mit 0 initialisiert ist
- ▶ Bonus-Frage: Wie lange würde das Initialisieren des Arrays `fib` dauern?

Lösung

# Dynamisches Programmieren: Technik

Die skizzierte Optimierung für die Fibonacci-Funktion wird **dynamisches Programmieren** genannt.

- ▶ Führe **komplexes** Problem auf **einfache Teilprobleme** zurück
- ▶ berechne Lösungen der Teilprobleme
  - ▶ speichere Teillösungen
- ▶ rekonstruiere Gesamtlösung aus Teillösungen

# Dynamisches Programmieren: Anwendungskriterien

Überlappende Teilprobleme Dasselbe Problem taucht mehrfach auf  
(Fibonacci)

Optimale Substruktur **Globale Lösung** setzt sich aus **lokalen Lösungen** zusammen

## Beispiel: Routing

- Der optimale Weg von Stuttgart nach Frankfurt ist 200km lang.
- Wenn der optimale Weg von Konstanz nach Frankfurt über Stuttgart führt,  
dann benötigt er 200km plus die Länge des optimalen Wegs von Konstanz nach Stuttgart

# Übung: Dynamisches Programmieren

Für eine natürliche Zahl  $n$  können 3 verschiedene Operationen durchgeführt werden:

- 1 subtrahiere 1 von  $n$
- 2 teile  $n$  durch 2, wenn  $n$  durch 2 teilbar ist
- 3 teile  $n$  durch 3, wenn  $n$  durch 3 teilbar ist

Finden Sie für ein gegebenes  $n$  die minimale Anzahl von Schritten, die nötig ist, um  $n$  zu 1 zu überführen. Vergleichen Sie die Komplexität des Brute-Force-Ansatzes mit dem des Dynamischen Programmierens.

Ein **Greedy-Algorithmus** entscheidet sich immer für denjenigen Schritt, der ihn dem Ziel am nächsten bringt (in diesem Fall: die Zahl am meisten reduziert). Führt der Greedy-Algorithmus in diesem Fall zur besten Lösung?

# Grenzen des Dynamischen Programmierens

DP ist nicht auf jedes Problem anwendbar!

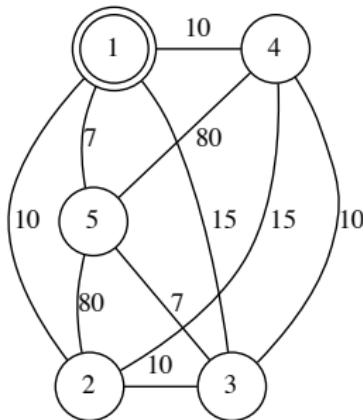
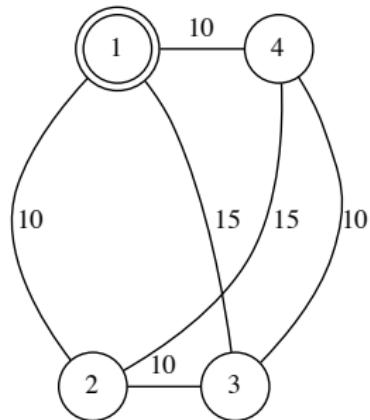
## Travelling Salesman Problem

Aufgabe: Rundreise durch Städte 1–5, minimiere die Gesamtstrecke.

Wenn zum Besuchen der Städte 1–4 die beste Reihenfolge

$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$  ist, kann die beste Reihenfolge für 1–5 auch

$1 \rightarrow 5 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 1$  sein.



# Grenzen des Dynamischen Programmierens

## Sortieren

Aufgabe: Sortiere die Liste (5, 3, 1, 2, 4)

Man kann die Aufgabe zwar aufteilen, aber in der Regel taucht jede Teilliste nur ein einziges Mal auf. Memorieren von Teilergebnissen ist also nicht hinreich.

## Dynamisches Programmieren versagt,...

- ▶ wenn Einzellösungen nicht wiederverwendet werden können
- ▶ Wenn die globale Lösung sich nicht einfach aus lokalen Lösungen zusammensetzen lässt

# Exkurs: Ackermann-Funktion

Warum überhaupt rekursiv programmieren?

$a(n, m)$

- ▶  $a(0, m) = m + 1$
- ▶  $a(n + 1, 0) = a(n, 1)$
- ▶  $a(n + 1, m + 1) = a(n, a(n + 1, m))$

$$a(1, m) = m + 2$$

$$a(2, m) = 2m + 3$$

$$a(3, m) = 8 \cdot 2^m - 3$$

$$a(4, m) = \underbrace{2^{2^{\dots^2}}}_{m+3 \text{ mal}} - 3$$

- ▶ rekursiv sehr einfach zu implementieren
- ▶ iterativ?
  - ▶ welche Werte für  $n$  und  $m$  werden zur Berechnung  $a(4, 2)$  benötigt?
  - ▶ selbst wenn die richtigen Werte gefunden sind, ist das Programm schwer verständlich

# Komplexität rekursiver Programme

## Berechnung von $m^n$

Iteratives Programm:

```
def pi(m,n):  
    p = 1  
    for x in range(n):  
        p = p * m  
    return p
```

- ▶ Zähle verschachtelte `for`-Schleifen
- ▶ Bestimme maximalen Wert der Zählvariable
- ▶  $\mathcal{O}(n)$

Rekursives Programm:

```
def pr(m,n):  
    if n==0:  
        return 1  
    else:  
        return m * pr(m,n-1)
```

- ▶ Wie Komplexität bestimmen?

# Rekurrenzrelationen

```
def pr(m,n):  
    if n==0:  
        return 1  
    else:  
        return m * pr(m,n-1)
```

Definiere  $r(n)$  als Anzahl der ZE, die zur Berechnung von  $\text{pr}(m, n)$  benötigt werden.

- ▶  $r(0) = 2$
- ▶  $r(n) = 3 + r(n - 1)$  für  $n > 0$
- ▶  $r(n) \approx 3 \cdot n \in \mathcal{O}(n)$

```
def pe(m,n):  
    if n==0:  
        return 1  
    else:  
        p = pe(m, n // 2)  
        if n % 2 == 0:  
            return p*p  
        else:  
            return p*p*m
```

- ▶  $r(0) = 2$
- ▶  $r(n) = 6 + r(\lfloor \frac{n}{2} \rfloor)$  für  $n > 0$
- ▶  $r(n) \approx 6 \cdot \log_2 n$

# Divide and Conquer

Die effiziente Lösung teilt das Problem ( $n$ -fache Multiplikation) in 2 Hälften und muss im nächsten Schritt nur ein halb so großes Problem lösen.

## Divide-and-Conquer-Algorithmus

Ein Algorithmus, der

- ▶ ein Problem in mehrere Teile aufspaltet,
- ▶ die Teilprobleme (rekursiv) löst und
- ▶ die Teillösungen zu einer Gesamtlösung kombiniert.

Ende Vorlesung 7

# Übung: Divide-and-Conquer-Suche

Schreiben Sie einen effizienten Algorithmus zur Suche in einem sortierten Array, der auf Divide and Conquer beruht.

- ▶ Gehen Sie davon aus, dass das Array von  $0 - (n - 1)$  besetzt ist
- ▶ Parameter einer rekursiven Funktion wären:
  - ▶ Das Array
  - ▶ Die Untergrenze der möglichen Positionen
  - ▶ Die Obergrenze der möglichen Positionen
  - ▶ Der zu suchende Schlüssel
- ▶ Rückgabewert ist der Index des Schlüssels, oder ein Marker für *nicht gefunden*, z.B. `None` oder `-1`

# Komplexität von rekursiven Algorithmen

- ▶ Allgemeiner Ansatz:
  - ▶ Stelle Rekurrenzrelation auf
  - ▶ Löse Rekurrenzrelation
- ▶ Spezialfall:  $r(n) = k + r(n - 1)$ ,  $r(0) = c$ 
  - ▶ Auf jeder Ebene werden  $k$  Schritte ausgeführt, Basisfall  $c$
  - ▶ Lösung:  $r(n) = k \cdot n + c$
  - ▶ Damit  $r(n) \in \mathcal{O}(n)$
- ▶ Wie sieht es aus für Divide-and-Conquer-Algorithmen?
  - ▶ Problem wird in etwa gleichgroße Teilprobleme zerlegt
  - ▶ Eines, mehrere, oder alle Teilprobleme werden gelöst
  - ▶ Die Teillösungen werden zur Gesamtlösung kombiniert

**Kosten?**

# Komplexität von Divide-and-Conquer-Algorithmen

## Master-Theorem

$$f(n) = \underbrace{a \cdot f(\left\lfloor \frac{n}{b} \right\rfloor)}_{\text{rekursive Berechnung der Teillösungen}} + \underbrace{c(n)}_{\text{Teilen und Rekombinieren}} \quad \text{mit } c(n) \in \mathcal{O}(n^d)$$

wobei  $a \in \mathbb{N}^{\geq 1}$ ,  $b \in \mathbb{N}^{\geq 2}$ ,  $d \in \mathbb{R}^{\geq 0}$ . Dann gilt:

- 1**  $a < b^d \Rightarrow f(n) \in \mathcal{O}(n^d)$
- 2**  $a = b^d \Rightarrow f(n) \in \mathcal{O}(\log_b n \cdot n^d)$
- 3**  $a > b^d \Rightarrow f(n) \in \mathcal{O}(n^{\log_b a})$

Anschaulich:

- 1** Teilen/Kombination dominiert Rekursion
- 2** Kombination und Rekursion haben gleichermaßen Einfluss auf Gesamtkomplexität
- 3** Rekursion dominiert Kombination

# Beispiel: Master-Theorem

$$f(n) = a \cdot f\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + c(n) \quad \text{mit} \quad c(n) \in \mathcal{O}(n^d)$$

Kosten für Potenzierung

$$f(n) = f\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 6$$

$$a = 1, b = 2, d = 0 \quad \Rightarrow \quad 1 = 2^0 \quad \Rightarrow \quad \text{Fall 2 : } f(n) \in \mathcal{O}(\log_b n \cdot n^d)$$

$$f(n) \in \mathcal{O}(\log_2 n \cdot n^0) = \mathcal{O}(\log n)$$

```
def pe(m,n):
    if n==0:
        return 1
    else:
        p = pe(m, n // 2)
        if n % 2 == 0:
            return p*p
        else:
            return p*p*m
```

# Übung/Hausaufgabe: Master-Theorem

1 Wenden Sie das Master-Theorem auf die folgenden Rekurrenz-Gleichungen an:

- 1  $f(n) = 4 \cdot f\left(\frac{n}{2}\right) + n$
- 2  $f(n) = 4 \cdot f\left(\frac{n}{2}\right) + n^2$
- 3  $f(n) = 4 \cdot f\left(\frac{n}{2}\right) + n^3$

2 Berechnen Sie anhand des Master-Theorems die Komplexität des Algorithmus zur binären Suche.

Lösung

Ende Vorlesung 8

# Die *Ordnungen* des Herrn Landau

- ▶  $\mathcal{O}$  steht für die *Ordnung* einer Funktion
- ▶ Eingeführt von Bachmann (1894, *Analytische Zahlentheorie*)
- ▶ Popularisiert und ergänzt von Edmund Landau (1909)
  - ▶ Nachfolger von Minkowski an der Universität Göttingen
  - ▶ Zeitgleich mit David Hilbert (*Hilbert-Programm*, *Hilberts 23 Probleme*) und Felix Klein (*Kleinsche Flasche*)
  - ▶ Von den Nazis 1934 „aus der Universität entfernt“



Edmund Georg  
Hermann  
Landau  
(1877–1938)

„Das Institut – das gibt es doch gar nicht mehr!“ - David Hilbert 1934 auf die Frage, ob das mathematische Institut in Göttingen „unter dem Weggang der Juden und Judenfreunde“ nach der Machtergreifung gelitten habe.

# Weitere Landau-Symbole

$g \in \mathcal{O}(f)$   $g$  wächst (langfristig, im wesentlichen) **höchstens** so schnell wie  $f$   
 $\lim_{x \rightarrow \infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}$

## $\Omega$ -, $\Theta$ -, $\sim$ -Notation

$g \in \Omega(f)$   $g$  wächst (langfristig, i. w.) **mindestens** so schnell wie  $f$   
 $\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = c \in \mathbb{R}$

$g \in \Theta(f)$   $g$  wächst (langfristig, i. w.) **genau** so schnell wie  $f$ ,  
bis auf einen **konstanten Faktor**  
 $\lim_{x \rightarrow \infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}^{>0}$

$g \sim f$   $g$  wächst (langfristig) **genau** so schnell wie  $f$ ,  
(konstanter Faktor 1, Sedgewick)  
 $\lim_{x \rightarrow \infty} \frac{g(x)}{f(x)} = 1$

# Übung: $\mathcal{O}, \Omega, \Theta, \sim$

- Betrachten Sie folgende Funktionen:

- $h_1(x) = x^2 + 100x + 3$
- $h_2(x) = x^2$
- $h_3(x) = \frac{1}{3}x^2 + x$
- $h_4(x) = x^3 + x$

$$g \in \mathcal{O}(f): \lim_{x \rightarrow \infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}$$

$$g \in \Omega(f): \lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = c \in \mathbb{R}$$

$$g \in \Theta(f): \lim_{x \rightarrow \infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}^{>0}$$

$$g \sim f: \lim_{x \rightarrow \infty} \frac{g(x)}{f(x)} = 1$$

Vervollständigen Sie die Tabelle. Zeile steht in Relation ... zu Spalte:

	$h_1$	$h_2$	$h_3$	$h_4$
$h_1$	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}$
$h_2$	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}$
$h_3$	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}$
$h_4$	$\Omega$	$\Omega$	$\Omega$	$\mathcal{O}, \Omega, \Theta, \sim$

# Das Master-Theorem und $\Theta$

## Master-Theorem (Version 2)

$$f(n) = \underbrace{a \cdot f\left(\left\lfloor \frac{n}{b} \right\rfloor\right)}_{\text{rekursive Berechnung der Teillösungen}} + \underbrace{c(n)}_{\text{Teilen und Rekombinieren}} \quad \text{mit } c(n) \in \Theta(n^d)$$

wobei  $a \in \mathbb{N}^{\geq 1}$ ,  $b \in \mathbb{N}^{\geq 2}$ ,  $d \in \mathbb{R}^{\geq 0}$ . Dann gilt:

- 1**  $a < b^d \Rightarrow f(n) \in \Theta(n^d)$
- 2**  $a = b^d \Rightarrow f(n) \in \Theta(\log_b n \cdot n^d)$
- 3**  $a > b^d \Rightarrow f(n) \in \Theta(n^{\log_b a})$

- Das Master-Theorem gilt auch für  $\Theta$  statt  $\mathcal{O}$
- Achtung: *Beide* Vorkommen von  $\mathcal{O}$  müssen durch  $\Theta$  ersetzt werden
- Damit: Abschätzung nach oben und unten

## Arrays

# Arrays/Felder

- ▶ Ein **Array** (deutsch: *Feld*, der Begriff ist aber mehrdeutig) ist eine Datenstruktur zur Speicherung einer Anzahl von gleichartigen Datensätzen
- ▶ Eindimensionale Standard-Arrays:
  - ▶ Zugriff auf die Elemente erfolgt über einen ganzzahligen Index
  - ▶ Arrays haben festgelegte Größe (z.B.  $n$  Elemente)
  - ▶ Indices in C laufen von 0 bis  $n - 1$ 
    - ▶ Andere Sprachen erlauben z.T. Indices von 1 –  $n$  oder von  $m - (m + n - 1)$
  - ▶ C hat keine Sicherheitsgurte
    - ▶ Keine Gültigkeitsprüfung beim Zugriff
    - ▶ Effizienz vor Sicherheit

# Arrays im Speicher

- ▶ Array-Elemente werden im Speicher sequentiell abgelegt
  - ▶ Kein Speicheroverhead pro Element!
- ▶ Die Adresse von Element  $i$  errechnet sich aus der **Basisadresse**  $b$  des Arrays, und der Größe eines einzelnen Elements  $s$ :  
$$addr(i) = b + i * s$$
- ▶ Damit gilt: Zugriff auf ein Element über den Index hat (unter den üblichen Annahmen) Kostenfunktion  $\mathcal{O}(1)$ 
  - ▶ Zugriffskosten sind unabhängig von  $i$
  - ▶ Zugriffskosten sind unabhängig von der Größe des Arrays
- ▶ Im Prinzip ist der Speicher eines modernen Rechners ein Array von Bytes
  - ▶ Adressen-Transformation ist lineare Abbildung

# Beispiel: Wegpunkt-Tabelle eines Luftraums

- ▶ Basiseigenschaften:
  - ▶ Geographische Länge (double, 8 B)
  - ▶ Geographische Breite(double, 8 B)
  - ▶ Name des Wegpunkte (char [4], 4 B)

(Leichte Vereinfachung, reale Wegpunkte haben 5 Zeichen)

- ▶ In C:

```
typedef struct waypoint
{
    double lon;
    double lat;
    char   name[4];
}waypoint;
```

```
waypoint wps[1024];
```

# Implementierung im Speicher

Index	lat	lon	name
0	60.3124	17.0122	WUXA
1	61.9811	17.9212	FARI
2	59.1373	18.1192	PIRI
3	62.3212	16.9981	BALA
4	60.0134	19.1966	KORU
...			
1023	0	0	\0\0\0\0

```
&wps = base
    &wps[0] = base
        &wps[0].lat = base
        &wps[0].lon = base+8
        &wps[0].name = base+16
    &wps[1] = base+20
        &wps[1].lat = base+20
        &wps[1].lon = base+28
        &wps[1].name = base+36
    &wps[2] = base+40
        &wps[2].lat = base+40
        &wps[2].lon = base+48
        &wps[2].name = base+56
    ...
    &wps[k] = base+k*20
        &wps[k].lat = base+k*20
        &wps[k].lon = base+k*20+8
        &wps[k].name = base+k*20+16
```

Addr	Value
base	lat1
base+4	lat2
base+8	lon1
base+12	lon2
base+16	name
base+20	lat1
base+24	lat2
base+28	lon1
base+32	lon2
base+36	name
base+40	lat1
base+44	lat2
base+48	lon1
base+52	lon2
base+56	name
base+60	lat1
base+64	lat2
base+68	lon1
base+72	lon2
base+76	name
base+80	lat1
base+84	lat2
base+88	lon1
base+92	lon2
base+96	name
...	...
base+20476	name

# Verwendung von Arrays

- ▶ Eigenständig
  - ▶ Für Daten, die weitgehend statisch sind
  - ▶ Für Daten, die eine überschaubare Maximalgröße haben (müssen)
  - ▶ Für mathematische Objekte (Vektoren, Matrizen)
- ▶ Als unterliegende Basisdatenstruktur für komplexere Datentypen
  - ▶ Listen (verwende Indexwerte zur Verkettung)
  - ▶ Bäume (verwende Indexwerte zur Verzeigerung)
  - ▶ Heaps (durch geschickte Organisation ohne(!) Verzeigerung)
  - ▶ Stacks (mit Indexwert als Stackpointer)

# Arrays zur Datenhaltung

- ▶ Arrays können als Tabellen für die Datenhaltung verwendet werden
  - ▶ Typischerweise ein **Schlüssel** per Array-Element
  - ▶ Zusätzliche Daten, die mit dem Schlüssel assoziiert sind
- ▶ Beispiele:
  - ▶ Meßreihen: Zeit (Schlüssel) und Temperatur
  - ▶ Bücher: ISBN (Schlüssel), Autor, Titel, Jahr ...
  - ▶ Studenten: Matrikelnummer, Name, Email, Notenlisten ...
- ▶ Umsetzung:
  - ▶ Array der Größe  $n$  mit  $k < n$  Einträgen
  - ▶ Zähler zeigt auf das erste freie Element

Ende	9
Index	Wert
0	Alpha
1	Beta
2	Gamma
3	Delta
4	Epsilon
5	Zeta
6	Eta
7	Theta
8	Iota
9	
10	
11	
12	
13	
14	
15	

# Standard-Operationen

- ▶ Iterieren über alle Array-Elemente
- ▶ Einfügen eines Elements
  - ▶ Am Ende
  - ▶ In der Mitte
- ▶ Löschen eines Eintrags
  - ▶ Am Ende
  - ▶ In der Mitte
  - ▶ Mit einem bestimmten Wert
- ▶ Suchen eines Eintrags
  - ▶ In unsortiertem Array
  - ▶ In sortiertem Array
- ▶ Sortieren (eigene Vorlesung)

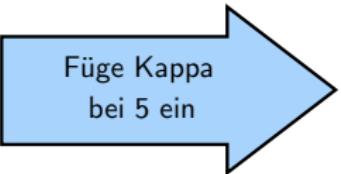
# Beispiel: Einfügen an Indexposition

Ende	9
------	---

Index	Wert
0	Alpha
1	Beta
2	Gamma
3	Delta
4	Epsilon
5	Zeta
6	Eta
7	Theta
8	Iota
9	
10	
11	
12	
12	
14	
15	

Ende	10
------	----

Index	Wert
0	Alpha
1	Beta
2	Gamma
3	Delta
4	Epsilon
5	Kappa
6	Zeta
7	Eta
8	Theta
9	Iota
10	
11	
12	
12	
14	
15	



Füge Kappa  
bei 5 ein

# Beispiel: Löschen an Indexposition

Ende	10
------	----

Index	Wert
0	Alpha
1	Beta
2	Gamma
3	Delta
4	Epsilon
5	Kappa
6	Zeta
7	Eta
8	Theta
9	Iota
10	
11	
12	
12	
14	
15	

Ende	9
------	---

Index	Wert
0	Alpha
1	Beta
2	Delta
3	Epsilon
4	Kappa
5	Zeta
6	Eta
7	Theta
8	Iota
9	
10	
11	
12	
12	
14	
15	

Lösche Eintrag

2

# Übung

- ▶ Betrachten Sie ein Array  $\text{arr}$  von  $n$  Elementen der Größe  $m$  GE, von denen die ersten  $k$  benutzt sind
- ▶ Löschen
  - ▶ Entwickeln Sie einen Algorithmus, der ein Element an der Stelle  $u$  entfernt
  - ▶ Wie viele ZE benötigt der Algorithmus?
- ▶ Einfügen (Mitte)
  - ▶ Entwickeln Sie einen Algorithmus, der ein Element an der Stelle  $u$  einfügt
  - ▶ Wie viele ZE benötigt der Algorithmus?
- ▶ Einfügen (Ende)
  - ▶ Entwickeln Sie einen Algorithmus, der ein Element an der Stelle  $k$  einfügt
  - ▶ Wie viele ZE benötigt der Algorithmus?

# Suchen im Array

- Lineare Suche:

```
def find_key(arr, n, key):  
    for i in range(0, n):  
        if arr[i].key() == key:  
            return i  
    return -1
```

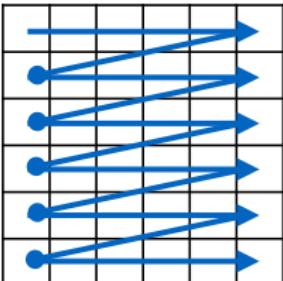
- Komplexität:  $\mathcal{O}(n)$
- Bei sortierten Array besser:
  - Binäre Suche
  - $\mathcal{O}(\log n)$
  - Tradeoff: Sortiertes Einfügen ist  $\mathcal{O}(n)$ , unsortiertes ist  $\mathcal{O}(1)$

# Zweidimensionale Arrays

- ▶ Arrays können auf den zwei- und mehrdimensionalen Fall verallgemeinert werden
  - ▶ Zeilen- oder Spalten liegen sequentiell in Speicher
- ▶ C-Familie, Python: **Row-major order**
  - ▶ `int array[5][9];`
  - ▶ Array mit 5 Zeilen a 9 Einträge (Spalten)
  - ▶ `array[2][7]:` 7. Element der 2. Zeile
- ▶ Fortran, OpenGL,...: **Column-major order**
  - ▶ `REAL, DIMENSION(9,5) :: A`

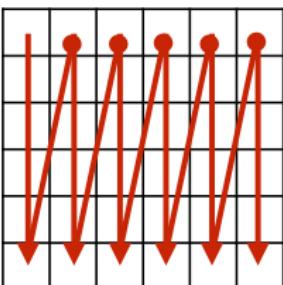
**Row-major order**

Zeilenweise Ablage



**Column-major order**

Spaltenweise Ablage



# Array Scorecard 1: Zugriff per Index

Voraussetzung:

- ▶ Standard-Array fester Größe  $m$
- ▶ Dicht gefüllt mit  $n < m$  Elementen ( $a[0] - a[n-1]$  besetzt)
- ▶ Kostenfunktion betrachtet Durchschnittsfall

Operation	Kostenfunktion
Zugriff per Index	$\mathcal{O}(1)$
Einfügen an Position $i$	$\mathcal{O}(n)$
Löschen an Position $i$	$\mathcal{O}(n)$
Array sortieren	$\mathcal{O}(n \log n)$

# Array Scorecard 2: Schlüsselverwaltung

Voraussetzung:

- ▶ Standard-Array fester Größe  $m$
- ▶ Dicht gefüllt mit  $n < m$  Elementen ( $a[0] - a[n-1]$  besetzt)
- ▶ Einträge sind Schlüssel, Schlüsselmenge ist sortierbar
- ▶ Kostenfunktion betrachtet Durchschnittsfall

Operation	Unsortiertes Array	Sortiertes Array
Schlüssel finden	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
Schlüssel einfügen	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Schlüssel bedingt einfügen	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Schlüssel löschen	$\mathcal{O}(n)$ *	$\mathcal{O}(n)$
Array sortieren	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$

Ende Vorlesung 9

---

\*Wenn der *Index* bekannt ist,  $\mathcal{O}(1)$  mit Umzugstrick.

**Listen**

# Listen

- ▶ Verkettete Listen sind eine dynamische Datenstruktur zum Speichern einer Menge von Elementen
  - ▶ Listen bestehen aus **Knoten** oder **Zellen** und **Zeigern** oder **Pointern**, die diese miteinander verbinden
  - ▶ Lineare Listen: Jede Zelle hat höchstens einen Vorgänger und höchstens einen Nachfolger
  - ▶ Jede Zelle außer der ersten hat genau einen Vorgänger
  - ▶ Jede Zelle außer der letzten hat genau einen Nachfolger
- ▶ Listeninhalte können homogen oder heterogen sein
  - ▶ Homogen: Alle Zellen haben eine *Nutzlast* vom selben Typ
  - ▶ Heterogen: Zellen haben verschiedene Nutzlasten
    - ▶ Implementierung typischerweise pseudo-homogen mit Pointern auf die tatsächliche Nutzlast
- ▶ Historisch:
  - ▶ Prozedurale Sprachen nutzen primär Arrays
  - ▶ Funktionale Sprachen nutzen primär Listen
    - ▶ LISP - LISt Processing
    - ▶ Scheme, Caml, Haskell, ...

# Einfach und doppelt verkettete Listen

- ▶ Zwei wesentliche Varianten
  - ▶ Einfach verkettete Listen („singly linked lists“)
    - ▶ Jede Zelle zeigt auf ihren Nachfolger
  - ▶ Doppelt verkettete Listen („doubly linked lists“)
    - ▶ Jede Liste hat zwei Pointer auf Vorgänger und Nachfolger

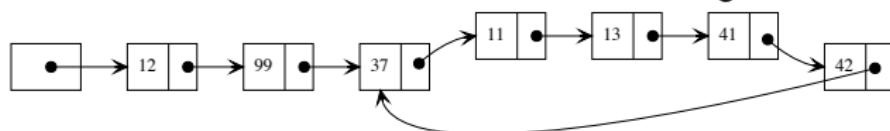
# Einfach verkettete Listen

- ▶ Listen bestehen aus Listenzellen
  - ▶ Nutzlast (Wert(e) oder Pointer)
  - ▶ Verweis auf Nachfolger (Pointer)
- ▶ Lineare Listen sind rekursiv definiert
  - ▶ Die leere Liste ist eine Liste
    - ▶ Repräsentiert als Ende-Marker (NULL/nil/' ()')
  - ▶ Eine Listenzelle, deren Nachfolger eine Liste ist, ist eine Liste
- ▶ Besonders effiziente Operationen:
  - ▶ Einfügen am Anfang (cons)
  - ▶ Ausfügen am Anfang (cdr)

Siehe auch Labor

# Zyklenerkennung

- ▶ Zyklische Listen sind keine echten Listen
  - ▶ Siehe Definition oben!
  - ▶ Sie können aber aus den selben Zellen aufgebaut werden



- ▶ Frage: Wie kann man zyklische Listen erkennen?
  - ▶ Ideen?
  - ▶ Behauptung: Das geht in  $\mathcal{O}(n)$  Zeit und mit  $\mathcal{O}(1)$  Speicher!

# Hase und Igel

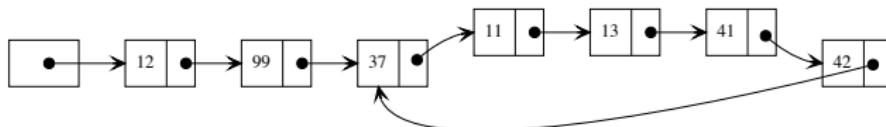
- Zyklenerkennung nach (nach Knuth) Floyd
  - Eingabe: Pointer auf eine Listenzelle
  - Ausgabe: True, falls Zyklus, False sonst

```
def list_is_cyclic(l):  
    hase = l  
    igel = l  
    while True:  
        hase = hase.succ  
        if not hase:  
            # Liste ist zyklenfrei  
            return False  
        hase = hase.succ  
        if not hase:  
            return False  
        igel = igel.succ  
        if hase == igel:  
            # Liste hat Zyklus  
            return True
```



# Übung: Hase und Igel

- ▶ Spielen Sie den Algorithmus von Hase und Igel mit folgener Liste

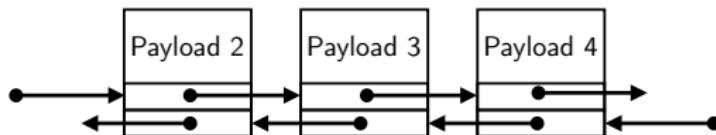


durch

- ▶ Nach wie vielen Schritten wird der Zyklus erkannt?
- ▶ Entwerfen Sie je ein weiteres Beispiel und spielen Sie es durch.
- ▶ Können Sie allgemein angeben, nach wie vielen Schritten der Algorithmus im schlimmsten Fall abbricht?

# Doppelt verkettete Listen

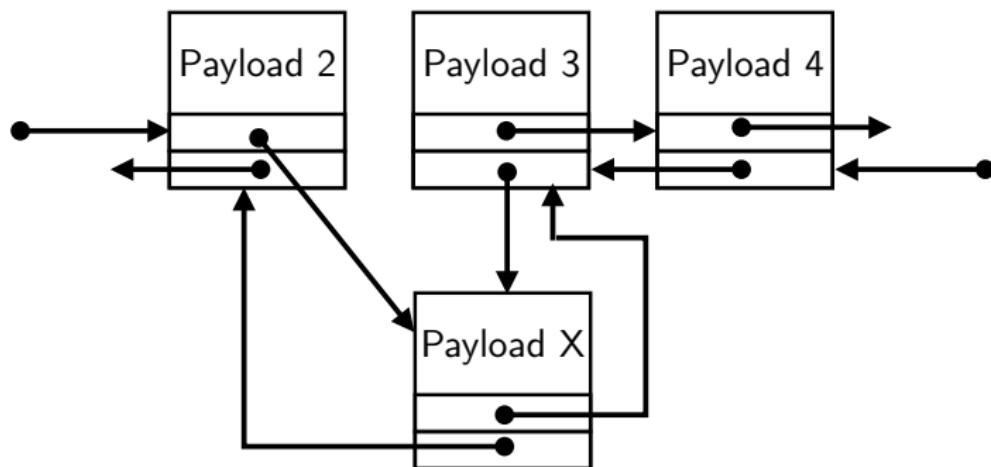
- Doppelt verkettete Listenzellen haben **zwei** Pointer
  - Auf den Vorgänger
  - Auf den Nachfolger



- Entsprechend ist die Liste vorne und hinten verankert
- Vorteile:
  - Einfügen ist vorne und hinten in  $\mathcal{O}(1)$  möglich
  - Einfügen ist vor und hinter einem beliebigen gegebenen Element einfach möglich
  - Ausfügen einer Zelle ist einfach und in  $\mathcal{O}(1)$  möglich

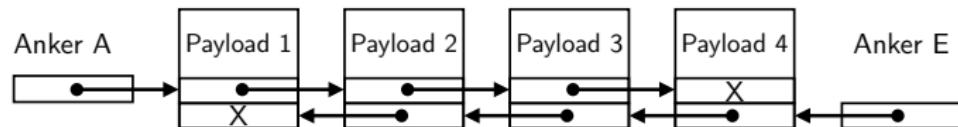
# Beispiel: Einfügen in doppelt verkettete Listen

- Einfügen vor Zelle X: *Aufschneiden* und Einsetzen



# Verankerung von doppelt verketteten Listen (1)

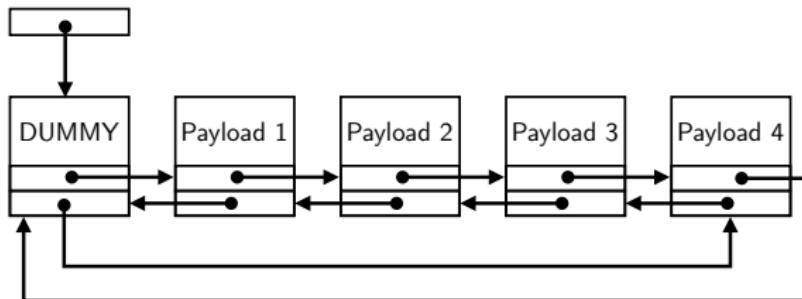
## ► Methode 1: Zwei Anker



- Zeiger auf erstes und letztes Element
- Vorteil: Minimaler Speicheroverhead
- Nachteile:
  - Bei Funktionsaufrufen müssen beide Anker übergeben werden
  - Viele Spezialfälle beim Einfügen und Ausfügen von Randzellen

# Verankerung von doppelt verketteten Listen (2)

## ► Methode 2: Zyklische Liste



- Ein Zeiger auf Dummy-Zelle
- Zellen bilden eine Ring-Struktur
- Jede Zelle hat echten Vorgänger und Nachfolger
- Vorteile:
  - Keine Spezialfälle
  - Liste wird durch einen Pointer repräsentiert
  - Keine Pointer auf Pointer notwendig ...
- Nachteil:
  - Etwas mehr Speicheroverhead (ca. 1 Payload)

# Übung: Einfügen und Ausfügen

- ▶ Gehen Sie von einem Listenzellentyp mit den Feldern `l.pred` (Vorgänger) und `l.succ` (Nachfolger) aus und nehmen Sie doppelt verkettete Listen mit zyklischer Struktur und einfachem Anker an
- ▶ Geben Sie Pseudo-Code für folgende Operationen an:
  - ▶ Einfügen einer Zelle `n` nach einer Zelle `z`, die bereits in der Liste ist, die an der Dummy-Zelle `a` verankert ist
  - ▶ Ausfügen einer Zelle `z`, die in einer Liste ist, die an der Dummy-Zelle `a` verankert ist

# Listen vs. Arrays

## Listen

Elemente liegen unabhängig im Speicher  
Länge dynamisch  
Direkter Zugriff nur auf Randelemente  
Teilen und Vereinigen einfach  
Suche immer sequentiell  
Einfügen/Ausfügen von Elementen billig

## Arrays

Elemente müssen im Speicher aufeinander folgen  
Länge in der Regel fest  
Freier Zugriff auf alle Elemente  
Teilen und Vereinigen erfordert Kopieren  
Bei sortierten Arrays binäre Suche mögliche  
Einfügen/Ausfügen bei Erhalten der Ordnung relativ teuer

# Listen Scorecard 1

Voraussetzung:

- ▶ Einfach und doppelt verkettete Listen
- ▶ Kostenfunktion betrachtet Durchschnittsfall

Operation	Einfach v.	Doppelt v.
Zugriff an Position $i$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Einfügen an Anfang	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Einfügen an Ende	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Löschen am Anfang	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Löschen am Ende	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Löschen einer gegebenen Zelle	$\mathcal{O}(n)^\dagger$	$\mathcal{O}(1)$
Liste sortieren	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$

---

<sup>†</sup>Bei \*\*-Programmierung  $\mathcal{O}(1)$  möglich.

# Listen Scorecard 2: Schlüsselverwaltung

Voraussetzung:

- ▶ Einträge sind Schlüssel
- ▶ Kostenfunktion betrachtet Durchschnittsfall

Operation	Einfach v.	Doppelt v.
Schlüssel finden	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Schlüssel einfügen	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Schlüssel bedingt einfügen	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Schlüssel löschen	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Aber beachte: Doppelt verkettete Listen haben doppelte Konstanten und doppelten Speicher-Overhead!

Ende Vorlesung 10

## **Sortieralgorithmen**

# Sortieren

Eingabe:

- ▶ Folge  $A$  von Daten  $(a_0, a_1, a_2, \dots, a_{k-1})$
- ▶ Ordnungsrelation  $\leq_A$  auf Daten

Ausgabe:

- ▶ Permutation von  $A$ :  $(a_{s_0}, a_{s_1}, a_{s_2}, \dots, a_{s_{k-1}})$
- ▶  $a_{s_i} \leq_A a_{s_{i+1}}$  für alle  $i \in \{0, \dots, k-2\}$

In der Vorlesung beispielhaft:

- ▶ Daten: natürliche Zahlen
- ▶ Ordnung:  $<_{\mathbb{N}}$

In der Praxis:

- ▶ Daten: Datensätze (Records) mit Zahl, String, Datum, ...
- ▶ Ordnung:  $<_{\mathbb{N}}, <_{\mathbb{R}}$ , lexikographisch, temporal, ...
- ▶ **Sortierschlüssel**: zum Sortieren verwendete Komponente

# Übung: Lexikographische Ordnung

Schreiben Sie eine Funktion `lexgreater (left, right)`, die für zwei Strings `left` und `right` (ohne deutsche Sonderzeichen) entscheidet, ob `left >lex right` gilt (Rückgabewert 1) oder nicht (Rückgabewert 0).

# Klassifikation von Sortieralgorithmen

- Verwendete Datenstrukturen

- Arrays Beliebiger Zugriff, Ein-/Ausfügen teuer

- Listen Sequentieller Zugriff, Ein-/Ausfügen günstig

- Bäume „Alles  $\mathcal{O}(\log n)$ “, oft als Zwischenstufe

- Verhältnis der benötigten Operationen

- Vergleiche Test der Ordnung von zwei Schlüsseln

- Zuweisungen Ändern von Elementen der Folge

- Benötigter zusätzlicher Speicher

- in-place zusätzlicher Speicherbedarf ist  $\mathcal{O}(1)$  (oft erlaubt:  
 $\mathcal{O}(\log n)$ )

- out-of-place zusätzlicher Speicherbedarf ist  $\mathcal{O}(n)$

- Stabilität: Auswirkung auf Elemente mit gleichem Schlüssel

- stabil relative Reihenfolge bleibt erhalten

- instabil relative Reihenfolge kann sich ändern

# Beispiel: In-place und Out-of-place

In-place

3	1	1	1
2	2	2	2
5	5	3	3
1	3	5	4
4	4	4	5

Out-of-place

3	1	3	1	3	1	3	1	3	1
2	2	2	2	2	2	2	2	2	2
5	5	5	5	5	5	5	5	5	5
1	1	1	1	1	1	1	1	1	1
4	4	4	4	4	4	4	4	4	4

Vorteil Zusätzlicher Speicher  $\mathcal{O}(1)$

Nachteil Nur Austausch von 2 Elementen

Vorteil Komplexere Tausch-Operationen möglich ( $\leadsto$  Mergesort)

Nachteil Zusätzlicher Speicher  $\mathcal{O}(n)$

Bei Listen oder indirektem Sortieren ist der Unterschied zwischen In-place und Out-of-place gering, da nur Zeiger / Indizes verändert werden.

# Beispiel: Stabiles und instabiles Sortieren

Original		Stabil		Instabil	
Name	Kurs	Name	Kurs	Name	Kurs
Anna	C	Bert	A	Ernie	A
Anton	B	Conny	A	Bert	A
Bert	A	Ernie	A	Conny	A
Christoph	C	Anton	B	Grit	B
Conny	A	Dirk	B	Dirk	B
Dirk	B	Grit	B	Anton	B
Ernie	A	Anna	C	Christoph	C
Frank	C	Christoph	C	Anna	C
Grit	B	Frank	C	Frank	C

- ▶ Einfache Sortierverfahren sind meist stabil, effiziente oft instabil
- ▶ Ob Instabilität ein Nachteil ist, hängt von Anwendung ab
  - ▶ behebbar durch zusätzliches eindeutiges Feld in Datenstruktur
  - ▶ bei eindeutigen Werten (z.B. Primärschlüsseln) unproblematisch

# Einfache Sortierverfahren: Selection Sort

- 1 finde kleinstes Element  $a_{\min}$  der Folge  $(a_0, \dots, a_{k-1})$
- 2 vertausche  $a_{\min}$  mit  $a_0$
- 3 finde kleinstes Element  $a_{\min}$  der Folge  $(a_1, \dots, a_{k-1})$
- 4 vertausche  $a_{\min}$  mit  $a_1$
- 5 ...

# Selection Sort auf Array

```
def sel_sort(arr):
    arrlen = len(arr)
    # i is the position to fill
    for i in range(arrlen - 1):
        # Find smallest unsorted
        min_i = i
        for j in range(i+1, arrlen):
            if arr[j] < arr[min_i]:
                min_i = j
        # Swap to correct position
        arr[min_i], arr[i] = \
            arr[i], arr[min_i]
    return arr
```

- ▶ Sortieren Sie die Folge (5, 4, 3, 1, 5, 2)
- ▶ Bestimmen Sie für die gegebene Folge und für beliebige Folgen mit  $n$  Elementen:
  - ▶ Anzahl der Daten-Vergleiche
  - ▶ Anzahl der Daten-Zuweisungen
  - ▶ In-/Out-of-Place
  - ▶ Stabilität

Lösung

# Selection Sort auf Liste

```
def selsortlist (l):
    res = NULL
    while len(l) > 0:
        i = l
        mini = i
        while (i.succ != NULL)
            i = i.succ
            if i.data < mini.data:
                mini = i
        remove(l, mini)
        append(res, mini)
    return res
```

Bestimmen Sie:

- ▶ Anzahl der Vergleiche
- ▶ Anzahl der Integer- oder Pointer-Zuweisungen
- ▶ Anzahl der Daten-Zuweisungen
- ▶ Stabilität

# Zusammenfassung Selection Sort

- ▶ Prinzip: Vertausche kleinstes Element der unsortierten Liste mit ihrem ersten Element
- ▶ Zeit-ineffizient: Andere Elemente bleiben unberührt
- ▶ Platz-effizient: Dreieckstausch zwischen erstem und kleinstem Element
- ▶ einfach zu implementieren
- ▶ In-place
- ▶ Instabil
- ▶  $\mathcal{O}(n^2)$

# Einfache Sortierverfahren: Insertion Sort

Out-of-place-Version verwendet zwei Folgen

- ▶ Eingabe *In*
- ▶ Ausgabe *Out* (anfangs leer)

- 1 verwende erstes Element von *In* als erstes Element von *Out*
- 2 füge zweites Element von *In* an korrekte Position in *Out*
- 3 füge drittes Element von *In* an korrekte Position in *Out*
- 4 ...

In-place-Version betrachtet nach Schritt *i* die ersten *i* Elemente als *Out*, die restlichen als *In*

# Insertion Sort auf Array

```
def ins_sort(arr):
    arrlen = len(arr)
    # Array arr[0]–arr[i−1] is sorted
    for i in range(1, arrlen):
        # Move # arr[i] into
        # the right position
        j = i
        while j>0 and arr[j]<arr[j-1]:
            arr[j], arr[j-1] = \
                arr[j-1], arr[j]
            j=j-1
    return arr
```

- ▶ Sortieren Sie die Folge (5, 4, 3, 1, 5, 2)
- ▶ Bestimmen Sie für die gegebene Folge und für beliebige Folgen mit  $n$  Elementen:
  - ▶ Anzahl der Daten-Vergleiche
  - ▶ Anzahl der Daten-Zuweisungen
  - ▶ In-/Out-of-Place
  - ▶ Stabilität

# Insertion Sort auf Liste

```
def inssortlist (l):
    res = NULL
    while l != NULL:
        first = l
        l = l.succ
        res = insert (res, first)
    return res

def insert (l, el):
    if l == NULL:
        el.succ = NULL
        return el
    if el.data < l.data:
        el.succ = l
        return el
    l.succ = insert(l.succ, el)
    return l
```

Bestimmen Sie:

- ▶ Anzahl der Vergleiche
- ▶ Anzahl der Integer/Pointer-Zuweisungen
- ▶ Anzahl der Daten-Zuweisungen
- ▶ Stabilität

# Zusammenfassung Insertion Sort

- ▶ Prinzip: Füge Elemente der Reihe nach an richtiger Position ein
- ▶ Zeit-ineffizient:
  - ▶ Für jede Einfüge-Operation wird gesamte Liste durchlaufen
  - ▶ Im Array müssen alle folgenden Elemente verschoben werden
- ▶ Platz-effizient: Nur ein Element als Zwischenspeicher
- ▶ einfach zu implementieren
- ▶ In-place
- ▶ Stabil
- ▶  $\mathcal{O}(n^2)$

# Effiziente Implementierung: Indirektes Sortieren

- ▶ Problem: Bei großen Datensätzen ist das Vertauschen von Einträgen sehr teuer
- ▶ Lösung: Indirektes Sortieren mit Permutationsindex
  - ▶ Vergleiche `Daten[Index[i]], Daten[Index[j]]`
  - ▶ Vertausche `Index[i], Index[j]`

The diagram illustrates the process of indirect sorting. It shows two initial arrays on the left and two final arrays on the right, separated by a curved arrow pointing from left to right.

Daten	
1	D
2	B
3	A
4	E
5	C

Index	
1	1
2	2
3	3
4	4
5	5

↷

Daten	
1	D
2	B
3	A
4	E
5	C

Index	
1	3
2	2
3	5
4	1
5	4

- ▶ Jeder Lesezugriff benötigt einen zusätzlichen Lesezugriff auf das neue Array
- ▶ Typischerweise akzeptabel, wenn der Index im Hauptspeicher ist

# Übung/Hausaufgabe: Bubble Sort

Der Sortieralgorithmus *Bubble Sort* für eine Folge  $S$  funktioniert wie folgt:

- 1 durchlaufe  $S$  von Anfang bis Ende
  - wann immer  $a_i > a_{i+1}$  gilt, vertausche  $a_i$  mit  $a_{i+1}$
- 2 wiederhole Schritt 1 solange, bis keine Vertauschungen mehr vorkommen

Aufgaben:

- 1 Sortieren Sie die Folge  $A = (3, 5, 2, 4, 1)$  mit Bubble Sort.
- 2 Welche Komplexität hat Bubble Sort ( $\mathcal{O}$ -Notation)?
- 3 Ein Nachteil von Bubble Sort liegt darin, dass kleine Elemente, die am Ende der Liste stehen, nur sehr langsam an die richtige Position „blubbern“. (*Cocktail Shaker Sort*) versucht, dieses Problem zu lösen, indem die Folge abwechselnd vom Anfang und vom Ende her durchlaufen wird. Führt dies zu einer niedrigeren Komplexität?

# Ineffizienz der einfachen Sortierverfahren

- ▶ Selection Sort
  - ▶ gesamte Folge wird durchlaufen, um ein Element an korrekten Platz zu verschieben
  - ▶ Folge besteht aus vollständig sortiertem und vollständig unsortiertem Teil
  - ▶ Informationen über nicht-minimale Elemente werden vergessen
- ▶ Insertion Sort
  - ▶ Einfügen hat linearen Aufwand
  - ▶ Array: alle folgenden Elemente verschieben
  - ▶ Liste: korrekte Position muss durch lineare Suche gefunden werden, obwohl Teilliste bereits sortiert ist
- ▶ Bubble / Shaker Sort
  - ▶ gesamte Folge wird durchlaufen, um ein Element an korrekten Platz zu verschieben
  - ▶ zusätzliche Vertauschungen

# Effiziente Sortierverfahren

- ▶ Divide and Conquer
  - ▶ Teile Folge in zwei Teile
  - ▶ sortiere Teile rekursiv
  - ▶ kombiniere sortierte Teile
  - ▶ Reihenfolge der Schritte unterschiedlich
    - ▶ **Quicksort**: Gesamtliste nach Größe der Elemente organisieren → teilen → Rekursion
    - ▶ **Mergesort**: teilen → Rekursion → Teillösungen kombinieren
- ▶ **Heapsort**: spezielle Datenstruktur
  - ▶ partiell sortierter Baum
  - ▶ partielles Sortieren erfordert  $\log n$  Schritte

# Quicksort: Prinzip

Eingabe: Folge  $S$

- 1 Wenn  $|S| \leq 1$ : fertig
- 2 Wähle **Pivot-Element**  $p \in S$ 
  - ▶ Pivot: Dreh- und Angelpunkt
  - ▶ idealerweise: Mittlere Größe
  - ▶ teile Folge in zwei Teilstufen  $S_<$  und  $S_{\geq}$ 
    - ▶  $\forall a \in S_< : a < p$
    - ▶  $\forall a \in S_{\geq} : a \geq p$
- 3 Sortiere  $S_<$  und  $S_{\geq}$  mittels Quicksort

## Übung: Partitionieren $S_<$ und $S_{\geq}$

Betrachten Sie die Folge  $S = (2, 17, 5, 3, 9, 4, 11, 13, 5)$

- ▶ Wählen Sie das Element in der Mitte der Folge als Pivot-Element  $p$  und teilen Sie die Folge in zwei Teilstufen  $S_<$  und  $S_{\geq}$ , so dass alle Elemente in  $S_<$  kleiner als das Pivot sind, und alle Elemente in  $S_{\geq}$  größer sind.
- ▶ Wiederholen Sie obige Aufgabe, aber mit dem ersten Folgenelement als Pivot-Element
- ▶ Wiederholen Sie die erste Aufgabe, aber erledigen Sie die Aufgabe nur durch Vertauschen von Elementen, nicht durch kopieren in eine neue Liste

# Partitionieren in $S_<$ und $S_{\geq}$

Idee für einen Algorithmus (nach Nico Lomuto):

- ▶ Eingabe: Folge  $S = (s_0, s_1, \dots, s_n)$
- ▶ Pivot-Element  $p = s_k \in S$
- ▶ Vorgehen
  - 1 Vertausche Pivot-Element und letztes Element
  - 2 Durchlaufe  $S$  mit zwei Zählern  $su$  und  $sp$  - initial sind beide 0
  - 3  $su$  wird bei jedem Schleifendurchlauf (aber erst am Ende) erhöht
  - 4 Wenn  $s_{su} < p$ , dann vertausche  $s_{sp}$  und  $s_{su}$  und erhöhe  $sp$
  - 5 Abbruch, wenn  $su = n - 1$
  - 6 Tausche das Pivot-Element  $s_n$  und  $s_{sp}$
- ▶ Ergebnis:  $(s_0, \dots, s_{sp-1}, p, s_{sp+1}, \dots, s_n)$ , so dass  $s_0 - s_{sp-1} < p$  und  $s_{sp+1} - s_n \geq p$

# Quicksort: Code

```
def q_part(arr, low, high):
    pivotindex = (low+high)/2
    pivotvalue = arr[pivotindex]
    # Move pivot out of place
    arr[high], arr[pivotindex] = arr[pivotindex], arr[high]
    sp = low
    for su in range(low, high):
        if arr[su]<pivotvalue:
            arr[sp], arr[su] = arr[su], arr[sp]
            sp=sp+1
    arr[sp], arr[high] = arr[high], arr[sp]
    return sp

def q_sort(arr, lo, hi):
    if lo < hi:
        pivotindex = q_part(arr, lo, hi)
        q_sort(arr, lo, pivotindex-1)
        q_sort(arr, pivotindex+1, hi)
```

# Quicksort mit Hoare-Partitionierung

```
def q_part_hoare(arr, lo, hi):
    i=lo
    j=hi
    pivotindex = (lo+hi)/2
    pivotvalue = arr[pivotindex]
    while i < j:
        while i < j and arr[i] < pivotvalue:
            i = i+1
        while i < j and arr[j] > pivotvalue:
            j = j-1
        arr[i], arr[j]=arr[j], arr[i]
        j=j-1
    return i

def q_sort(arr, lo, hi):
    if lo < hi:
        pivotindex = q_part_hoare(arr, lo, hi)
        q_sort(arr, lo, pivotindex-1)
        q_sort(arr, pivotindex+1, hi)
```

- ▶ Partitionierungs-Algorithmus von Hoare sucht “große” Elemente von links und “kleine” Elemente von rechts
- ▶ Passendes Paar ⇒ Tausch
- ▶ Abbruch: Such-Indices kreuzen sich
- ▶ Typischerweise weniger Swaps als Lomuto

**Erstaunlich schwer zu implementieren!**

# Übung: Quicksort

- Betrachten Sie die Folge  $S = (2, 17, 5, 3, 9, 4, 11, 13, 5)$
- Sortieren Sie diese mit dem Quicksort-Algorithmus
  - 1 Wählen Sie als Pivot immer das erste Element der Folge
  - 2 Wählen Sie als Pivot immer das Element in der Mitte der Folge

# Quicksort historisch

- ▶ Entwickelt von Tony Hoare (*Sir Charles Antony Richard Hoare*)
  - ▶ Austauschstudent an der *Lomonossow-Universität Moskau*
  - ▶ Anwendung: Effizientes Nachschlagen von Vokabeln für automatische Übersetzung
    - ▶ ... auf magnetischen Bändern
    - ▶ ... im Jahr 1959
- ▶ Meilenstein der Analyse: Doktorarbeit von Robert Sedgewick (1975)
  - ▶ Z.B. Auswahlmethode der Pivot-Elemente
  - ▶ Doktorvater: Donald Knuth



# Übung: Analyse Quicksort

- 1** Was ist der Best Case für Quicksort?
  - ▶ Welche Komplexität hat Quicksort dann? (Tip: Master-Theorem)
- 2** Was ist der Worst Case für Quicksort?
  - ▶ Welche Komplexität hat Quicksort dann?
- 3** Ist Quicksort stabil?
- 4** Arbeitet Quicksort in-place?

# Warum ist Quicksort effizient?

- ▶ QS sortiert zunächst grob, dann immer feiner
- ▶ Analogie mit Spielkarten:
  - 1 nach rot und schwarz aufteilen
  - 2 nach Herz und Karo bzw. Pik und Kreuz aufteilen
  - 3 nach 7–10 und B–A aufteilen
  - 4 ...
- ▶ Elemente, zwischen denen einmal ein Pivot lag, werden nie mehr verglichen
- ▶ Ineffizienz von Selection / Insertion Sort wird vermieden:
  - ▶ kein wiederholtes Durchlaufen derselben Folge
  - ▶ jede Information wird genutzt
  - ▶  $\text{arr}[i] < \text{pivot} \rightsquigarrow \text{links}$ ;  $\text{arr}[i] > \text{pivot} \rightsquigarrow \text{rechts}$

# Vor- und Nachteile von Quicksort

## Vorteile

- ▶ in der Praxis oft effizientestes Sortierverfahren (wenn optimiert)
  - ▶ Oft Standard-Sortierverfahren in C/C++ (`qsort()`)
- ▶ In-place

## Nachteile

- ▶ Auswahl des Pivot-Elements entscheidend für Effizienz
  - ▶ größtes oder kleinstes  $\rightsquigarrow$  worst-case
  - ▶ Problem bei fast sortierten Folgen und Auswahl des ersten oder letzten Elements als Pivot
  - ▶ Abhilfe: **median-of-three**: Median von erstem, letztem, mittlerem Element (aber auch keine Garantie)
- ▶ Ineffizient für kurze Folgen
  - ▶ Overhead durch Rekursion und Zeigerverwaltung
  - ▶ Abhilfe: Verwende für kurze Folgen einfaches Sortierverfahren

# Divide and Conquer

- ▶ Quicksort teilt die zu sortierende Folge nach Größe
  - ▶ Ideal: Hälfte kleine Elemente, Hälfte große Elemente
  - ▶ Problem: Wir raten den trennenden Wert  $\leadsto$  keine Garantie!
- ▶ Alternative: Teile die zu sortierende Folge nach Position(en)
  - ▶ Kann gleichmäßige Teilung garantieren
  - ▶ Aber: Sortierte Teilergebnisse können nicht einfach aneinandergehängt werden

Dieser Ansatz führt zu **Mergesort!**

# Mergesort: Prinzip

Eingabe: Folge  $S$

- 1 Wenn  $|S| \leq 1$ : gib  $S$  zurück
- 2 Teile  $S$  in zwei gleich lange Folgen  $L$  und  $R$
- 3 Sortiere  $L$  und  $R$  (rekursiv)
- 4 Vereinige  $L$  und  $R$  zu  $S'$ :
  - 1 solange  $L$  oder  $R$  nicht leer sind:
    - 2  $m := \min(l_1, r_1)$
    - 3 entferne  $m$  aus  $L$  bzw.  $R$
    - 4 hänge  $m$  an  $S'$  an
- 5 gib  $S'$  zurück

# Mergesort: Code

```
mrg_sort(arr):
    if len(arr) <= 1:
        return arr
    arr1 = arr[:len(arr)/2]
    arr2 = arr[len(arr)/2:]
    arr1 = mrg_sort(arr1)
    arr2 = mrg_sort(arr2)
    e1 = 0; e2 = 0
    for i in range(len(arr)):
        if e1 >= len(arr1):
            arr[i] = arr2[e2]; e2 = e2+1
        elif e2 >= len(arr2):
            arr[i] = arr1[e1]; e1 = e1+1
        elif arr1[e1] <= arr2[e2]:
            arr[i] = arr1[e1]; e1 = e1+1
        else:
            arr[i] = arr2[e2]; e2 = e2+1
    return arr
```

- ▶ arr: Zu sortierendes Array
- ▶ arr1: Erste Hälfte
- ▶ arr2: Erste Hälfte
- ▶ e1: Index von aktuellem Element von arr1
- ▶ e2: Index von aktuellem Element von arr2
- ▶ i: Position des nächsten Elements in der gemergten Liste

## Übung: Mergesort

Sortieren Sie die Folge  $S = (2, 17, 5, 3, 9, 4, 11, 13, 5)$  mit Mergesort.

Wenn sich eine Folge nicht in zwei exakt gleich große Hälften teilen lässt, erhält die zweite (rechte) Teilfolge das zusätzliche Element.

# Mergesort auf Liste

Unterschiede zum Array:

- ▶ Halbieren der Liste elementweise
  - ▶ „eins links, eins rechts“
  - ▶ effizienter als Halbieren in der Mitte (2 Durchläufe)
- ▶ Mischen der sortierten Listen allein durch Zeiger-Manipulation
  - ▶ wie bei Insertion Sort
  - ▶ kein Overhead durch zusätzliches Array

# Mergesort: Analyse

- 1** Was ist der Best/Worst Case für Mergesort?
- 2** Was ist die Komplexität?
- 3** Ist Mergesort stabil, ...
  - ▶ ... wenn ein Array in der Mitte geteilt wird?
  - ▶ ... wenn eine Liste elementweise in zwei Listen aufgeteilt wird?
- 4** Arbeitet Mergesort in-place?

# Warum ist Mergesort effizient?

- ▶ Mergesort sortiert zuerst im Kleinen, dann im Großen
- ▶ Da die Teillisten sortiert sind, wird das kleinste Element in  $\mathcal{O}(1)$  gefunden
- ▶ Effiziente Nutzung von Information: Elemente werden nur mit ähnlich großen verglichen (Listenanfänge)
- ▶ Kein wiederholtes Durchlaufen derselben Folge

# Optimierung von Mergesort

- ▶ Nachteil: Ineffizienz durch Rekursion bei kurzen Folgen
  - ▶ Abhilfe: Verwende Insertion Sort für kurze Folgen
- ▶ Nachteil: Overhead durch Rekursion: Sortieren beginnt erst auf maximaler Rekursionstiefe
  - ▶ Abhilfe: **Bottom-up Mergesort** (iterative Variante)
    - 1 Betrachte jedes Element als ein-elementige Liste
    - 2 Mische je zwei benachbarte Listen
    - 3 Mische benachbarte zwei-elementige Listen
    - 4 Mische benachbarte vier-elementige Listen
    - 5 ...
  - ▶ Eliminiert Overhead durch Rekursion
  - ▶ Einziger Unterschied: Jede Liste (bis auf die letzte) hat Länge  $2^i$  statt  $\frac{n}{2^i}$

# Vor- und Nachteile von Mergesort

## Vorteile

- ▶ Auch im Worst Case effizient
- ▶ Stabilität einfach erreichbar

## Nachteile

- ▶ Bei Arrays: Zusätzliches Array zum Mischen nötig
- ▶ Etwa um Faktor 2 langsamer als effiziente Implementierung von Quicksort (S. Skiena: Algorithm Design Manual, 2009)

Ende Vorlesung 13

## **Heaps und Heapsort**

# Heaps als Datenstruktur

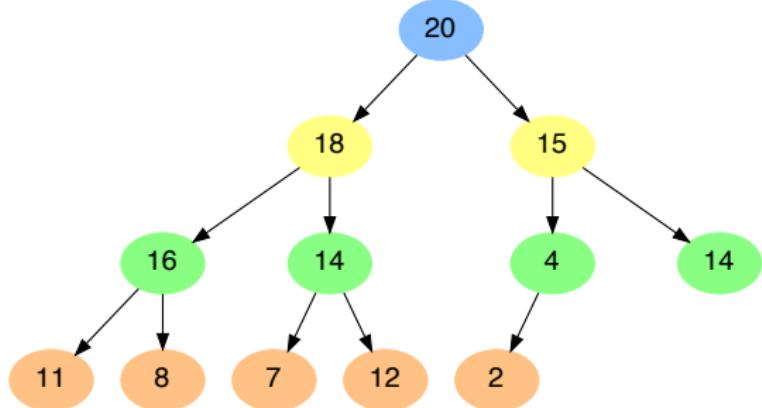
## Definition: Heap

Ein (binärer) **Heap** ist ein (fast) vollständiger binärer Baum, in dem für jeden Knoten gilt, dass er in einer definierten Ordnungsrelation zu seinen Nachfolgern steht.

- ▶ **Max-Heap:** Jeder Knoten ist  $\geq$  als seine Nachfolger
- ▶ **Min-Heap:** Jeder Knoten ist  $\leq$  als seine Nachfolger
- ▶ Fast vollständiger Binärbaum:
  - ▶ Alle Ebenen bis auf die unterste sind vollständig
  - ▶ Die unterste Ebene ist von links durchgehend besetzt
- ▶ Finden des größten (bzw. kleinsten) Elements mit  $\mathcal{O}(1)$  möglich

**Achtung:** Die Datenstruktur **Heap** ist etwas anderes, als der **Heap** (von dynamischem Speicher), der z.B. von `malloc()` und `free()` verwaltet wird!

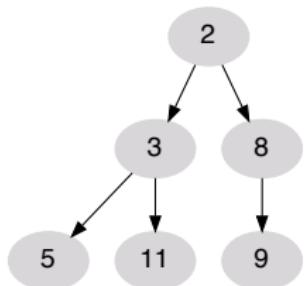
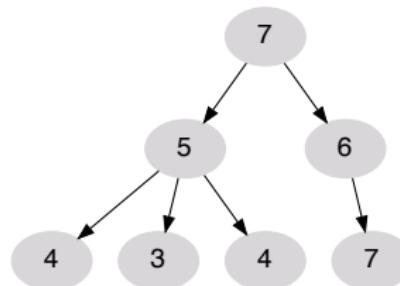
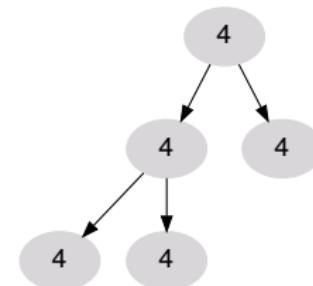
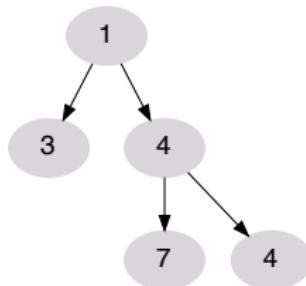
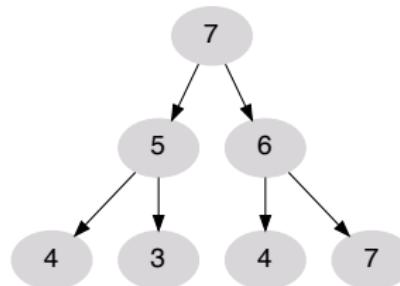
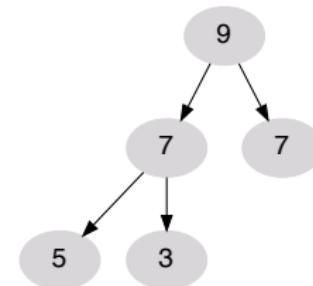
# Ein Heap



- ▶ Binärbaum:
  - ▶ Jeder Knoten hat maximal zwei Nachfolger
- ▶ Fast vollständig:
  - ▶ Alle Ebenen außer der letzten sind gefüllt
  - ▶ Auf der letzten Ebene fehlen Knoten nur rechts
- ▶ Max-Heap:
  - ▶ Jeder Knoten ist größer oder gleich seinen Nachfolgern

# Übung: Charakterisierung von Heaps

Welcher der folgenden Bäume ist ein Max-Heap? Welcher ist ein Min-Heap? Begründen Sie Ihre Entscheidung!



# Anwendungen für Heaps

- ▶ Priority-Warteschlangen (Queues)
  - ▶ Verwaltung von Aufgaben mit Prioritäten
  - ▶ Als Selbstzweck oder in anderen Algorithmen
    - ▶ CPU scheduling
    - ▶ Event handling
    - ▶ Auswahl von Klauseln zur Resolution
- ▶ Heapsort
  - ▶ Effizientes In-Place Sortierverfahren
  - ▶ Garantiert  $O(n \log n)$

# Übung: Heaps bauen

- ▶ Betrachten Sie die folgende Menge von Wörtern:  
 $W = \{da, bd, ab, aa, b, ac, cb, ba, d, bc, dd\}$
- ▶ Bauen Sie 3 verschiedene Max-Heaps für die Wörter aus  $W$ .
- ▶ Verwenden sie die lexikographische Ordnung auf Wörtern  
(also z.B.  $dd > da, da > d, \dots$ )

# Wichtige Operationen auf Heaps

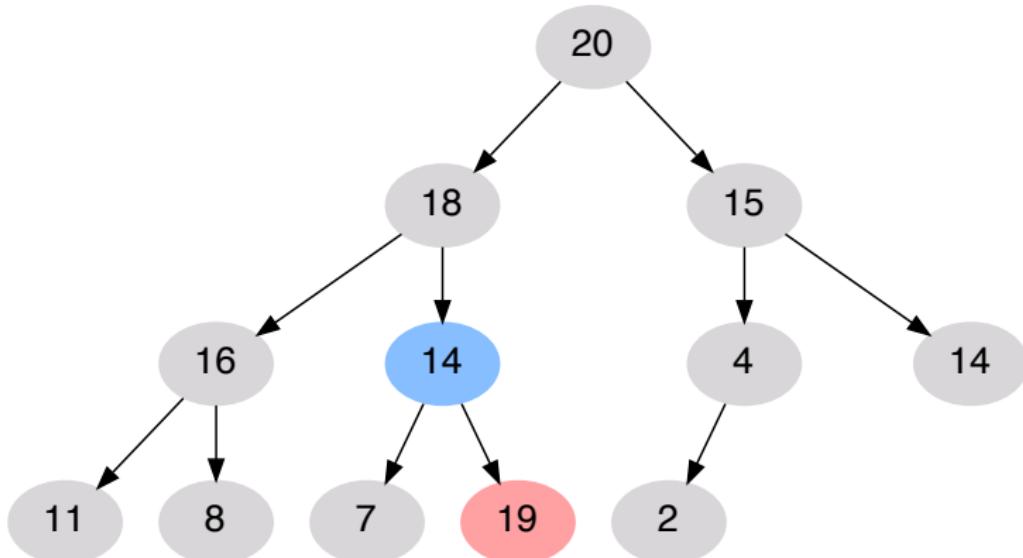
Wir gehen im folgenden immer von Max-Heaps aus. Für Min-Heaps gelten die entsprechenden Aussagen analog!

- ▶ `find_max`: Finde das maximale Element eines Heaps
  - ▶ Möglich in  $\mathcal{O}(1)$
- ▶ `heapify`: Stelle die Heap-Eigenschaft eines fast vollständigen Binärbaums her
  - ▶ `bubble_up`: Lasse einen großen Knoten nach oben steigen
  - ▶ `bubble_down`: Lasse einen kleinen Knoten nach unten sinken
- ▶ `extract_max`: Entferne das maximale Element eines Heaps und gib es zurück
  - ▶ Möglich in  $\mathcal{O}(\log n)$
- ▶ `insert`: Füge ein neues Element in den Heap ein
  - ▶ Möglich in  $\mathcal{O}(\log n)$

# Bubble-Up

Repariere Heap-Eigenschaft, wenn ein Knoten zu tief liegt

- Idee: Tausche großen Knoten mit Elternteil



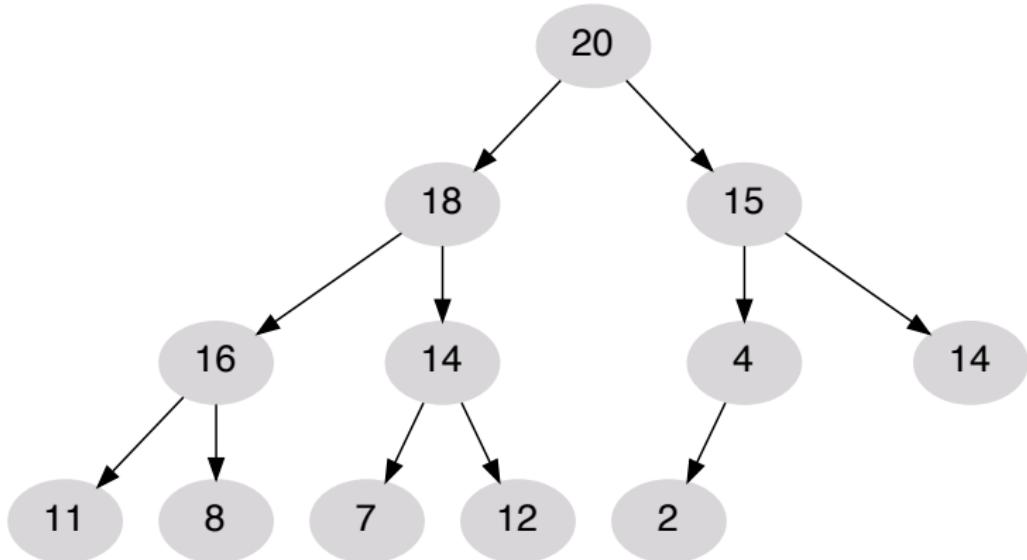
# Einfügen in Heaps

- ▶ Wir müssen zwei Eigenschaften erhalten:
  - ▶ Shape (fast vollständiger Binärbaum)
  - ▶ Heap (Kinder sind nie größer als die Eltern)
- ▶ Einfügen eines neuen Elements:
  - ▶ Füge Element am Ende des Heaps ein (linkester freier Platz auf der untersten Ebene)
    - ▶ Damit: Shape-Eigenschaft ist erhalten!
    - ▶ Heap-Eigenschaft ist i.A. verletzt
  - ▶ Dann: Bubble-up des neuen Elements
    - ▶ Dadurch: Wiederherstellung der Heap-Eigenschaft

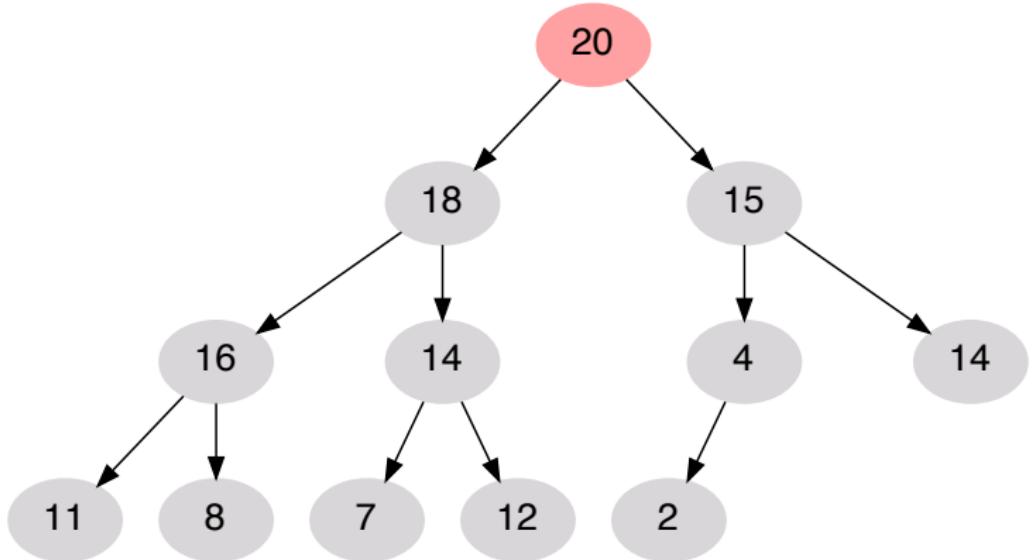
# Größtes Element entfernen und Bubble-Down

- ▶ Ziel: Größtes Element (Wurzel) aus dem Heap Entfernen
- ▶ Idee: Ersetze größtes Element durch letztes Element
  - ▶ Shape-Eigenschaft bleibt erhalten
  - ▶ Lasse neue Wurzel nach unten sinken
- ▶ Nach-unten-sinken: **Bubble-down**
  - ▶ Wenn Knoten  $\geq$  als alle Kinder: Abbruch
  - ▶ Sonst: Tausche Knoten mit seinem größten Kind
  - ▶ Wiederhole, bis Abbruch (Kinder sind kleiner oder Knoten ist Blatt)

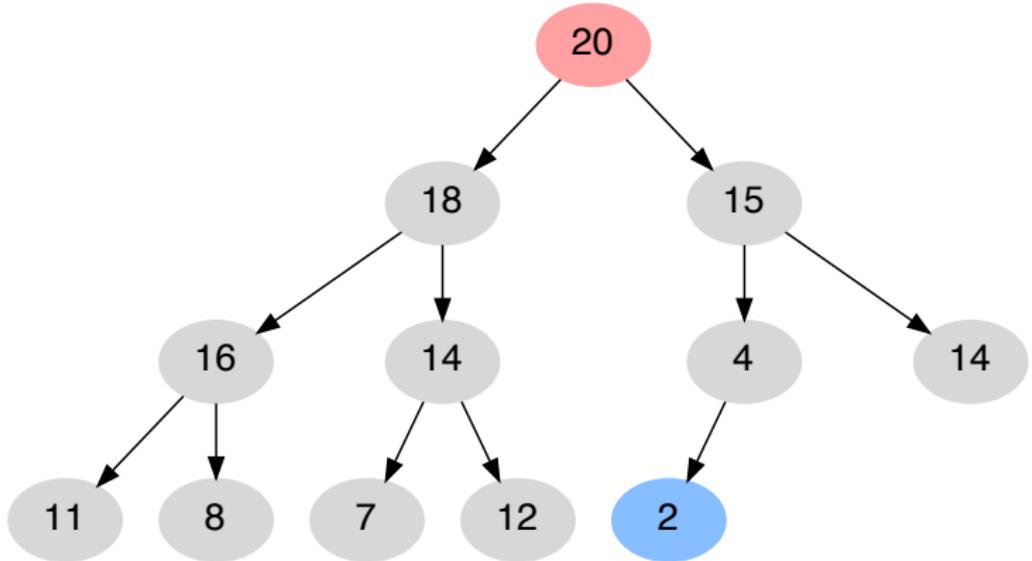
# Wurzel extrahieren



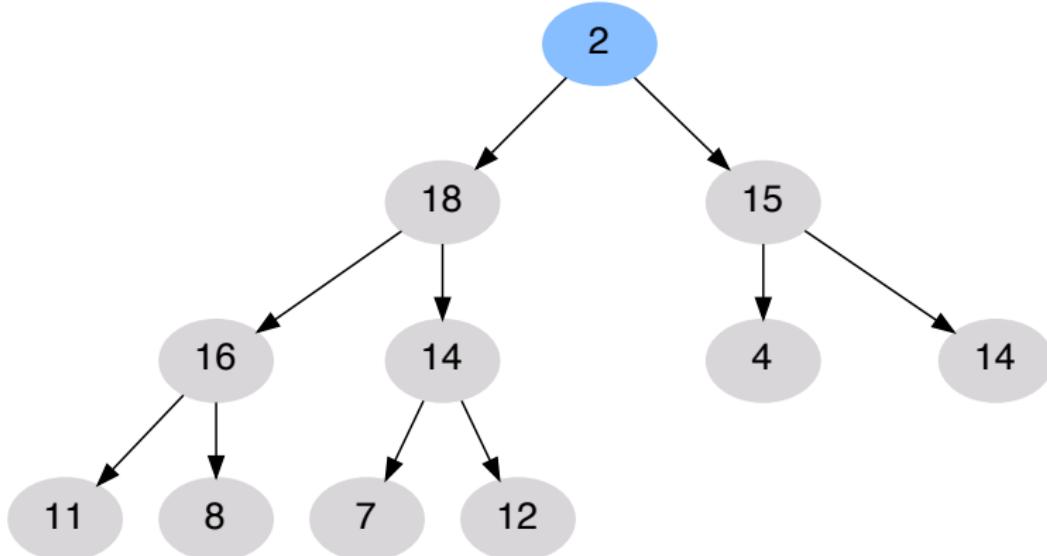
# Wurzel extrahieren



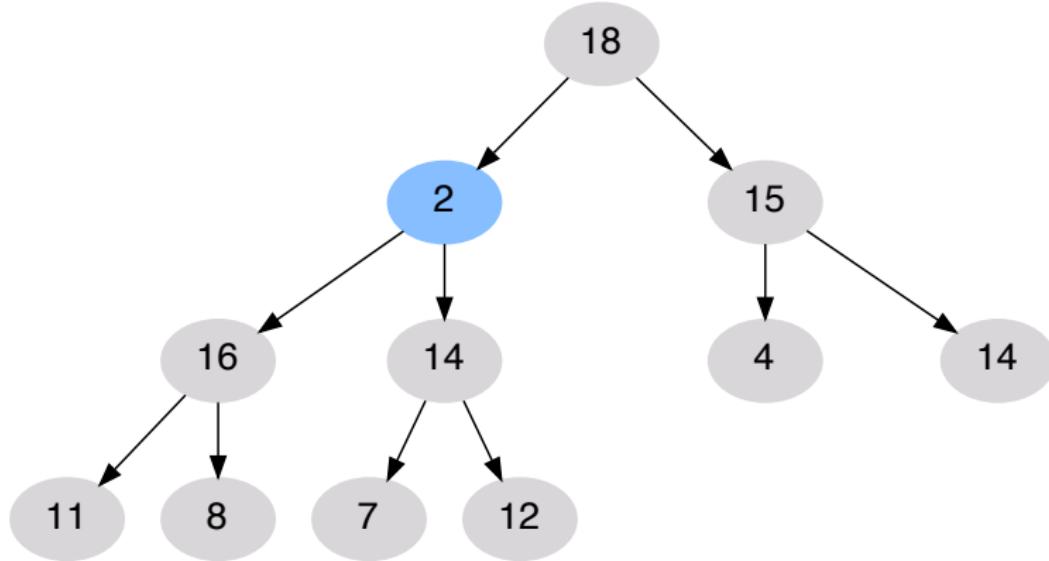
# Wurzel extrahieren



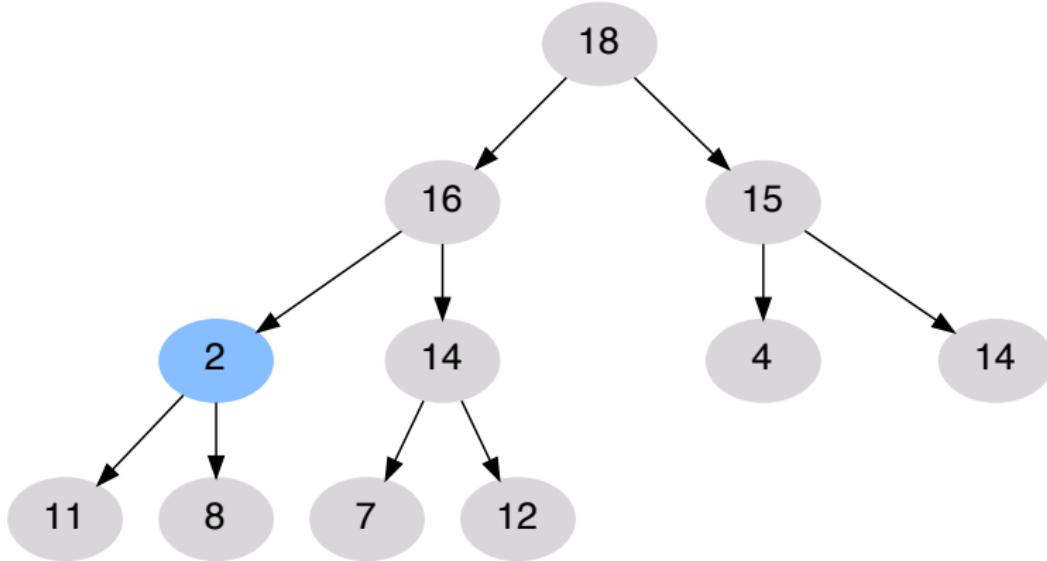
# Wurzel extrahieren



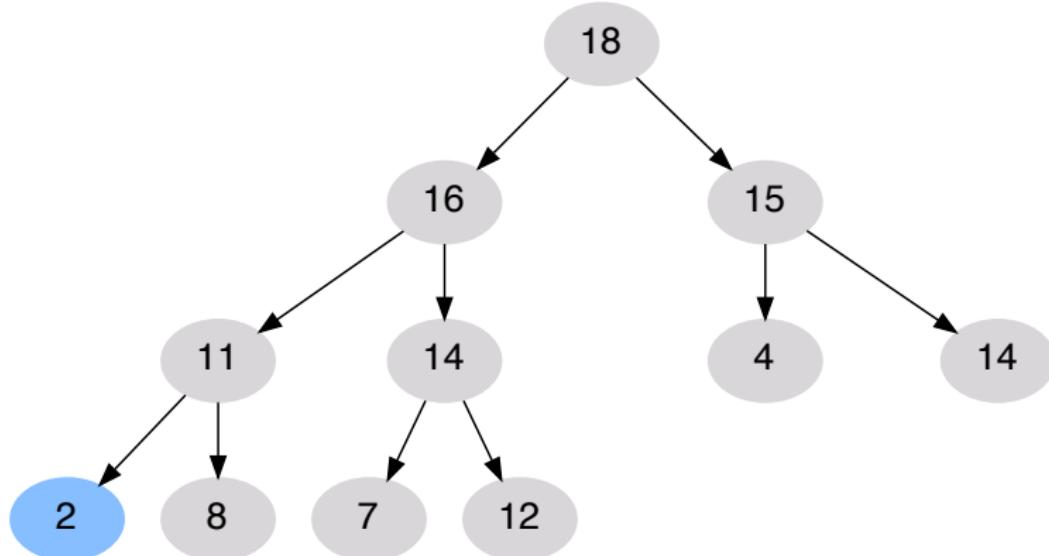
# Wurzel extrahieren



# Wurzel extrahieren

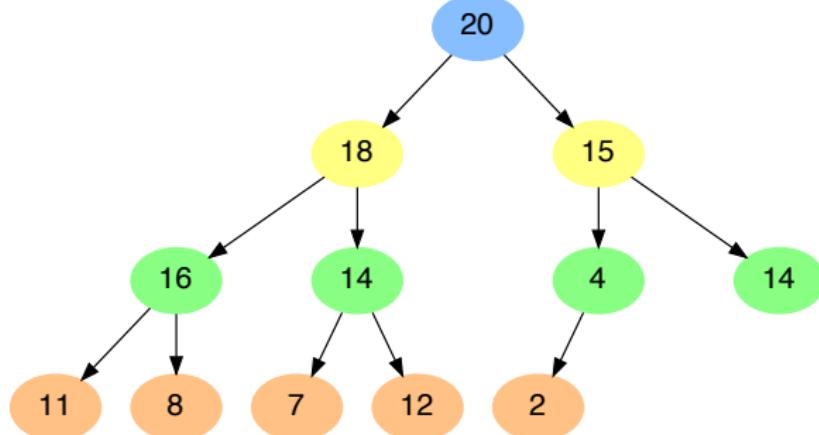


# Wurzel extrahieren



**Heap-Eigenschaft wieder hergestellt!**

# Heaps als Arrays



$$\begin{aligned}\text{lchild}(i) &= 2i + 1 \\ \text{rchild}(i) &= 2i + 2 \\ \text{parent}(i) &= \left\lfloor \frac{i-1}{2} \right\rfloor\end{aligned}$$

Idx	Wert	Tiefe
0	20	Ebene 1
1	18	Ebene 2
2	15	
3	16	
4	14	Ebene 3
5	4	
6	14	
7	11	
8	8	
9	7	
10	12	Ebene 4
11	2	
12		
13		
14		

# Basis-Operationen: Bubble-up

```
def bubble_up(arr, node)
    if node == 0
        # node is root
        return
    parent = (node - 1) // 2
    if arr[node] > arr[parent]
        arr[node], arr[parent] = \
            arr[parent], arr[node]
        bubble_up(arr, parent)
```

- ▶ Falls (Wert von) `node` größer als (Wert von) `parent(node)`, vertausche `node` und `parent(node)`
- ▶ Wiederhole den Schritt, bis er fehlschlägt
  - ▶ Fehlschlag: `node` ist Wurzel
  - ▶ Fehlschlag: `node` ist  $\leq$  seinem Elternknoten

# Basis-Operationen: Bubble-down

```
def bubble_down(arr ,node ,end)
    lci = 2*node+1
    if lci < end
        # node is not a leaf
        gci = lci
        rci = lci + 1
        if rci < end
            if arr[rci] > arr[lci]
                gci = rci
        if arr[gci] > arr[node]
            arr[node],arr[gci] = \
            arr[gci],arr[node]
    bubble_down(arr , gci , end)
```

- ▶ Falls (Wert von) `node` kleiner als eines seiner Kinder ist, vertausche `node` mit dem größten Kind
- ▶ Wiederhole den Schritt, bis er fehlschlägt
  - ▶ Fehlschlag: `node` ist Blatt
  - ▶ Fehlschlag: `node` ist  $\geq$  seinen Kindern
- ▶ Variablen:
  - ▶ `lci` - Left Child Index
  - ▶ `rci` - Right Child Index
  - ▶ `gci` - Greater Child Index

# Basis-Operationen: Heapify

```
def heapify(arr):  
    end = len(arr)  
    last = (end - 1) // 2  
    # last inner node  
    for node in range(last, -1, -1):  
        bubble_down(arr, node, end)
```

- ▶ lasse jeden inneren Knoten nach unten sinken
- ▶ beginne mit tiefstmöglicher Ebene
- ▶ Komplexität naiv:  
 $\frac{n}{2} \cdot (\log n - 1) \sim \mathcal{O}(n \cdot \log n)$

Aber:

- ▶ 1/2 der Knoten ist bereits in der unteren Ebene
- ▶ 1/4 muss höchstens 1 Ebene bubbeln
- ▶ 1/8 höchstens 2 Ebenen ...
- ▶ Und:  $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$

Also:

- ▶ Im Mittel muss ein Knoten höchstens zwei Ebenen bubbeln

**heapify**  $\in \mathcal{O}(n)$   
(!)

# Übung: Operationen auf Heaps

Gegeben sei die Menge  $W = \{da, bd, ab, aa, b, ac, cb, ba, d, bc, dd\}$ .

Erzeugen Sie aus  $W$  einen (Max-)Heap:

- 1 Beginnen Sie mit einem leeren Array, fügen Sie die Elemente der Reihe nach ein und bringen Sie sie mittels `bubble_up` an die richtige Position.
- 2 Beginnen Sie mit dem unsortierten Array und führen Sie für diesen die Funktion `heapify` durch.

Zählen Sie hierbei jeweils die Vertauschungs-Operationen.

Hinweis: Die Übung wird einfacher, wenn Sie das Array bereits in Form eines fast vollständigen Binärbaums aufschreiben!

# Heapsort: Prinzip

Eingabe: Zu sortierende Folge als Array  $A$

- 1 Transformiere  $A$  in Max-Heap (heapify)
- 2 Vertausche Wurzel ( $a_0$ ) mit letztem Element des Arrays  
(Heap endet jetzt mit vorletztem Element)
- 3 Lasse neue Wurzel herabsinken (bubble\_down)
- 4 Vertausche neue Wurzel mit vorletztem Element  
(Heap endet jetzt mit drittletztem Element)
- 5 Lasse neue Wurzel herabsinken
- 6 ...

Im Prinzip ist Heapsort ein *Selection Sort*, bei dem immer das größte Element **effizient** selektiert wird.

# Heapsort: Code

```
def heapsort(arr):
    last = len(arr)-1
    heapify(arr)
    for i in range(last,0,-1):
        arr[0],arr[i] = \
            arr[i],arr[0]
        bubble_down(arr,0,i)
    return arr
```

- ▶ arr: zu sortierendes Array
- ▶ i: Grenze zwischen Heap und sortiertem Array

# Übung: Heapsort

Betrachten Sie die Folge (22, 42, 22, 9, 3, 0, 7, 14, 15).

- ▶ Sortieren Sie diese Folge mit Heapsort
  - ▶ Schritt 1: Heapify
  - ▶ Schritt 2: Aufbau des sortierten Arrays
- ▶ Bonus: Zählen Sie die Anzahl der Vertauschungen
- ▶ Bonus: Zählen Sie die Anzahl der Vergleiche

# Warum ist Heapsort effizient?

- ▶ Entfernen des größten Elements:  $\mathcal{O}(1)$
- ▶ Wiederherstellung der Heap-Eigenschaft:
  - ▶ Vergleiche erfolgen nur entlang **eines** Pfades Wurzel → Blatt
  - ▶ Maximale Pfadlänge:  $\mathcal{O}(\log n)$
  - ▶ Elemente auf **verschiedenen** Pfaden werden **nicht** verglichen

# Heapsort: Vor- und Nachteile

## Vorteile

- ▶ In-place
- ▶  $\mathcal{O}(n \log n)$  auch im worst case

## Nachteile

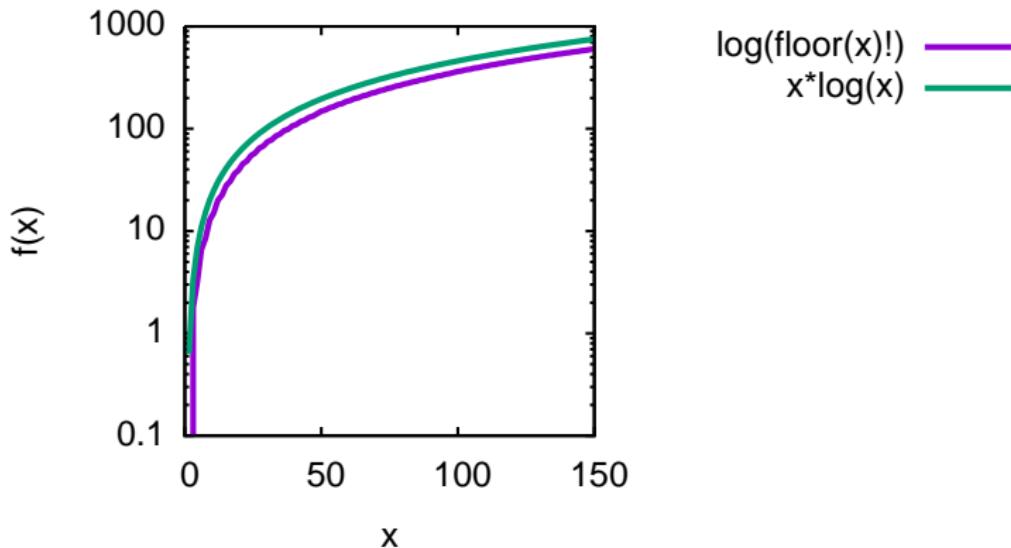
- ▶ Relativ große Konstanten
  - ▶ Erst heapify
  - ▶  $n$ -mal bubble\_down  $\leadsto n \log n$ , aber im Normalfall 2 Vergleiche pro Ebene
- ▶ funktioniert nur auf Arrays gut
- ▶ instabil

# Heapsort: Zusammenfassung

- ▶ Betrachte Array als Max-Heap
- ▶ Entferne sukzessive größtes Element
- ▶ Stelle danach Heap-Eigenschaft wieder her ( $\mathcal{O}(\log n)$ )
- ▶ in-place
- ▶ instabil
- ▶  $\mathcal{O}(n \log n)$
- ▶ erfordert Arrays

## **Sortieren – Abschluss**

# Mathematischer Fakt



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

Folgerung aus Stirling-Formel (Cormen, Leiserson, Rivest)

# Sortieren: Untere Schranke für Komplexität

- ▶ alle effizienten Sortieralgorithmen haben Komplexität  $\mathcal{O}(n \log n)$
- ▶ geht es noch besser, oder ist Sortieren  $\Theta(n \log n)$ ?
- ▶ Offensichtliche untere Schranke für Vergleiche:  $\lceil \frac{n}{2} \rceil$   
(sonst würde ein Element nicht verglichen)
- ▶ Kann man  $\mathcal{O}(n)$  erreichen?
  - ▶ Eingabe hat  $n$  Elemente
  - ▶ Ausgabe ist Permutation der Eingabe:  $n!$  Möglichkeiten  
(jede Permutation kann die richtige sein)
  - ▶ Unterschiedliche Ausgaben resultieren aus Ergebnis der Vergleiche
  - ▶  $m$  Vergleiche  $\leadsto 2^m$  mögliche Ausgaben
  - ▶  $2^m \geq n! \Rightarrow m \geq \log(n!) \Leftrightarrow m \geq n \log n$  (Stirling)
- ▶ mindestens  $n \log n$  Vergleiche für Folge der Länge  $n$  nötig!

---

Sortieren einer Folge der Länge  $n$  ist bestenfalls  $\Theta(n \log n)$ .

# Sortieren: Zusammenfassung

## Einfache Verfahren

- ▶ vergleichen jedes Paar von Elementen
- ▶ bearbeiten in jedem Durchlauf nur ein Element
- ▶ verbessern nicht die Position der anderen Elemente
- ▶  $\mathcal{O}(n^2)$

## Effiziente Verfahren

- ▶ Unterschiedliche Ansätze:
  - ▶ erst grob, dann fein: [Quicksort](#)
  - ▶ erst im Kleinen, dann im Großen: [Mergesort](#)
  - ▶ spezielle Datenstruktur: [Heapsort](#)
- ▶ Effizienzgewinn durch
  - ▶ Vermeidung unnötiger Vergleiche
  - ▶ effiziente Nutzung der in einem Durchlauf gesammelten Information
  - ▶ Verbessern der Position [mehrerer](#) Elemente in einem Durchlauf
- ▶  $\mathcal{O}(n \log n)$  (*as good as it gets*)

## **Schlüssel und Werte**

# Dictionaries

- ▶ Ziel: Dynamische Assoziation von **Schlüsseln** und **Werten**
  - ▶ Symboltabellen
  - ▶ Dictionaries
  - ▶ Look-up tables
  - ▶ Assoziative Arrays
  - ▶ ...
- ▶ Prinzip: Speichere Paare von Schlüsseln und Werten
  - ▶ Z.B. Wort und Liste von Webseiten, auf denen es vorkommt
  - ▶ Z.B. Matrikelnummer und Stammdatensatz
  - ▶ Z.B. Datum und Umsatz
  - ▶ Z.B. KFZ-Kennzeichen und Wagenhalter
  - ▶ Z.B. Name und Mitarbeiterstammdaten
  - ▶ Häufig: Schlüssel (String) und Pointer auf beliebige Daten
  - ▶ ...

# Operationen auf Dictionaries

- ▶ Wichtige Operationen auf Dictionaries
  - ▶ Anlegen eines leeren Dictionaries
  - ▶ Einfügen von Schlüssel/Wert-Paaren
  - ▶ Löschen von Schlüssel/Wert-Paaren (normalerweise anhand eines Schlüssels)
  - ▶ Finden von Schlüssel/Wert-Paaren anhand eines Schlüssels
  - ▶ Geordnete Ausgabe aller Schlüssel/Wert Paare
- ▶ Zu klärende Frage: Mehrfachschlüssel
  - ▶ Mehrfachschlüssel verbieten
  - ▶ Ein Schlüssel, Liste/Menge von Werten
  - ▶ Mehrfachschlüssel erlauben

# Übung: Dictionaries

- ▶ Wie können Sie den *abstrakten Datentyp* Dictionary mit den bekannten Datenstrukturen realisieren?
  - ▶ Betrachten Sie verschiedene Optionen!
- ▶ Welchen Komplexitäten haben die Kernoperationen?
  - ▶ Nehmen Sie an, dass das Dictionary  $n$  Schlüssel enthält

Ende Vorlesung 15

## **Binäre Suchbäume**

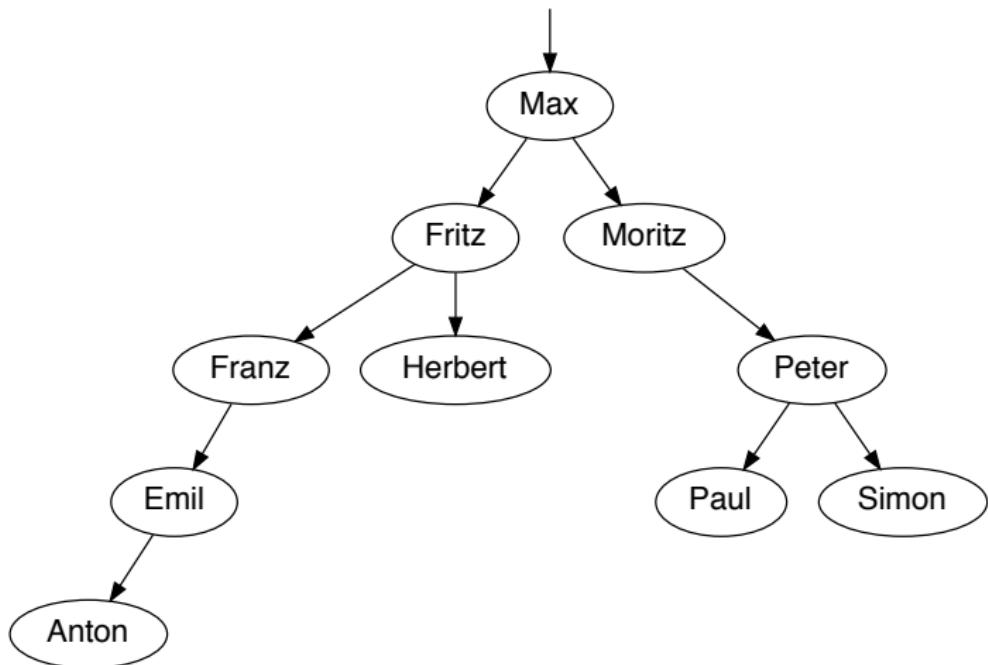
# Binäre Suchbäume

## Definition: Binärer Suchbaum

Eine **binärer Suchbaum** ist ein Binärbaum mit folgenden Eigenschaften:

- ▶ Die Knoten des Baums sind mit Schlüsseln aus einer geordneten Menge  $K$  beschriftet
- ▶ Für jeden Knoten  $N$  gilt:
  - ▶ Alle Schlüssel im linken Teilbaum von  $N$  sind kleiner als der Schlüssel von  $N$
  - ▶ Alle Schlüssel im rechten Teilbaum von  $N$  sind größer als der Schlüssel von  $N$
- ▶ Geordnete Menge  $K$ :
  - ▶ Auf  $K$  ist eine Ordnungsrelation  $>$  definiert
  - ▶ Wenn  $k_1, k_2 \in K$ , dann gilt entweder  $k_1 > k_2$  oder  $k_2 > k_1$  oder  $k_1 = k_2$

# Binärer Suchbaum – Beispiel

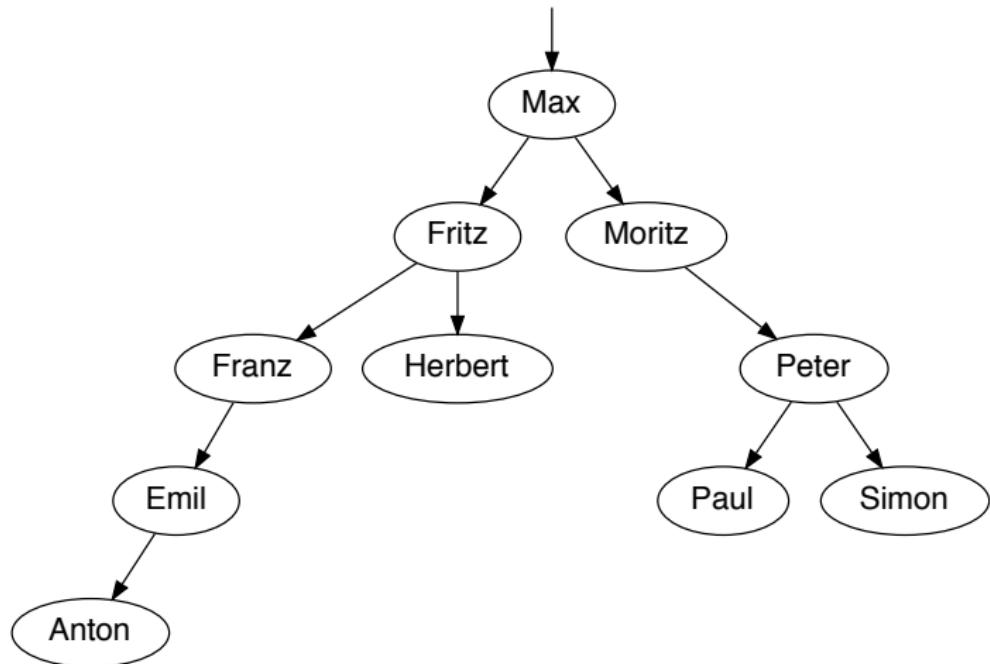


Schlüsselmenge  $K$ : Strings mit normaler alphabetischer Ordnung

# Suche im Suchbaum

- ▶ Gegeben: Baum  $B$  mit Wurzel  $W$ , Schlüssel  $k$
- ▶ Algorithmus: Suche  $k$  in  $B$ 
  - ▶ Wenn  $B$  leer ist: Ende,  $k$  ist nicht in  $B$
  - ▶ Wenn  $W.\text{key} > k$ : Suche im linken Teilbaum von  $B$
  - ▶ Wenn  $W.\text{key} < k$ : Suche im rechten Teilbaum von  $B$
  - ▶ Sonst:  $W.\text{key} = k$ : Ende, gefunden

# Binärer Suchbaum – Suche



- Wie finde ich raus, ob „Kurt“ im Baum ist?
- Wie finde ich raus, ob „Emil“ im Baum ist?

# Übung: Komplexität der Suche

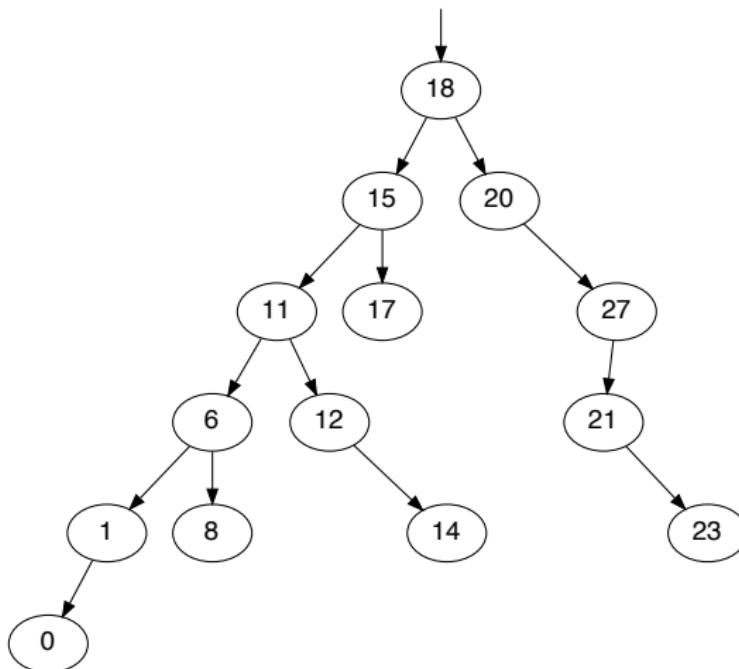
- ▶ Wie ist die Zeitkomplexität der Suchoperation in einem Baum mit  $n$  Knoten...
  - ▶ ...im schlimmsten Fall?
  - ▶ ...im besten Fall?

# Einfügen

- ▶ Gegeben: Baum  $B$  mit Wurzel  $W$ , Schlüssel  $k$
- ▶ Gesucht: Baum  $B'$ , der aus  $B$  entsteht, wenn  $k$  eingefügt wird
- ▶ Idee:
  - ▶ Suche nach  $k$
  - ▶ Falls  $k$  nicht in  $B$  ist, setze es an der Stelle ein, an der es gefunden worden wäre
- ▶ Implementierung z.B. funktional:
  - ▶ Wenn  $B$  leer ist, dann ist ein Baum mit einem Knoten mit Schlüssel  $k$  der gesuchte Baum
  - ▶ Ansonsten:
    - ▶ Wenn  $W.key > k$ : Ersetze den linken Teilbaum von  $B$  durch den Baum, der entsteht, wenn man  $k$  in ihn einfügt
    - ▶ Wenn  $W.key < k$ : Ersetze den rechten Teilbaum von  $B$  durch den Baum, der entsteht, wenn man  $k$  in ihn einfügt
    - ▶ Ansonsten:  $k$  ist schon im Baum

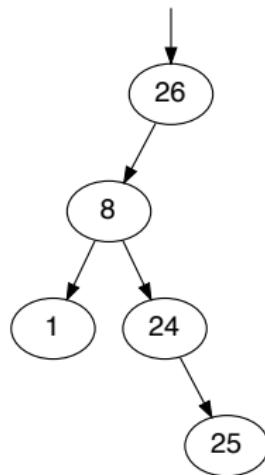
## Einfügen: Beispiel

Füge  $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$  in dieser Reihenfolge in einen Anfangs leeren Baum ein



# Übung: Einfügen

- Fügen Sie die Schlüssel  $K = (15, 0, 3, 5, 4, 27, 14, 21, 28, 6)$  in dieser Reihenfolge in den gegebenen binären Suchbaum ein



# Implementierung: Datentyp

```
class TreeNode(object):
    """
    Binary tree node
    """
    def __init__(self, key, lchild = None, rchild = None):
        self.key      = key
        self.lchild   = lchild
        self.rchild   = rchild
```

- ▶ Der leere Baum wird durch `None` repräsentiert
- ▶ In C:
  - ▶ struct mit `key` und Pointern `lchild, rchild`
  - ▶ Der leere Baum ist `NULL`

# Implementierung: Suchen

```
def find(tree , key):
    if tree:
        if key < tree.key:
            return find(tree.lchild , key)
        if key > tree.key:
            return find(tree.rchild , key)
        return tree
    else:
        return None
```

- Rückgabe: Knoten mit dem gesuchten Schlüssel oder None

# Implementierung: Einfügen

```
def insert(tree, key):
    if not tree:
        return TreeNode(key)
    else:
        if key < tree.key:
            tree.lchild = insert(tree.lchild, key)
        elif key > tree.key:
            tree.rchild = insert(tree.rchild, key)
        else:
            print "Error: Duplicate key"
    return tree
```

- Funktionaler Ansatz: Die Funktion gibt den neuen Baum zurück

# Geordnete Ausgabe

- ▶ Aufgabe: Alle Schlüssel in der geordneten Reihefolge ausgeben
  - ▶ Analog: Jede Operation, die über alle Schlüssel geordnet iteriert
- ▶ Idee:
  - ▶ Gib den linken Teilbaum (rekursiv) aus
  - ▶ Gib den Schlüssel der Wurzel aus
  - ▶ Gib den rechten Teilbaum (rekursiv) aus
- ▶ Was ist die Abbruchbedingung der Rekursion?

# Übung: Durchlaufen

- Wie ist die Komplexität der *geordneten Ausgabe*?
  - ... im *best case*
  - ... im *worst case*

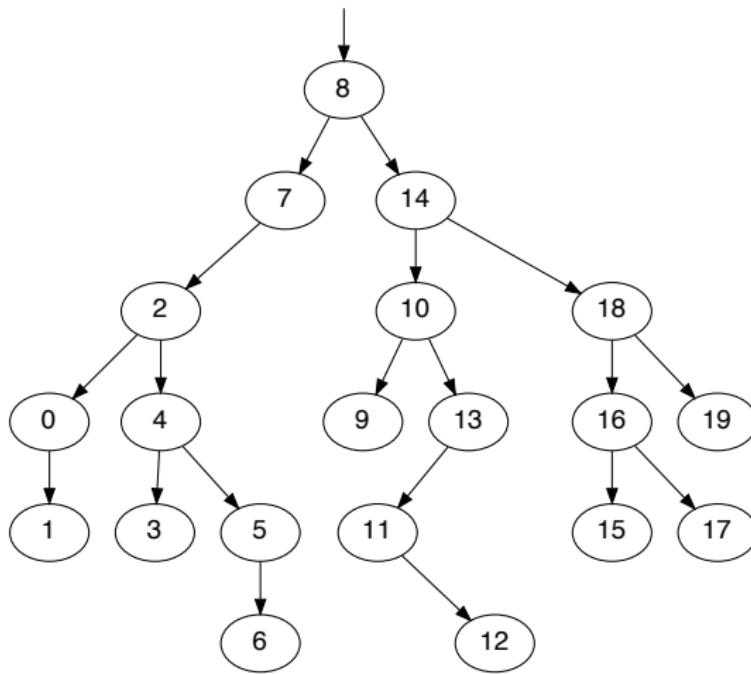
# Diskussion: Löschen

- ▶ Frage: Wie können wir einen Knoten aus dem Baum löschen?
- ▶ Diskussionsgrundlage:
  - ▶ Fall 1: Knoten ist ein Blatt
  - ▶ Fall 2: Knoten hat einen Nachfolger
  - ▶ Fall 3: Knoten hat zwei Nachfolger
- ▶ Was ist die Komplexität einer Löschoperation?

# Löschen in binären Suchbäumen (1)

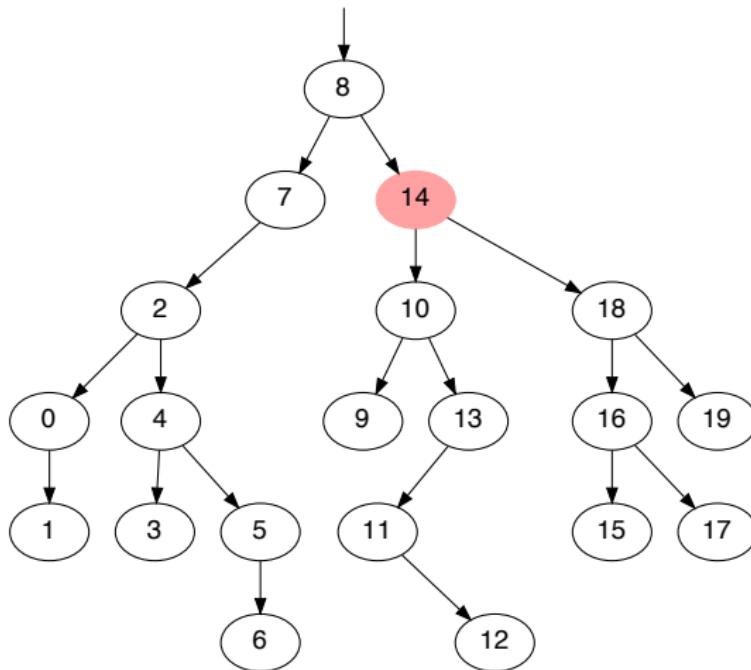
- ▶ Problem: Entferne einen Knoten  $K$  mit gegebenen Schlüssel  $k$  aus dem Suchbaum
  - ▶ ... und erhalte die Binärbaumeigenschaft
  - ▶ ... und erhalte die Suchbaumeigenschaft
- ▶ Fallunterscheidung:
  - ▶ Fall 1: Knoten hat **keinen** Nachfolger
    - ▶ Lösung: Schneide Knoten ab
    - ▶ Korrektheit: Offensichtlich
  - ▶ Fall 2: Knoten hat **einen** Nachfolger
    - ▶ Lösung: Ersetze Knoten durch seinen einzigen Nachfolger
    - ▶ Korrektheit: Alle Knoten in diesem Baum sind größer (bzw. kleiner) als die Knoten im Vorgänger des gelöschten Knotens
  - ▶ Fall 3: Knoten hat **zwei** Nachfolger
    - ▶ Lösung?

# Löschen: Beispiel



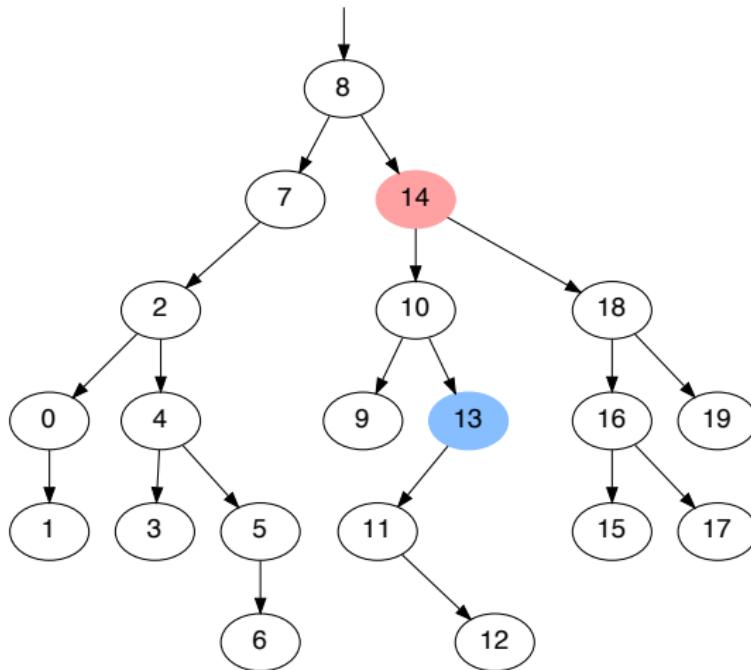
Aufgabe: Lösche Knoten 14

# Löschen: Beispiel



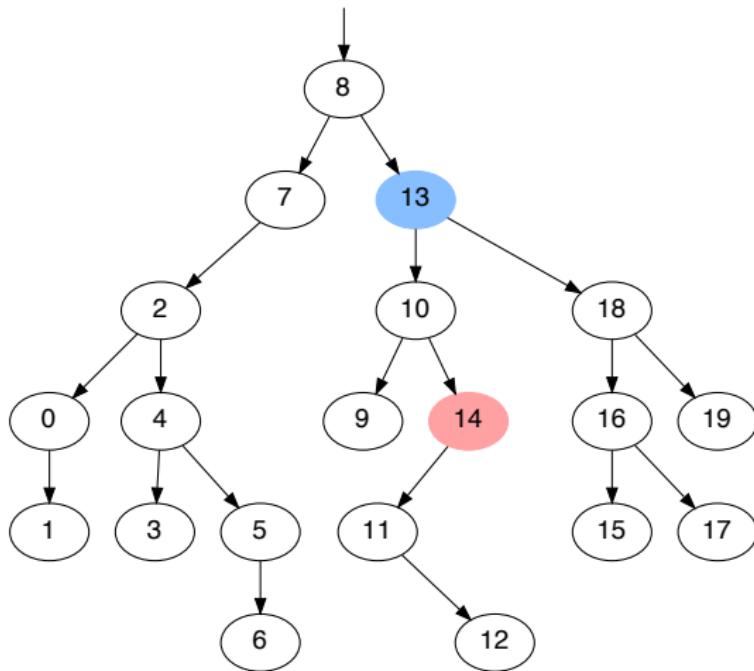
Lokalisiere Knoten 14

# Löschen: Beispiel



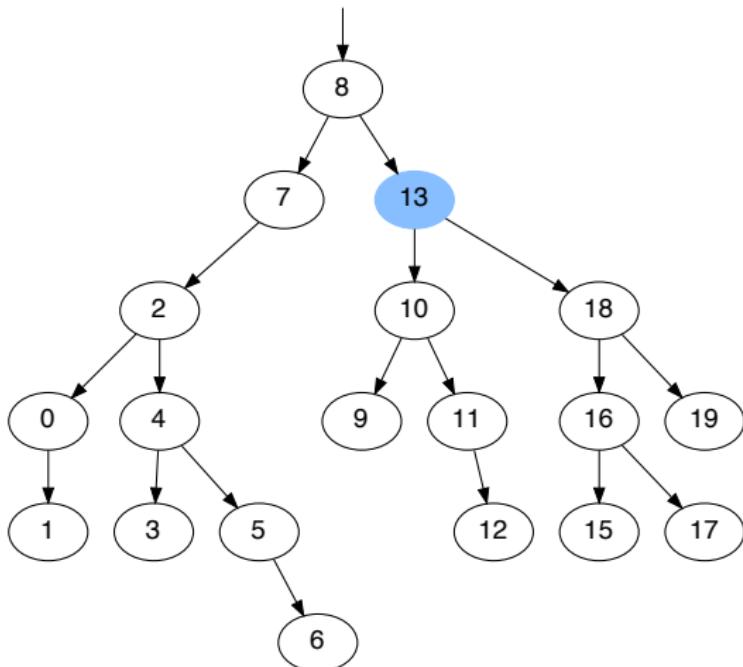
Suche Ersatzknoten (größter Knoten im linken (kleinen) Teilbaum)

# Löschen: Beispiel



Tausche 14 und Ersatzknoten 13

# Löschen: Beispiel



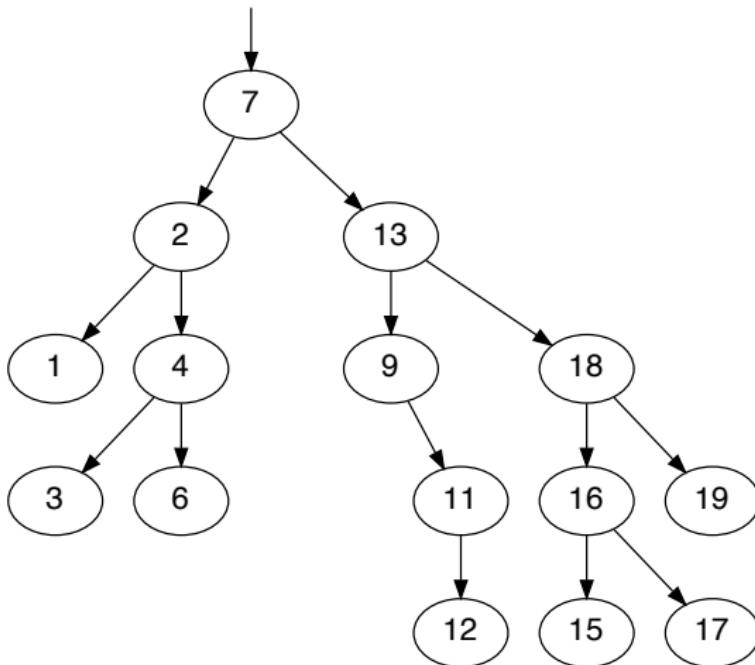
Rekursives Löschen von Knoten 14 im linken Teilbaum von 13

## Löschen in binären Suchbäumen (2)

- ▶ Problem: Entferne einen Knoten  $K$  mit gegebenen Schlüssel  $k$  aus dem Suchbaum
- ▶ Fall 3: Knoten  $K$  hat **zwei** Nachfolger
  - ▶ Lösung:
    - ▶ Suche größten Knoten  $G$  im linken Teilbaum
    - ▶ Tausche  $G$  und  $K$  (oder einfacher: Ihre Schlüssel/Werte)
    - ▶ Lösche rekursiv  $k$  im linken Teilbaum von (nun)  $G$
  - ▶ Anmerkungen
    - ▶ Wie viele Nachfolger hat  $G$  ursprünglich?
    - ▶ Also: Fall 3 kommt höchstens ein Mal pro Löschvorgang vor
    - ▶ Komplexität:  $\mathcal{O}(\log n)$  (wir folgen nur einem Ast)

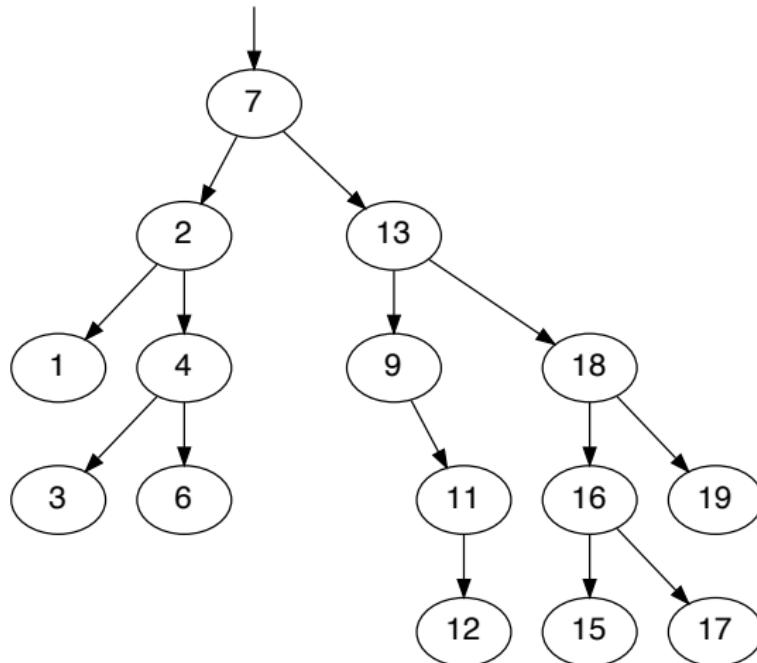
# Löschen: Beispiel

Wir löschen: ( 14 , 10 , 0 , 5 , 8 )



# Übung: Löschen

Löschen Sie die Knoten (4, 7, 12, 6, 11) in dieser Reihenfolge



# Löschen: Implementierung (1)

Finde größten Knoten in einem Baum und gib ihn zurück

```
def find_max(tree):  
    while tree.rchild:  
        tree = tree.rchild  
return tree
```

## Löschen: Implementierung (2)

```
def delete(tree, key):
    if not tree:
        print "Error: Key does not exist"
    if key < tree.key:
        tree.lchild = delete(tree.lchild, key)
    elif key > tree.key:
        tree.rchild = delete(tree.rchild, key)
    else:
        if tree.lchild and tree.rchild:
            max_node = find_max(tree.lchild)
            tmp = max_node.key
            max_node.key = tree.key
            tree.key = tmp
            tree.lchild = delete(tree.lchild, key)
        elif tree.lchild:
            return tree.lchild
        elif tree.rchild:
            return tree.rchild
        else:
            return None
    return tree
```

# Binärer Suchbaum: Scorecard

Voraussetzung:

- ▶ Baum hat  $n$  Knoten
- ▶ Kostenfunktion betrachtet Durchschnittsfall

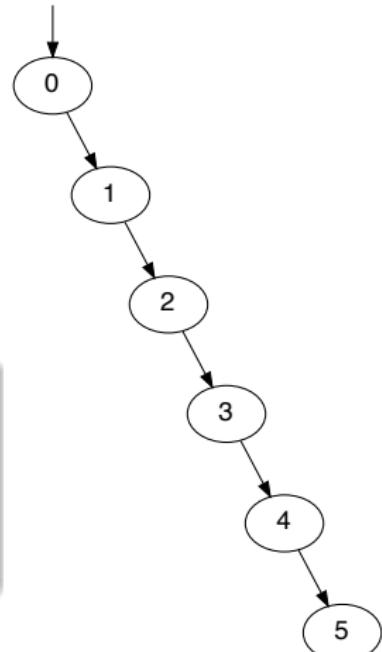
Operation	Kosten
Leeren Baum anlegen	$\mathcal{O}(1)$
Schlüssel finden	$\mathcal{O}(\log n)$
Schlüssel einfügen	$\mathcal{O}(\log n)$
Schlüssel bedingt einfügen	$\mathcal{O}(\log n)$
Schlüssel löschen	$\mathcal{O}(\log n)$
Sortierte Ausgabe:	$\mathcal{O}(n)$

Ende Vorlesung 16

# Unbalancierte Bäume

- ▶ Problem: Binäräbäume können entarten
  - ▶ Z.B. durch Einfügen einer sortierten Schlüsselmenge
  - ▶ Z.B. durch systematisches Ausfügen von kleinen/großen Schlüsseln
  - ▶ Worst case: Baum wird zur Liste
    - ▶ Viele Operationen kosten dann  $\mathcal{O}(n)$

- ▶ Lösung: Rebalancieren von Bäumen!
  - ▶ Relativ billig ( $\mathcal{O}(1)/\mathcal{O}(\log n)$  pro Knoten)
  - ▶ Kann gutartigen Baum garantieren!



# Balancierte Bäume

- ▶ Idee: An **jedem Knoten** sollen rechter und linker Teilbaum ungefähr gleich groß sein
  - ▶ Divide-and-Conquer funktioniert nahezu optimal
  - ▶ „Alles ist  $\mathcal{O}(\log n)$ “ z.B. mit dem Master-Theorem
- ▶ Größenbalancierter Binärbaum
  - ▶ Beide Teilbäume haben in etwa ähnlich viele Knoten
  - ▶ Optimal für Divide-and-Conquer
- ▶ Höhenbalancierter Binärbaum
  - ▶ Beide Teilbäume haben in etwa die gleiche Höhe
  - ▶ Gut genug für  $\log n$
  - ▶ Einfacher zu erreichen
- ▶ (Zugriffs-)Gewichtbalancierter Binärbaum
  - ▶ Die Wahrscheinlichkeit, in einen der beiden Bäume absteigen zu müssen, ist etwa gleich groß
  - ▶ Hängt von der zu erwartenden Verteilung der Schlüssel ab
  - ▶ Zu kompliziert für heute - bei Interesse Stichwort *Splay Trees*

# Diskussion

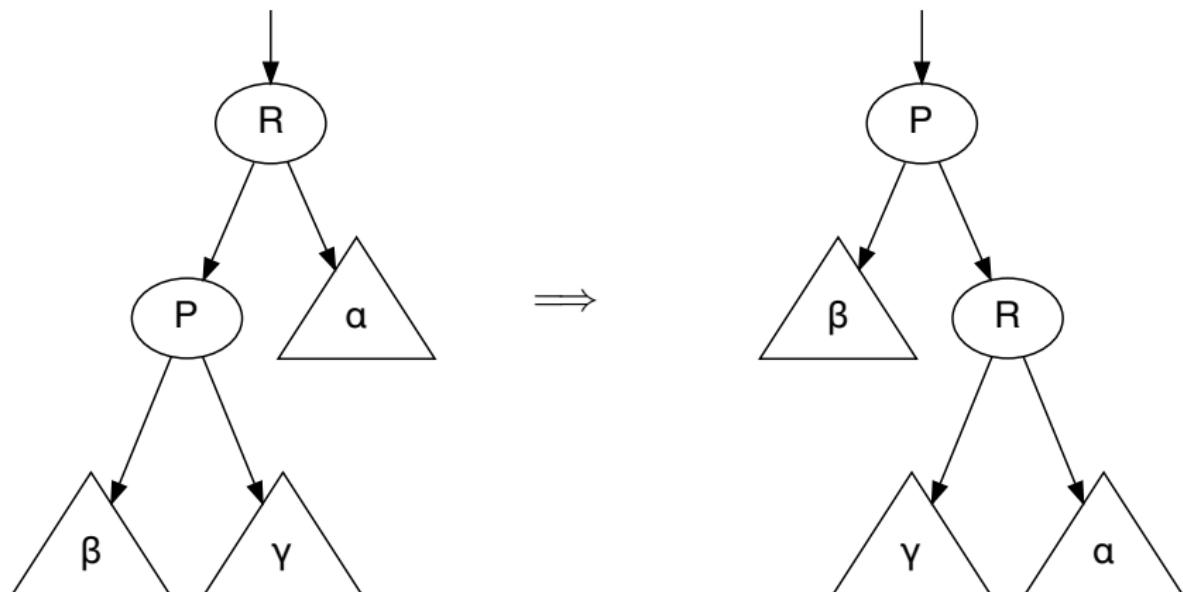
- ▶ Wie komme ich von einem unbalancierten Baum zu einem balancierten Baum?
- ▶ ... unter Erhalt der Suchbaumeigenschaft
- ▶ ... billig

# Rotationen

- ▶ Rotationen machen einen Nachbarknoten der Wurzel eines Teilbaums zur neuen Wurzel dieses Teilbaums
  - ▶ Dabei: Suchbaumkriterium bleibt erhalten
  - ▶ Höhe der Teilbäume ändert sich um 1
  - ▶ Terminologie: Der zur Wurzel beförderte Knoten heißt **Pivot**
- ▶ Rechts-Rotation:
  - ▶ Der linke Nachfolger der Wurzel ist der Pivot und wird neue Wurzel
  - ▶ Der rechte Nachfolger des Pivot wird linker Nachfolger der Wurzel
  - ▶ Die alte Wurzel wird rechter Nachfolger des Pivot
- ▶ Links-Rotation:
  - ▶ Alles andersrum

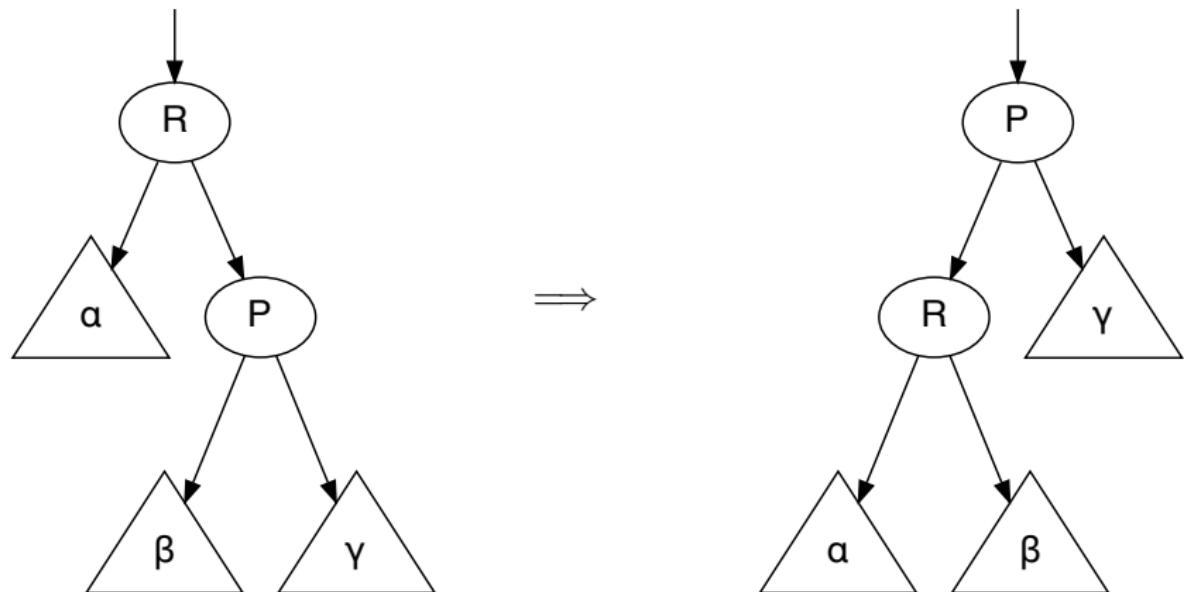
# Rechts-Rotation

Anmerkung:  $\alpha, \beta, \gamma$  sind beliebige Teilbäume



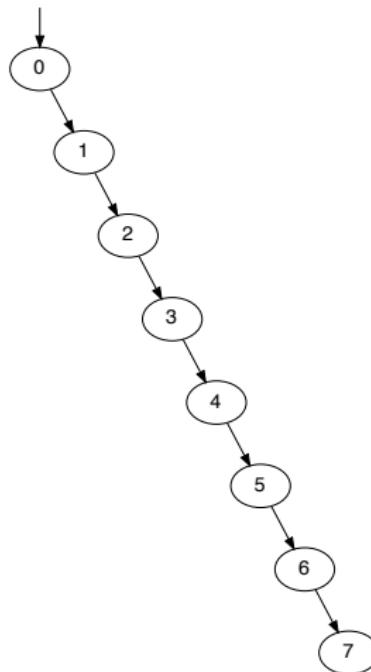
# Links-Rotation

Anmerkung:  $\alpha, \beta, \gamma$  sind beliebige Teilbäume



# Übung: Rebalancieren

Rebalancieren Sie den folgenden Baum nur mit Rechts- und Linksrotationen. Wie gut werden Sie? Wie viele Rotationen brauchen Sie?



# Rotationen Implementiert

```
def rotate_r(tree):
    pivot = tree.lchild
    tree.lchild = pivot.rchild
    pivot.rchild = tree
    return pivot

def rotate_l(tree):
    pivot = tree.rchild
    tree.rchild = pivot.lchild
    pivot.lchild = tree
    return pivot
```

# AVL-Bäume



- ▶ Georgy **Adelson-Velsky** and E. M. **Landis**: „An algorithm for the organization of information“ (1962)
- ▶ Automatisch balancierende binäre Suchbäume
  - ▶ Höhenbalanciert
  - ▶ Maximaler erlaubter Höhenunterschied für die Kinder einer Wurzel:  $+/- 1$
  - ▶ Höhe ist **garantiert** logarithmisch in Anzahl der Knoten
- ▶ Höhenunterschied wird in jedem Knoten gespeichert
- ▶ Anpassen bei Ein- oder Ausfügen
- ▶ Wird der Unterschied größer als 1: Rebalancieren mit Rotationen
  - ▶ Einfügen: Maximal zwei Rotationen notwendig ( $\sim \mathcal{O}(1)$ )
  - ▶ Löschen: Möglicherweise Rotationen bis zur Wurzel ( $\sim \mathcal{O}(\log n)$ )

# Definition AVL-Baum

## AVL-Baum

Sei  $B$  ein binärer Suchbaum, bei der jeder Knoten  $k$  neben dem Schlüssel auch mit der **Balance**  $b(k)$  beschriftet ist, wobei  $b$  wie folgt definiert ist:

$$b(k) = \text{height}(k.\text{rchild}) - \text{height}(k.\text{lchild})$$

$B$  heißt **AVL-Baum**, wenn für jeden Knoten  $k$  gilt:  $b(k) \in \{-1, 0, 1\}$ .

Beispiel:

- ▶ 0  $\rightsquigarrow$  beide Teilbäume gleich hoch
- ▶ -2  $\rightsquigarrow$  linker Teilbaum 2 Stufen höher (damit kein AVL-Baum)

Die Höhe eines AVL-Baums mit  $n$  Knoten ist **immer**  $\mathcal{O}(\log n)$ .

# Operationen auf AVL-Bäumen

Suchen wie einfacher Binärbaum

Ausgeben wie einfacher Binärbaum

Einfügen kann AVL-Eigenschaft verletzen

Löschen kann AVL-Eigenschaft verletzen

Einfügen und Löschen erfordern **Rebalancierung** des Baums

# Einfügen in AVL-Bäume

- 1 füge Knoten wie in gewöhnlichen Binärbaum ein
  - 2 durchlaufe Baum vom neuen Knoten bis zur Wurzel, passe Balance an
  - 3 wenn im Knoten  $k$  der Höhenunterschied 2 (oder -2) ist:  
füre Links- (oder Rechts-) Rotation durch
    - 1 wenn das Pivot  $p$  ein **anderes** Vorzeichen hat als die Wurzel:  
**Doppelrotation**
      - 1 **Rechts-** Rotation mit  $p$  als Wurzel
      - 2 **Links-** Rotation mit  $k$  als Wurzel
    - 2 sonst (gleiches Vorzeichen):  
**Einzelrotation**
      - 1 **Links-** Rotation mit  $k$  als Wurzel
- Balanceanpassung nur auf dem aktuellen Pfad (bottom-up)
  - Maximal **eine** Doppelrotation notwendig
  - Komplexität:  
 $\mathcal{O}(\log n)$  (Finden) +  $\mathcal{O}(1)$  (Rotieren)  $\sim \mathcal{O}(\log n)$

# Übung: Einfügen in AVL-Bäume

Fügen Sie in einen anfangs leeren AVL-Baum der Reihe nach die Zahlen 1 bis 7 und danach 9 und 8 ein.

<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

Lösung

# AVL-Bäume: Code

```
def avlrotate_rl(tree):
    tree.rchild, delta1 = avlrotate_r(tree.rchild)
    tree, delta2 = avlrotate_l(tree)
    return tree, delta1+delta2
```

```
def avlrotate_lr(tree):
    tree.lchild, delta1 = avlrotate_l(tree.lchild)
    tree, delta2 = avlrotate_r(tree)
    return tree, delta1+delta2
```

```
rtable = {
    (-1, 0) : ( 0,  1,  0),
    (-1,-1) : ( 1,  1,  0),
    (-1, 1) : ( 0,  2,  0),
    (-2,-1) : ( 0,  0, -1),
    (-2,-2) : ( 1,  0, -1)}
```

```
def avlrotate_r(tree):
    pivot = tree.lchild
    tbal = tree.balance
    pbal = pivot.balance
    tree.lchild = pivot.rchild
    pivot.rchild = tree
    tree.balance, pivot.balance, delta = rtable[(tbal, pbal)]
    return pivot, delta
```

```
ltable = {
    ( 1, 0) : ( 0, -1,  0),
    ( 1, 1) : (-1, -1,  0),
    ( 1,-1) : ( 0, -2,  0),
    ( 2, 1) : ( 0,  0, -1),
    ( 2, 2) : (-1,  0, -1)}
```

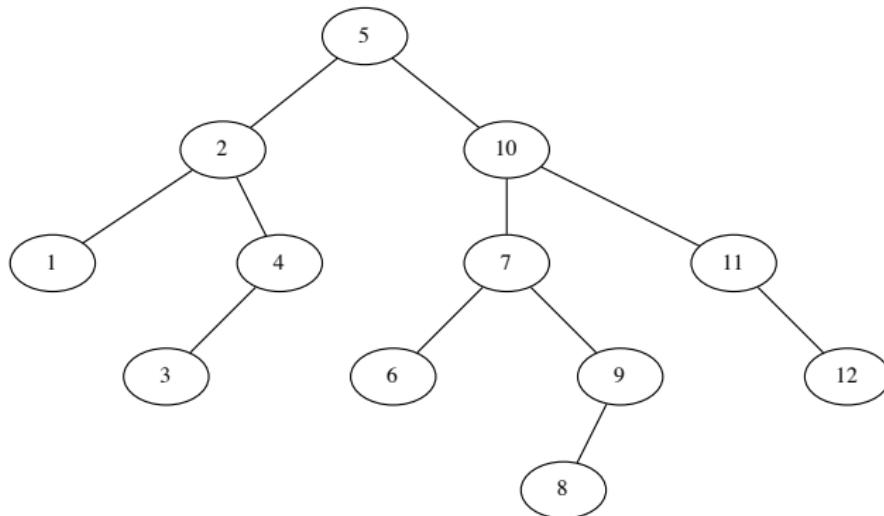
```
def avl_insert(tree, key):
    if not tree:
        return AVLNode(key), 1
    elif key < tree.key:
        tree.lchild, delta = avl_insert(tree.lchild,
                                         key)
        if delta:
            tree.balance = tree.balance-delta
        if tree.balance == -2:
            print "Rebalance left", tree.key
            if tree.lchild.balance <= 0:
                print "left-left"
                tree, delta = avlrotate_r(tree)
            else:
                print "left-right", tree.key
                tree, delta = avlrotate_lr(tree)
            # We know from theoretical analysis
            # does not grow here
            return tree, 0
        else:
            # branch has possibly grown deeper
            return tree, abs(tree.balance)
    else:
        return tree, 0
    if tree.key > key:
        tree.rchild, delta = avl_insert(tree.rchild,
                                         key)
        if delta:
            tree.balance = tree.balance+delta
        if tree.balance == 2:
            print "Rebalance right", tree.key
            if tree.rchild.balance >= 0:
                print "right-right", tree.key
                tree, delta = avlrotate_lr(tree)
            else:
```

# Löschen aus AVL-Bäumen

- 1 lösche Knoten wie aus gewöhnlichem Binärbaum
  - 2 durchlaufe Baum vom gelöschten Knoten zur Wurzel
  - 3 wenn der Höhenunterschied 2 (oder -2) ist:  
füre Links- (oder Rechts-) Rotation durch
    - 1 wenn das Pivot ein **anderes** Vorzeichen hat als die Wurzel:  
**Doppelrotation**
    - 2 sonst (gleiches Vorzeichen oder 0):  
**Einzelrotation**
- ▶ möglicherweise **mehrere** Rotationen notwendig
  - ▶ Komplexität:  
 $\mathcal{O}(\log n)$  (Löschen) +  $\mathcal{O}(\log n) \cdot \mathcal{O}(1)$  (Rotieren)  $\sim \mathcal{O}(\log n)$

# Übung: Löschen aus AVL-Bäumen

Gegeben sei der AVL-Baum  $B$ :



Löschen Sie aus  $B$  nacheinander die Knoten 1 und 11 (und stellen Sie jeweils die AVL-Eigenschaft wieder her).

Lösung

# AVL-Bäume: Zusammenfassung

- ▶ optimierte binäre Suchbäume
  - ▶ Höhe **immer**  $\mathcal{O}(\log n)$
- ▶ nach Einfügen und Löschen: ggf. **Rebalancierung**
  - ▶  $\mathcal{O}(\log n)$
  - ▶ keine Verschlechterung gegenüber gewöhnlichen binären Suchbäumen
- ▶ Implementierung komplex
  - ▶ viele Fallunterscheidungen

Ende Vorlesung 17

# Hashing: Motivation

Betrachtete Datenstrukturen:

- |                              |                           |                                |
|------------------------------|---------------------------|--------------------------------|
| ► Sortiertes Array           | ► Unsortierte Liste       | ► AVL-Bäume                    |
| Einfügen $\mathcal{O}(n)$    | Einfügen $\mathcal{O}(1)$ | Einfügen $\mathcal{O}(\log n)$ |
| Löschen $\mathcal{O}(n)$     | Löschen $\mathcal{O}(n)$  | Löschen $\mathcal{O}(\log n)$  |
| Suchen $\mathcal{O}(\log n)$ | Suchen $\mathcal{O}(n)$   | Suchen $\mathcal{O}(\log n)$   |
- Kompromiss zwischen Effizienzwerten für verschiedene Operationen
  - Ideale Zeitkomplexität: Array für alle **möglichen** Schlüsselwerte (d.h. Schlüssel wird direkt als numerischer Array-Index interpretiert)
    - Einfügen  $\mathcal{O}(1)$
    - Löschen  $\mathcal{O}(1)$
    - Suchen  $\mathcal{O}(1)$
    - Platzkomplexität inakzeptabel
    - 25 Studenten mit 7-stelligen Matrikelnummern: 10 Millionen Array-Einträge

# Hashing: Begriff

„kleinhacken“: Aus den Schlüsselwerten Kleinholz machen



# Hashing: Prinzip

Hash-Funktion  $h$ : Abbildung von **Schlüsselwerten** auf (dichte, endliche) Teilmenge von  $\mathbb{N}$

- ▶ Hashwert wird als Index im Array (**Hash-Tabelle**) verwendet
  - ▶ Beispiel:  $h(\text{„Peter“}) = 4$ ,  $h(\text{„Paul“}) = 3$ ,  $h(\text{„Mary“}) = 1$

Hash-Wert	Name	Geburtsdatum	Kurs
1	Mary	3. 6. 1990	AI13A
2			
3	Paul	22. 3. 1991	AI13C
4	Peter	18. 11. 1990	AI13B
5			

- ▶ Wertebereich von  $h$  muss
  - ▶ größer sein als Zahl der **genutzten** Schlüsselwerte
  - ▶ deutlich kleiner sein als Zahl der **möglichen** Schlüsselwerte
- ▶  $h$  muss einfach zu berechnen sein

# Hash-Funktionen

- ▶ transformiere Schlüssel  $s$  in Zahl  $i$ 
  - ▶ String: z.B. Summe der ASCII-Werte
  - ▶ Datum: z.B. Tage seit 1. 1. 1900
- ▶ berechne aus (möglicherweise sehr großer) Zahl  $i$  Hash-Wert  $h(i)$ 
  - ▶ z.B.  $h(i) = (i \bmod N)$  für Hash-Tabelle mit  $N$  Einträgen
- ▶ Wünschenswerte Eigenschaften:
  - ▶ Berücksichtigung aller Bits
    - ▶ Änderung von  $i$  um 1 Bit  $\leadsto$  Änderung von  $h(i)$
    - ▶ Vermeidung von gleichen Hash-Werten (**Kollisionen**)
    - ▶ Schlecht: Gleicher Hash-Wert für Schmidt, Schmid und Schmied
  - ▶ kleine Änderung von  $i$   $\leadsto$  große Änderung von  $h(i)$ 
    - ▶ Gleichmäßige Verteilung ähnlicher Schlüssel
    - ▶ Schlecht: Alle Schmidts liegen dicht beieinander
    - ▶ Vermeidung von **Clustering**

## Kollision

Eine **Kollision** tritt auf, wenn zwei verschiedene Schlüssel, die denselben Hash-Wert haben, in eine Hash-Tabelle eingefügt werden.

Kollisionen sind unvermeidbar, wenn Hashing effektiv sein soll  
(Hash-Tabelle deutlich kleiner als Schlüsselbereich)

**Chaining:** verkettete Liste in der Hash-Tabelle

- ▶ Kollision  $\leadsto$  in Liste einfügen
- ▶ Verteilen von  $n$  Schlüsseln auf  $N$  Töpfe
- ▶ Hash-Tabelle kann **kleiner** sein als Anzahl vorhandener Schlüssel
- ▶ aber: zu kleines  $N$ : lineare Suche mit durchschnittlicher Länge  $\frac{n}{N}$
- ▶ Verwendet in: Java HashMap, C Sharp Dictionary

# Kollisionsbehandlung (I)

Anderer Ansatz: Finden eines alternativen Felds für  $j$

- ▶ zusätzliche Komplikation: Markieren gelöschter Felder notwendig
- ▶ sonst kann  $j$  nach dem Löschen von  $i$  nicht mehr gefunden werden, ohne gesamte Tabelle zu durchsuchen

Annahme: Vorhandener Schlüssel  $i$ , neuer Schlüssel  $j$ ,  $h(i) = h(j)$

**Linear Probing:** In nächstes freies Feld einfügen

- ▶ Kollision an  $h(j)$   
    ~ Einfügen an Position  $((h(j) + m) \bmod N)$ , wobei  $m \in \mathbb{N}$  von 1 hochgezählt wird
- ▶ Nachteil: Begünstigt Clustering (zusammenhängende belegte Felder)
- ▶ Frage: kann Clustering durch Formel  $(h(j) + m \cdot k)$  für ein konstantes  $k$  vermieden werden?

# Kollisionsbehandlung (II)

Annahme: Vorhandener Schlüssel  $i$ , neuer Schlüssel  $j$ ,  $h(i) = h(j)$

**Re-hashing:** Zweite Hash-Funktion  $r$  bestimmt „Sprungweite“

- ▶ Berechne Rehash-Funktion  $r(j)$
- ▶ Teste Positionen  $(h(j) + m \cdot r(j)) \bmod N$
- ▶ Vermeidet Clustering
- ▶ Anforderungen an  $r$ :
  - ▶ wird nie 0 (warum?)
  - ▶ Wertebereich kleiner  $N$  (warum?)
  - ▶ Werte **teilerfremd zu  $N$** 
    - ▶ sonst werden nicht alle möglichen Felder getroffen
    - ▶ einfachste Lösung:  $N$  ist Primzahl

# Übung: Hashing

Gegeben sei eine Hash-Tabelle der Größe 11 (Feldindizes 0–10).

Verwenden Sie alle drei angegebenen Verfahren, um nacheinander die Schlüssel 1, 17, 1024, 55, 69, 122 in die Tabelle einzufügen. Zählen Sie jeweils die Kollisionen.

Verwenden Sie als Hash-Funktionen

- ▶  $h(x) = x \bmod 11$
- ▶  $r(x) = (x \bmod 4) + 1$

# Hashing: Komplexität Best und Worst Case

Einfügen / Finden eines Schlüssels  $i$  in eine Hash-Tabelle mit

- ▶  $N$  Feldern und
- ▶  $n$  vorhandenen Schlüsseln

## Best Case

- ▶ keine Kollisionen
- ▶  $i$  wird in erstem Feld gefunden / gespeichert
- ▶ 1 Berechnung von  $h$  (und  $r$ ),  
1 Lesezugriff,  
Einfügen: 1 Schreibzugriff
- ▶ Gesamt:  $\mathcal{O}(1)$

## Worst Case

- ▶ Kollisionen mit allen vorhandenen Einträgen
- ▶ 1 Berechnung von  $h$  (und  $r$ ),  
 $n$  Lesezugriffe,  
Einfügen: 1 Schreibzugriff
- ▶ Gesamt:  $\mathcal{O}(n)$

# Hashing: Komplexität Average Case

Best und Worst Case wenig aussagekräftig:

- ▶ treten in Praxis fast nie auf
- ▶ berücksichtigen nicht  $N$
- ▶ „Effizienz von Hashing zwischen direktem Zugriff und linearer Suche.“
- ▶ besser oder schlechter als  $\log n$  ( $\sim$  AVL-Baum)?

Average Case: Erwartungswert für Anzahl der notwendigen Vergleiche („Probes“)

- ▶ abhängig vom Füllstand  $\alpha = \frac{n}{N}$ 
  - ▶  $N$ : verfügbare Felder;  $n$ : besetzte Felder
- ▶ Annahme: ideale Hash-Funktion
  - ▶  $h(i)$  ist gleichverteilt über  $N$
  - ▶ jeder Wert gleich wahrscheinlich

# Komplexität im Average Case [Knuth, Sedgewick]

Unterscheidung:

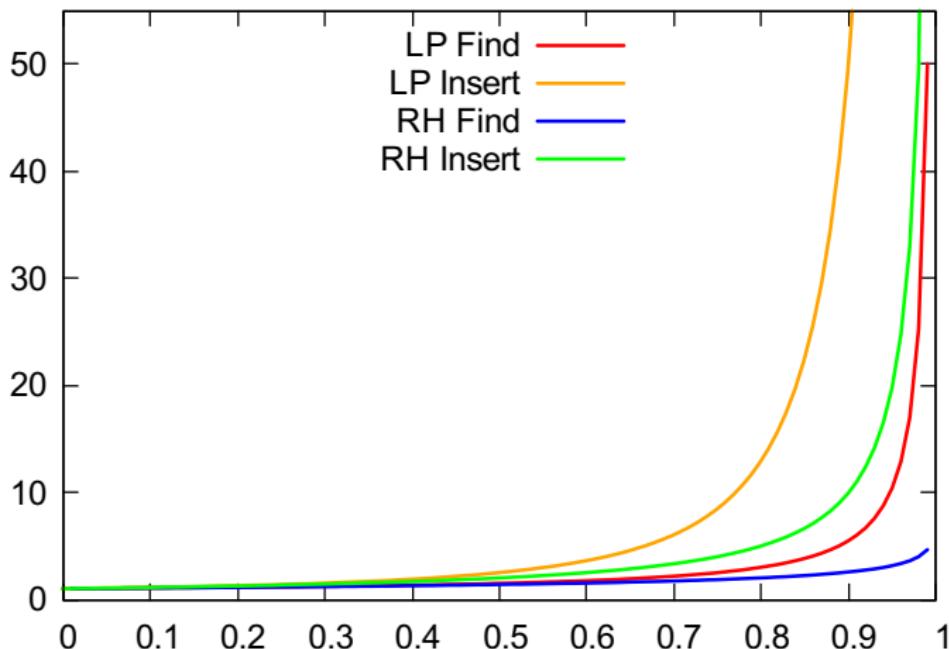
- ▶ Linear Probing  $\leftrightarrow$  Re-hashing
  - ▶ Linear Probing begünstigt Clustering
- ▶ Finden  $\leftrightarrow$  Einfügen
  - ▶ Einfügen aufwändiger, da alle Einträge mit gleichem Hash getestet werden müssen.
  - ▶ Löschen  $\sim$  Finden
  - ▶ Erfolglose Suche  $\sim$  Einfügen

Verfahren	Finden	Einfügen
Linear Probing	$\frac{1}{2} \left(1 + \frac{1}{1-\alpha}\right)$	$\frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2}\right)$
Re-hashing	$\frac{-\ln(1-\alpha)}{\alpha}$	$\frac{1}{1-\alpha}$

# Beispiele: Average Case

Verfahren	Finden	Einfügen		
Linear Probing	$\frac{1}{2} \left(1 + \frac{1}{1-\alpha}\right)$	$\frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2}\right)$		
Re-hashing	$\frac{-\ln(1-\alpha)}{\alpha}$	$\frac{1}{1-\alpha}$		
Füllstand $\alpha$	Linear Probing	Re-hashing		
	Finden	Einfügen	Finden	Einfügen
0	1	1	„0/0“	1
1 / 4	1,2	1,4	1,2	1,3
1 / 2	1,5	2,5	1,4	2
3 / 4	2,5	8,5	1,8	4
9 / 10	5,5	50,5	2,6	10

# Hash-Verfahren: Graphischer Vergleich



- Zugriffszeiten **unabhängig von  $n$**
- bei zu großem  $\alpha$ : in größere Tabelle umkopieren  $\sim \mathcal{O}(n)$

# Hashing von Strings

- ▶ Summe der ASCII-Werte
  - ▶ Anagramme haben gleichen Hash-Wert ( $h(\text{„ein“}) = h(\text{„nie“})$ )
    - ~ Kollisionen
  - ▶ ASCII-Werte beginnen bei 65 ~ schlechte Abdeckung des Wertebereichs
- ▶ gesamten String als binäre Integer interpretieren:  
„Baden-Wuerttemberg“
  - ~ 426164656E7E577565727474656D62657267<sub>hex</sub> (144 bit)
  - ~ 5782551710550143385235732306515534408938087<sub>dec</sub>
  - ▶ Integer-Operationen nur bis 64 bit effizient
  - ▶ Abhilfe: nur erste/letzte 64 bit verwenden
    - ~ nur letzte 8 Byte relevant
    - ~ alle Wörter auf „-lichkeit“ haben selben Hashwert

# Effizientes Hashing von Strings: Horner-Schema

- ▶ fasse String als Zahl zur Basis  $b$  auf  
„Baden“  $\sim 42_{\text{hex}} \cdot b^4 + 61_{\text{hex}} \cdot b^3 + 64_{\text{hex}} \cdot b^2 + 65_{\text{hex}} \cdot b^1 + 6E_{\text{hex}} \cdot b^0$
- ▶ berechne Hashwert iterativ  
$$((((42_{\text{hex}} \bmod n \cdot b + 61_{\text{hex}}) \bmod n \cdot b + 64_{\text{hex}}) \bmod n \cdot b + 65_{\text{hex}}) \bmod n \cdot b + 6E_{\text{hex}}) \bmod n$$
- ▶ ähnliche Verfahren für Zufallszahlen und Kryptographie

```
def hash(string, n):  
    h = 0  
    for i in range(len(string)):  
        h = (h * b + ord(string[i])) % n  
    return h
```

## Auswahl des Parameters $b$

- ▶ Primzahl  $\sim$  gleichmäßige Verteilung über Wertebereich
- ▶  $2^m - 1$  für  $m \in \mathbb{N} \sim$  effiziente Berechnung
- ▶ häufig verwendet: 31 (z.B. String-Hashfunktion von Java)

# Übung: Horner-Schema

Berechnen Sie die Hash-Werte für die Strings „DHBW“ und „Stuttgart“, basierend auf den ASCII-Werten der Buchstaben.

Verwenden Sie als Parameter  $b = 31$  und  $n = 101$ .

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	:	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	-	111	o	127	[backspace]

Lösung

# Zusammenfassung: Hashing

- ▶ Voraussetzung
  - ▶ Schlüsselbereich  $K$  (unendlich) groß
  - ▶ Anzahl tatsächlich verwendeter Schlüssel  $n$  klein
- ▶ Ziel: annähernd konstante Zugriffszeiten
- ▶ Methode
  - ▶ bilde Schlüsselmenge auf überschaubaren Integer-Bereich  $0 \dots N$  ab
  - ▶ verwende Array zur Speicherung  $\sim \mathcal{O}(1)$
  - ▶ bei Konflikten: Chaining oder Probing
  - ▶ bei zu hohem Füllstand: in größere Tabelle umkopieren  $\sim \mathcal{O}(n)$

# Zusammenfassung: Vor- und Nachteile

- ▶ Vorteile
  - ▶ Zugriffszeit nur abhängig von **Füllstand**, nicht Gesamtgröße
  - ▶ bei Füllstand 1/2 wie AVL-Baum mit 7 Elementen
- ▶ Nachteile
  - ▶ abhängig von guter Hash-Funktion
  - ▶ häufiges **Löschen** verschlechtert Zugriffszeiten
  - ▶ erfordert Überwachung des Füllstands und ggf. Umkopieren
  - ▶ **sortierte Ausgabe** nicht unterstützt
- ▶ besonders effizient wenn
  - ▶  $N \gg n$
  - ▶  $n$  im Voraus absehbar
  - ▶ Anzahl Lesezugriffe  $\gg$  Anzahl Schreibzugriffe
  - ▶ Löschoperationen selten oder nie
- ▶ typische Anwendungsbereiche
  - ▶ Symboltabelle im Compilerbau
  - ▶ Datenbanken (Wörterbücher, Telefonbücher, Kundendateien, ...)
  - ▶ Logik: Erfüllbarkeits-Caching für Teilformeln

## **Graphalgorithmen**

# Graphen

## Graph

Ein gerichteter Graph (Digraph) besteht aus einer Knotenmenge  $V$  und einer Kantenmenge  $E \subseteq V \times V$ .

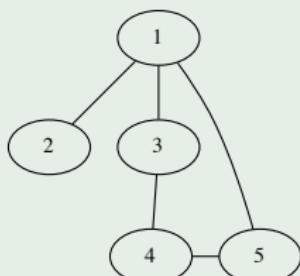
Ein ungerichteter Graph  $V$  ist ein gerichteter Graph  $(V, E)$ , bei dem die Relation  $E$  symmetrisch ist.

$$(a, b) \in E \Leftrightarrow (b, a) \in E$$

(Default: Ungerichtete Graphen)

$$V = \{1, 2, 3, 4, 5\}$$

$$E = \{(1, 2), (2, 1), (1, 3), (3, 1), (1, 5), (5, 1), (3, 4), (4, 3), (4, 5), (5, 4)\}$$



# Beschriftete Graphen

## Beschrifteter Graph

Ein **knoten-/kantenbeschrifteter Graph**  $G = (V, E)$  ist ein Graph mit Beschriftungsfunktion(en)  $v : V \rightarrow L_V$  bzw.  $e : E \rightarrow L_E$  für Mengen  $L_V, L_E$ .

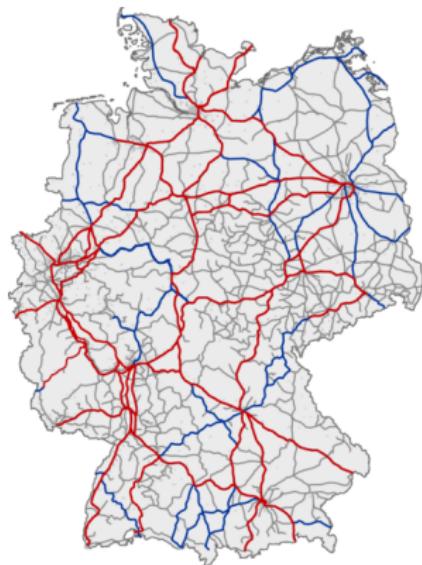
Ist  $L_V$  bzw.  $L_E$  eine Menge von Zahlen, spricht man auch von einem **gewichteten Graphen**.

## Beschriftungen

$$\begin{aligned} G &= (V, E) = (\{1, 2, 3, 4, 5\}, \{(1, 2), (2, 1), (1, 3), \dots\}) \\ L_V &= \{\text{Köln, Hamburg, Bremen, Stuttgart, Frankfurt}\} \\ L_E &= \mathbb{N} \\ v &= \{1 \mapsto \text{Köln}, 2 \mapsto \text{Hamburg}, \dots\} \\ e &= \{(1, 2) \mapsto 430, (1, 3) \mapsto 321, (3, 4) \mapsto 626, \dots\} \end{aligned}$$

# Anwendung von Graphen

- ▶ Netzwerke
  - ▶ Straßen, Wasser-, Stromversorgung
  - ▶ Computernetzwerke
  - ▶ soziale Netzwerke
- ▶ Technik
  - ▶ Schaltkreise
  - ▶ Flussdiagramme
  - ▶ Links im WWW
- ▶ Hierarchien
  - ▶ Vererbung in OO-Sprachen
  - ▶ Taxonomien



# Repräsentation von Graphen: Adjazenzmatrix

## Adjazenzmatrix

Die Adjazenzmatrix  $A$  für einen Graphen mit  $n$  Knoten hat die Dimension  $n \times n$ . Die Elemente nehmen die Werte 0 oder 1 an.  $A(x, y) = 1$  bedeutet, dass es eine Kante von  $x$  nach  $y$  gibt.

Die Adjazenzmatrix kann als zweidimensionales Array repräsentiert werden.

$$V = \{1, 2, 3, 4, 5\}$$

$$E = \{(1, 2), (2, 1), (1, 3), (3, 1), (1, 5), (5, 1), (3, 4), (4, 3), (4, 5), (5, 4)\}$$

	1	2	3	4	5
1	0	1	1	0	1
2	1	0	0	0	0
3	1	0	0	1	0
4	0	0	1	0	1
5	1	0	0	1	0

# Repräsentation von Graphen: Adjazenzlisten

## Adjazenzliste

Die Adjazenzliste  $L$  für einen Knoten  $x$  in einem Graphen  $G$  enthält alle Knoten  $y$ , zu denen es eine von  $x$  ausgehende Kante gibt.

$$V = \{1, 2, 3, 4, 5\}$$

$$\begin{aligned} E = & \{(1, 2), (2, 1), (1, 3), (3, 1), \\ & (1, 5), (5, 1), (3, 4), (4, 3), \\ & (4, 5), (5, 4)\} \end{aligned}$$

$$\begin{aligned} 1 &\mapsto (2, 3, 5) \\ 2 &\mapsto (1) \\ 3 &\mapsto (1, 4) \\ 4 &\mapsto (3, 5) \\ 5 &\mapsto (1, 4) \end{aligned}$$

Vorteil gegenüber Matrix:

- Platz  $\mathcal{O}(|E|)$  statt  $\mathcal{O}(|V|^2)$
- vor allem bei **mageren** (sparse) Graphen ( $|E| \sim |V|$ )

Nachteil gegenüber Matrix:

- Entscheidung, ob  $(x, y) \in E$  gilt, ist  $\mathcal{O}(|E|)$  statt  $\mathcal{O}(1)$
- vor allem bei **dichten** (dense) Graphen ( $|E| \sim |V|^2$ )

# Repräsentation von Graphen-Beschriftungen: Matrix

Bei beschrifteten Graphen können Knotenbeschriftungen in einem Array und Kantenbeschriftungen in der Adjazenzmatrix repräsentiert werden.

$n$	$v(n)$
1	Köln
2	Hamburg
3	Bremen
4	Stuttgart
5	Frankfurt

$e(m, n)$	1	2	3	4	5
1	$\infty$	430	321	$\infty$	190
2	430	$\infty$	$\infty$	$\infty$	$\infty$
3	321	$\infty$	$\infty$	626	$\infty$
4	$\infty$	$\infty$	626	$\infty$	205
5	190	$\infty$	$\infty$	205	$\infty$

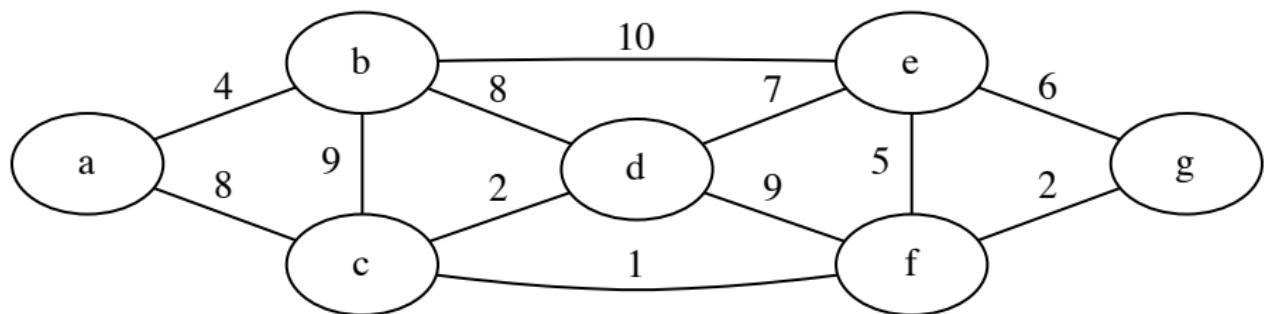
# Repräsentation von Graphen-Beschriftungen: Liste

In einer Adjazenzliste können die Kantengewichte zusammen mit der Kante gespeichert werden.

$n$	$v(n)$	
1	Köln	1 $\mapsto$ ((2, 430), (3, 321), (5, 190))
2	Hamburg	2 $\mapsto$ ((1, 430))
3	Bremen	3 $\mapsto$ ((1, 321), (4, 626))
4	Stuttgart	4 $\mapsto$ ((3, 626), (5, 205))
5	Frankfurt	5 $\mapsto$ ((1, 190), (4, 205))

# Übung: Adjazenzmatrix und -liste

Geben Sie für den Graphen die Adjazenzmatrix sowie die Adjazenzlisten an.



# Graphen: Definitionen

## Pfad, Zyklus, Baum

Für einen Graphen  $G$  ist ein **Pfad** eine Folge von Knoten  $(v_1, v_2, \dots, v_k)$ , so dass gilt:  $\forall i < k : (v_i, v_{i+1}) \in E$ .

Ein Knoten  $y$  ist von einem Knoten  $x$  **erreichbar**, wenn es einen Pfad von  $x$  nach  $y$  gibt.

Ein Graph  $G = (V, E)$  heißt **verbunden**, wenn jeder Knoten in  $V$  von jedem anderen Knoten erreichbar ist.

Ein **Zyklus** ist ein (nichtleerer) Pfad mit  $v_1 = v_k$ .

(Bei ungerichteten Graphen: Jede Kante darf nur in einer Richtung benutzt werden.)

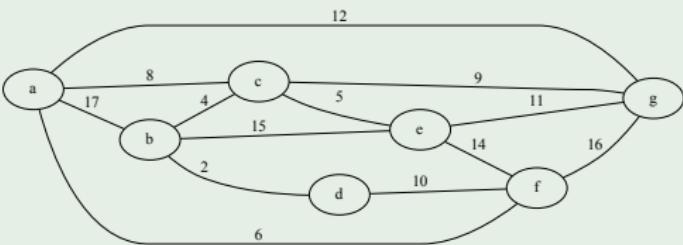
Ein **Baum** ist ein verbundener azyklischer Graph.

# Problemstellungen für Graphen

- ▶ Minimaler Spannbaum
  - ▶ Gegeben: Ungerichteter verbundener kantengewichteter Graph  $G$
  - ▶ Gesucht: verbundener Teilgraph  $G_{min}$  von  $G$  mit minimaler Summe der Kantengewichte
  - ▶ Beobachtung:  $G_{min}$  muss ein Baum sein
  - ▶ Anwendung: Versorgungsnetze
- ▶ Kürzeste Pfade
  - ▶ Gegeben: Verbundener kantengewichteter Graph  $G$
  - ▶ Gesucht: Kürzeste Pfade von  $x$  nach  $y$  für alle Knoten  $x, y$
  - ▶ Anwendung: Routing

# Minimaler Spannbaum

	a	b	c	d	e	f	g
a	$\infty$	17	8	$\infty$	$\infty$	6	12
b		$\infty$	4	2	15	$\infty$	$\infty$
c			$\infty$	$\infty$	5	$\infty$	9
d				$\infty$	$\infty$	10	$\infty$
e					$\infty$	14	11
f						$\infty$	16
g							$\infty$



Übung: Versuchen Sie, einen minimalen Spannbaum zu finden.

# Prim-Algorithmus

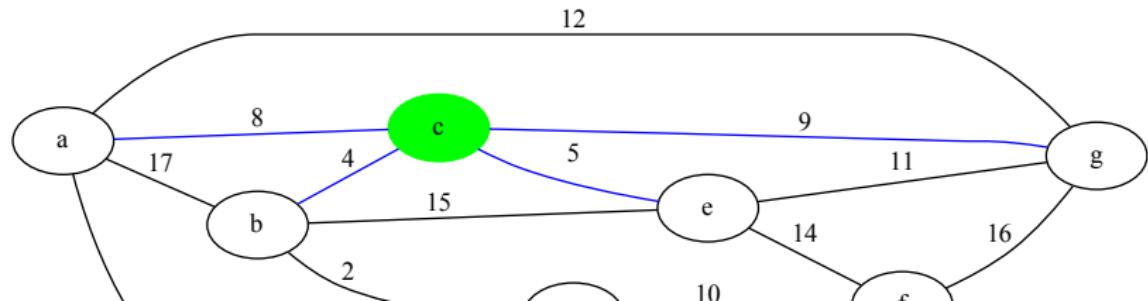
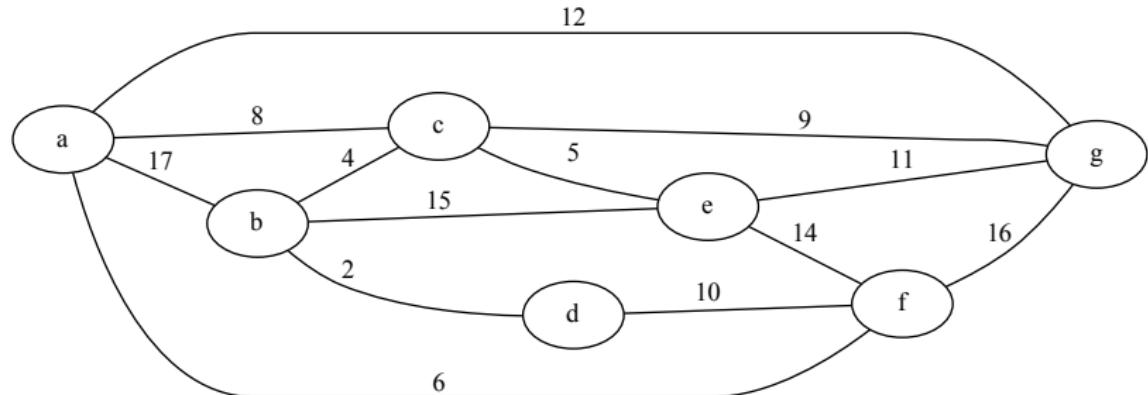
Erstentwicklung: Vojtěch Jarník, tschechischer Mathematiker, 1930,  
wiederentdeckt 1957 von Robert C. Prim, amerikanischer  
Mathematiker, 1959 von Edsger W. Dijkstra, holländischer Informatiker

Eingabe: Graph  $G = (V, E)$

Ausgabe: MST  $T = (V, E_T)$

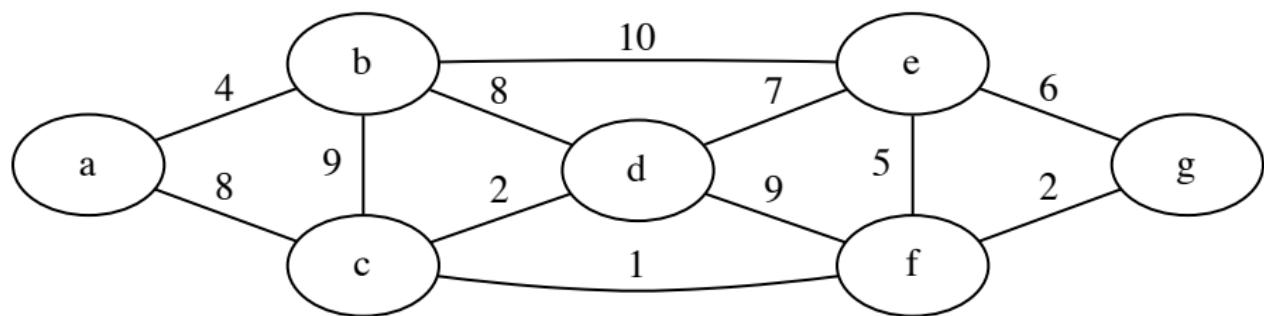
- 1  $E_T = \emptyset$
- 2 wähle  $v_{start}$ ;  $V_b = \{v_{start}\}$ ;  $V_n = V \setminus \{v_{start}\}$
- 3 solange  $V_n$  Knoten enthält
  - 1  $e_n = (v_b, v_n)$  sei billigste Kante zwischen Knoten aus  $V_b$  und  $V_n$
  - 2 nimm  $e_n$  zu  $E_T$  hinzu
  - 3 entferne  $v_n$  aus  $V_n$
  - 4 nimm  $v_n$  zu  $V_b$  hinzu
- 4 gib  $(V, E_T)$  zurück

# Beispiel: Prim-Algorithmus



# Übung: Prim-Algorithmus

Bestimmen Sie einen minimalen Spannbaum für folgenden Graphen.  
Beginnen Sie mit einem zufälligen Knoten. Falls Sie genug Zeit haben,  
wiederholen Sie die Übung mit einen zweiten Startknoten. Was  
können Sie beobachten?



Zusatzaufgabe

# Gruppenübung: Komplexität des Prim-Algorithmus

Welche Komplexität hat der naive Prim-Algorithmus?

Eingabe: Graph  $G = (V, E)$

Ausgabe: MST  $T = (V, E_T)$

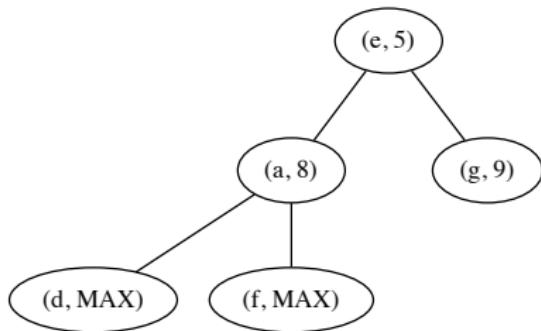
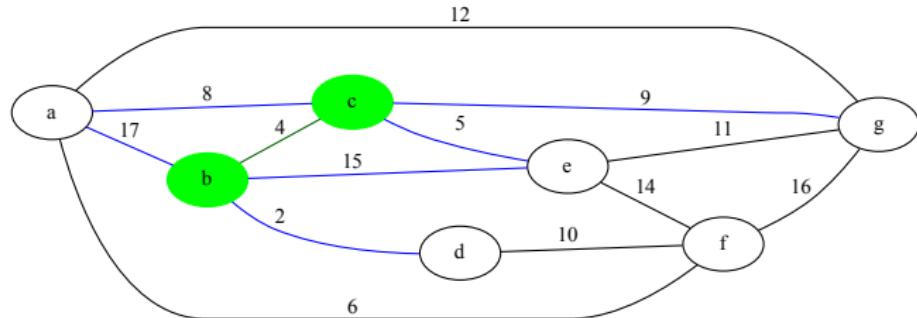
- 1  $E_T = \emptyset$
- 2 wähle  $v_{start}$ ;  $V_b = \{v_{start}\}$ ;  $V_n = V \setminus \{v_{start}\}$
- 3 solange  $V_n$  Knoten enthält
  - 1  $e_n = (v_b, v_n)$  sei billigste Kante zwischen Knoten aus  $V_b$  und  $V_n$
  - 2 nimm  $e_n$  zu  $E_T$  hinzu
  - 3 entferne  $v_n$  aus  $V_n$
  - 4 nimm  $v_n$  zu  $V_b$  hinzu
- 4 gib  $(V, E_T)$  zurück

Ende Vorlesung 19

# Optimierung des Prim-Algorithmus: Idee

- ▶  $E$  wird durch **Adjazenzliste** repräsentiert
- ▶ organisiere  $V_n$  als **Min-Heap**
  - ▶ Elemente: Knoten
  - ▶ Gewicht: Kosten der billigsten Kante zu einem Knoten aus  $V_b$
- ▶ entferne der Reihe nach Knoten mit minimalem Gewicht
  - ▶ anschließend: Bubble-down der neuen Wurzel
- ▶ nach jedem neuen Knoten  $v_n$ : Update der Gewichte der mit  $v_n$  direkt verbundenen Knoten
  - ▶ ggf. bubble-up der Knoten

# Beispiel: Prim-Algorithmus mit Heap



# Komplexität des optimierten Prim-Algorithmus

- ▶ für jeden **Knoten** einmal bubble-down
  - ▶  $\mathcal{O}(|V| \log |V|)$
- ▶ für jede **Kante** einmal bubble-up
  - ▶  $\mathcal{O}(|E| \log |V|)$
  - ▶ jede Kante wird nur einmal betrachtet
  - ▶ Kanten möglicherweise ungleich verteilt über Knoten
  - ▶ wichtig: Adjazenz**liste**, bei Matrix:  $|V|^2 \log |V|$
- ▶ Damit:  $\mathcal{O}(|V| \log |V| + |E| \log |V|)$
- ▶ verbundener Graph  $\leadsto |V| \leq |E| + 1$ 
  - ▶ Gesamtkomplexität:  $\mathcal{O}(|E| \log |V|)$  statt  $\mathcal{O}(|E| \cdot |V|)$

# Routing: Der Dijkstra-Algorithmus

- ▶ Problem: Finde (garantiert) den kürzesten/günstigsten Weg von A nach B
- ▶ Anwendungen:
  - ▶ Luftverkehr
  - ▶ Straßenverkehr
  - ▶ Routing von Paketen (UPS, DHL)
  - ▶ Routing von Paketen (Cisco, DE-CIX)
  - ▶ ...
- ▶ Grundlage: Gewichteter Graph ( $V, E$ )
  - ▶  $V$  sind die besuchbaren/passierbaren Orte
  - ▶  $E$  sind die Verbindungen
  - ▶  $e : E \rightarrow \mathbb{N}$  sind die Kosten einer Einzelverbindung
- ▶ Besonderheit: Wir finden den kürzesten Weg von A zu jedem anderen Ziel (das ist im Worst-Case nicht schwieriger)



Edsger W. Dijkstra  
(1930-2002)  
Turing-Award 1972  
*„Goto Considered Harmful“*, 1968

# Dijkstra: Idee

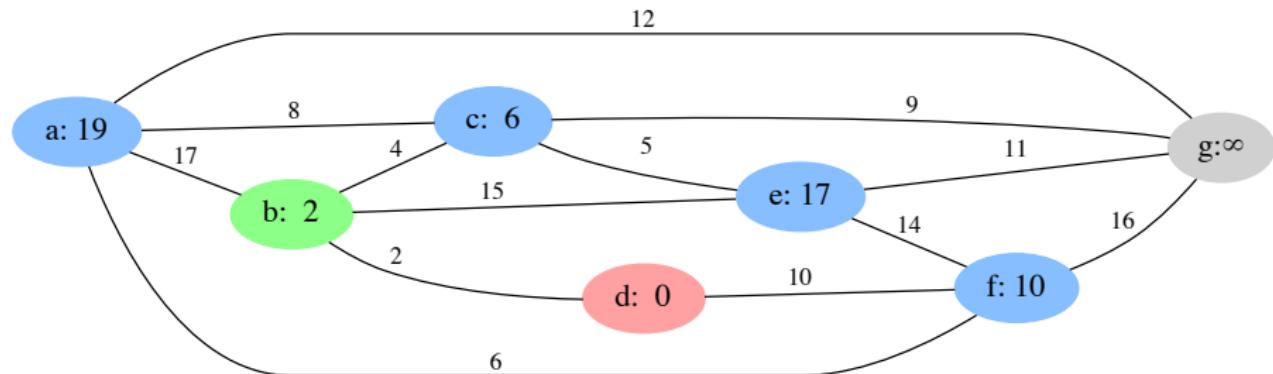
Eingabe: Graph  $(V, E)$ , Kantengewichtsfunktion  $e$ , Startknoten  $v_s$

Ausgabe: Funktion  $d : V \rightarrow \mathbb{N}$  mit Entferungen von  $v_s$

Variablen: Liste der besuchten Knoten  $B$ , aktueller Knoten  $v_c$

- 1 Setze  $d(v_s) = 0$ ,  $d(v_i) = \infty$  für alle  $v_i \neq v_s$ ,  $B = \emptyset$
- 2 Setze  $v_c = v_s$
- 3 Für alle Nachbarn  $v$  von  $v_c$ :
  - 1 Berechne  $d_{tmp} = d(v_c) + e(v_c, v)$
  - 2 Wenn  $d_{tmp} < d(v)$  gilt, setze  $d(v) = d_{tmp}$
- 4 Füge  $v_c$  zu  $B$  hinzu
- 5 Wenn  $B = V$  gilt: fertig  
(oder wenn für alle Knoten  $v \in V \setminus B$  gilt:  $d(v) = \infty$ )
- 6 Sonst: Wähle als neuen  $v_c$  den Knoten aus  $V \setminus B$  mit geringster Entfernung
- 7 Fahre bei 3 fort

# Dijkstra: Beispiel



## Farbschema

Normaler Knoten	Aktueller Knoten $v_c$
Gewichtet	Besucht/Fertig

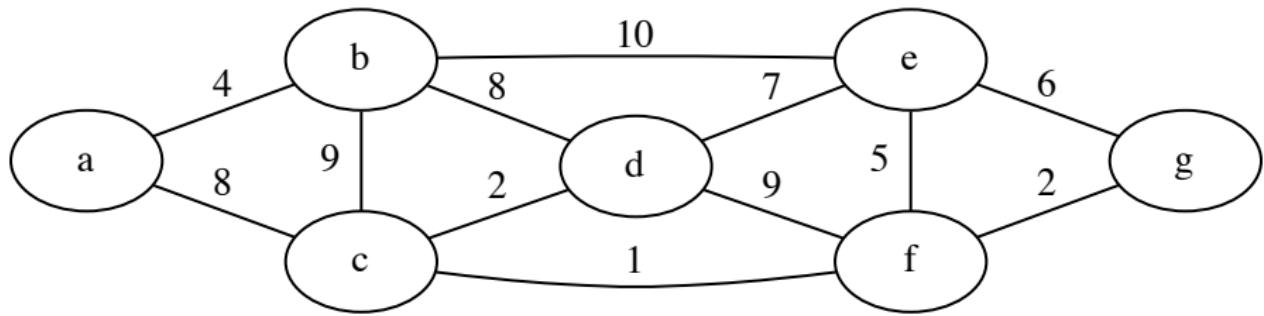
# Dijkstra Implementiert

```
def dijkstra(graph, start):
    vc = graph.node(start)
    vc.updateDist(0)
    while vc:
        for n,w in vc.adj_list:
            node = graph.node(n)
            node.updateDist(vc.dist+w)
        vc.setVisited()
        vc = None
        for n in graph.nodes.values():
            if not n.visited:
                if n.dist!=Infty:
                    if not vc or n.dist < vc.dist:
                        vc = n
```

- ▶ vc: Aktueller Knoten
- ▶ n: Knotename,
- ▶ w: Gewicht der aktuellen Kante
- ▶ node: Knotendatenstruktur
  - ▶ updateDist()
  - ▶ dist - geschätzte Distance
  - ▶ adj\_list
  - ▶ setVisited()
  - ▶ visited

# Übung: Dijkstra-Algorithmus

Bestimmen Sie die kürzesten Entfernungen vom Knoten c im folgenden Graphen.



Zusatzübung

# Dijkstra: Komplexität

- ▶ jeden Knoten besuchen:  $\mathcal{O}(|V|)$ 
  - ▶ alle Nachbarn updaten: maximal  $\mathcal{O}(|V|)$  pro Knoten  
(aber insgesamt maximal  $\mathcal{O}(|E|)$  (!))
  - ▶ nächsten Knoten finden:  $\mathcal{O}(|V|)$
- ▶ Naiv insgesamt:
  - ▶  $\mathcal{O}(|V| \cdot (|V| + |V|)) = \mathcal{O}(|V| \cdot 2|V|) = \mathcal{O}(|V|^2)$
  - ▶ Alternativ mit Anmerkung (!):  $\mathcal{O}(|E| + |V|^2)$   
(damit auch  $\mathcal{O}(|V|^2)$ , da  $|E| \leq |V|^2$ , aber oft  $|E| \ll |V|^2$ )
- ▶ Mit Heap und Adjazenzlisten geht es besser!

Ende Vorlesung 20

# Gruppenübung: Dijkstra optimieren

- ▶ Bei naiver Implementierung:  $\mathcal{O}(|E| + |V|^2)$  (und damit  $\mathcal{O}(|V|^2)$ )
- ▶ Erreichbar (mit Mitteln dieser Vorlesung):  $\mathcal{O}((|E| + |V|) \cdot \log |V|)$ 
  - ▶ Im worst-case auch  $\mathcal{O}(|V|^2)$  (denn  $|E| \leq |V|^2$ )
  - ▶ Aber die meisten Graphen sind dünn besetzt
- ▶ Frage: Wie?

# Greedy-Algorithmen

- ▶ Prim und Dijkstra sind Greedy-Algorithmen
- ▶ Greedy: Wähle lokal beste Lösung
  - ▶ billigsten neuen Knoten
  - ▶ kürzesten Weg zum Nachbarknoten
- ▶ liefert globales Optimum, weil dieses sich aus lokalen Optima zusammensetzt
- ▶ Gegenbeispiele
  - ▶ Rucksack-Problem: 5 Liter Platz im Rucksack; Wasserbehälter mit 2,3,4 Liter vorhanden
  - ▶ Problem des Handlungsreisenden: alle Städte (Knoten des Graphen) besuchen
  - ▶ Steps to one

# Zusammenfassung: Graphen

- ▶ Knoten und Kanten
- ▶ Adjazenzmatrix und -liste
- ▶ gerichtet und gewichtet
- ▶ Pfade, Zyklen und Bäume
- ▶ Minimaler Spannbaum
  - ▶ Prim-Algorithmus
    - ▶ naiv:  $\mathcal{O}(|V| \cdot |E|)$  (mit  $|E| \leq |V|^2$  also  $\mathcal{O}(|V|^3)$ )
    - ▶ optimiert:  $\mathcal{O}(\log |V| \cdot |E|)$  (mit  $|E| \leq |V|^2$  also  $\mathcal{O}(\log |V| \cdot |V|^2)$ )
- ▶ Kürzeste Wege
  - ▶ Dijkstra-Algorithmus
  - ▶ naiv:  $\mathcal{O}(|V|^2)$ , optimiert:  $\mathcal{O}((|E| + |V|) \cdot \log |V|)$

# Zusammenfassung: Algorithmen (I)

- ▶ Arten von Algorithmen
  - ▶ Dynamisches Programmieren
  - ▶ Divide and Conquer
  - ▶ Greedy
- ▶ Komplexität
  - ▶ elementare Operationen
  - ▶  $\mathcal{O}$ -Notation
  - ▶ logarithmisch, linear,  $n \log n$ , quadratisch, exponentiell, ...
  - ▶ Master-Theorem
- ▶ Datenstrukturen
  - ▶ Array
  - ▶ Liste
    - ▶ einfach verkettet
    - ▶ doppelt verkettet
    - ▶ mit Dummy-Element

# Zusammenfassung: Algorithmen (II)

- ▶ Sortieren
  - ▶ Selection Sort, Insertion Sort, Bubble Sort
  - ▶ Quicksort (LL- und LR-Pointer)
  - ▶ Mergesort (Top-Down und Bottom-Up)
  - ▶ Heapsort
  - ▶ in-place vs. out-of-place
  - ▶ stabil vs. instabil
  - ▶ direkt vs. indirekt
- ▶ Suchen
  - ▶ Binärer Suchbaum: Einfügen und Löschen
  - ▶ AVL-Baum: Rebalancieren
  - ▶ Hashing: Hash-Funktionen, Kollisionsbehandlung
- ▶ Graphen
  - ▶ gerichtet vs. ungerichtet
  - ▶ knoten- und kantenbeschriftet
  - ▶ Minimaler Spannbaum: Prim
  - ▶ Kürzeste Wege: Dijkstra

**Ende Material / Anfang Einzelvorlesungen**

# Ziele Vorlesung 1

- ▶ Kennenlernen (oder Wiedererkennen)
- ▶ Übersicht über die Vorlesung
- ▶ Was ist ein Algorithmus?
- ▶ Beispiel: Euklid
- ▶ Abschweifung: Divisionsrest

Zur Vorlesung 1

# Zusammenfassung

- ▶ Kennenlernen (oder Wiedererkennen)
- ▶ Übersicht über die Vorlesung
- ▶ Was ist ein Algorithmus?
- ▶ Beispiel: Euklid
- ▶ Modulus

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 2

- ▶ Rückblick/Wiederholung
- ▶ Euklid (zweite Runde)
- ▶ Algorithmen und Datenstrukturen
- ▶ Effizienz und Komplexität von Algorithmen

# Rückblick/Wiederholung

- ▶ Algorithmenbegriff
- ▶ Beispiel von Algorithmenklassen
  - ▶ Suchen/Sortieren
  - ▶ Optimieren
  - ▶ Kompression
  - ▶ ...
- ▶ Spezifikation von Algorithmen
  - ▶ Informal
  - ▶ Semi-Formal
  - ▶ (Pseudo-)Code
  - ▶ Flussdiagramme
  - ▶ ...
- ▶ Der GGT-Algorithmus von Euklid

Zur Vorlesung 2

# Zusammenfassung

- ▶ Rückblick/Wiederholung
- ▶ Euklid (zweite Runde)
  - ▶ Ersetze wiederholte Subtraktion durch einen Modulus
  - ▶ Zahl der Schleifendurchläufe reduziert sich von linear nach *logarithmisch*
- ▶ Algorithmen und Datenstrukturen
  - ▶ Gute Datenstrukturen ermöglichen gute Algorithmen
  - ▶ Zeitunterschied ist signifikant!
- ▶ Effizienz und Komplexität von Algorithmen
  - ▶ Abstraktion: Elementare Operationen brauchen 1 ZE

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 3

- ▶ Komplexität konkret
  - ▶ Was zählen wir?
  - ▶ Wovon abstrahieren wir?
- ▶ *Big-O Notation*
  - ▶ Definition
  - ▶ Rechenregeln

# Rückblick/Wiederholung

- ▶ Euklid (zweite Runde)
  - ▶ Analyse von Euklid (klassisch und mit Modulo)
- ▶ Algorithmen und Datenstrukturen
  - ▶ Namen unsortiert, sortiert, ...
- ▶ Effizienz und Komplexität von Algorithmen
  - ▶ Zeitkomplexität
  - ▶ (Speicher-)Platzkomplexität
  - ▶ *Average case* und *Worst case*

Zur Vorlesung 3

# Zusammenfassung

- ▶ Komplexität konkret
  - ▶ Was zählen wir? (abstrakte Operationen, abstrakte Speicherstellen)
  - ▶ Wovon abstrahieren wir? (Kosten einer konkreten Operation)
- ▶ *Big-O Notation*
  - ▶ Definition
$$g \in \mathcal{O}(f) \text{ gdw. } \exists k \in \mathbb{N} \quad \exists c \in \mathbb{R}^{\geq 0} \quad \forall n > k : g(n) \leq c \cdot f(n)$$
  - ▶ Rechenregeln
    - ▶ Konstanter Faktor
    - ▶ Summe
    - ▶ Transitivität
    - ▶ Grenzwertbetrachtung

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 4

- ▶ Rückblick
- ▶ *Big-O Notation*
  - ▶ Grenzwertregel
  - ▶ l'Hôpital
  - ▶ Beispiele

# Rückblick/Wiederholung

- ▶ Komplexität konkret
  - ▶ Was zählen wir?
  - ▶ Wovon abstrahieren wir?
- ▶ *Big-O Notation*
  - ▶ Definition

## $\mathcal{O}$ -Notation

Für eine Funktion  $f$  bezeichnet  $\mathcal{O}(f)$  die Menge aller Funktionen  $g$  mit

$$\exists k \in \mathbb{N} \quad \exists c \in \mathbb{R}^{\geq 0} \quad \forall n > k : g(n) \leq c \cdot f(n)$$

- ▶ Rechenregeln
  - ▶ Konstanter Faktor
  - ▶ Summe
  - ▶ Transitivität
  - ▶ Grenzwertbetrachtung

# Zusammenfassung

- ▶ Rückblick
- ▶ *Big-O Notation*
  - ▶ Definition
  - ▶ Rechenregeln
  - ▶ Beispiele

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 5

- ▶ Rückblick
- ▶ Logarithmen
  - ▶ Ideen
  - ▶ Rechenregeln
  - ▶ Anwendungen
- ▶ Fakultät ( $n!$ ) und die Stirling-Formel

# Rückblick/Wiederholung

- ▶ *Big-O Notation*
  - ▶ Für eine Funktion  $f$  bezeichnet  $\mathcal{O}(f)$  die Menge aller Funktionen  $g$  mit  $\exists k \in \mathbb{N} \quad \exists c \in \mathbb{R}^{\geq 0} \quad \forall n > k : g(n) \leq c \cdot f(n)$
- ▶ Rechenregeln
  - ▶  $f \in \mathcal{O}(f)$
  - ▶ Konstanter Faktor
  - ▶ Summe
  - ▶ Transitivität
  - ▶ Grenzwert!
- ▶ Regel von l'Hôpital
- ▶ Beispiele
  - ▶ Offen:  $n^n$  vs.  $n \cdot 2^n$

Zur Vorlesung 5

# Zusammenfassung

- ▶ Rückblick
- ▶ Logarithmen
  - ▶ Ideen
  - ▶ Rechenregeln
  - ▶ Anwendungen
- ▶ Wir können  $n!$  für große  $n$  annähern
  - ▶  $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$
  - ▶  $n! \in \mathcal{O}(n^n)$
  - ▶  $n! \notin \mathcal{O}(e^{c \cdot n})$

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 6

- ▶ Rückblick
- ▶ Beispiele für Komplexitäten
- ▶ Algorithmenansätze und Komplexität
  - ▶ Iteration
  - ▶ Rekursion
- ▶ Fibonacci optimiert
  - ▶ Einführung in dynamische Programmierung

# Rückblick/Wiederholung

- ▶ Analyse:  $n^n$  gegen  $n \cdot 2^n$
- ▶ Logarithmen
  - ▶ Ideen
  - ▶ Rechenregeln
  - ▶ Anwendungen
- ▶ Stirlingsche Formel, Folgerung:  $n!$  steigt schneller als  $e^{c \cdot n}$  für beliebiges  $c$ , langsamer als  $n^n$

Zur Vorlesung 6

# Zusammenfassung

- ▶ Rückblick
- ▶ Beispiele für Komplexitäten
- ▶ Algorithmenansätze und Komplexität
  - ▶ Iteration
  - ▶ Rekursion
- ▶ Fibonacci optimiert
  - ▶ Fibonacci naiv rekursiv: Langsam
  - ▶ Fibonacci iterativ: Schwierig zu verallgemeinern
  - ▶ Allgemeiner Ansatz: Tabulieren von Zwischenergebnissen

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 7

- ▶ Rückblick
- ▶ Dynamische Programmierung
  - ▶ Beispiele
  - ▶ Voraussetzungen
  - ▶ Grenzen
- ▶ Komplexität rekursiver Programme
  - ▶ Rekurrenzrelationen

# Rückblick/Wiederholung

- ▶ Komplexitätsklassen
  - ▶ „Gut“:  $\mathcal{O}(1)$ /konstant,  $\mathcal{O}(\log n)$ /logarithmisch,  $\mathcal{O}(n)$  (linear,  $\approx 10\text{GB/s}$ )
  - ▶ „Oft gut genug“:  $\mathcal{O}(n \log n)$  ( $\approx 350\text{ MB/s}$ )
  - ▶ „Geht manchmal noch“:  $\mathcal{O}(n^2)$ ,  $\mathcal{O}(n^3)$
  - ▶ „I.a. schwierig“:  $\mathcal{O}(2^n)$  ( $\approx 33\text{ Worte/s}$ ) ...
- ▶ Dynamische Programmierung
  - ▶ Fibonacci naive rekursiv/iterative/DP

Zur Vorlesung 7

# Zusammenfassung

- ▶ Rückblick
- ▶ Elegante und schnelle Algorithmen: Dynamische Programmierung
  - ▶ Gesamtlösung setzt sich aus Teillösungen zusammen
  - ▶ Überlappende Teilprobleme
  - ▶ Grenzen
- ▶ Komplexität rekursiver Programme
  - ▶ Rekurrenzrelationen

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 8

- ▶ Rückblick
- ▶ Divide-und-Conquer-Algorithmen (2)
  - ▶ Binäre Suche
- ▶ Komplexität rekursiver Programme (2)
  - ▶ Rekurrenz-Relationen und Lösungsansätze
  - ▶ Das Master-Theorem

- ▶ Dynamische Programmierung
  - ▶ Idee: Speichere Teilergebnisse und verwende sie wieder
  - ▶ Beispiel: Fibonacci
  - ▶ Beispiel:  $n$  nach 1
- ▶ Komplexität rekursiver Programme (1)
  - ▶ Beispiel:  $m^n$  naiv  $O(n)$ , clever  $O(\log n)$
  - ▶ Kosten mit Rekurrenzrelationen abschätzen
- ▶ Divide and Conquer
  - ▶ Beispiel: Effizientes Potenzieren

Zur Vorlesung 8

# Zusammenfassung

- ▶ Rückblick
- ▶ Divide-und-Conquer-Algorithmen (2)
  - ▶ Binäre Suche
- ▶ Komplexität rekursiver Programme (2)
  - ▶ Rekurrenz-Relationen und Lösungsansätze
  - ▶ Das Master-Theorem und Übungen

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 9

- ▶ Rückblick
- ▶ Die Geschwister von  $\mathcal{O}$ :  $\Omega$ ,  $\Theta$  und  $\sim$
- ▶ Arrays

# Rückblick/Wiederholung

- ▶ Rückblick
- ▶ Divide-und-Conquer-Algorithmen
  - ▶ Binäre Suche
- ▶ Komplexität rekursiver Programme
  - ▶ Rekurrenz-Relationen und Lösungsansätze
  - ▶ Das Master-Theorem und Übungen

## Master-Theorem

$$f(n) = \underbrace{a \cdot f\left(\frac{n}{b}\right)}_{\text{rekursive Berechnung der Teillösungen}} + \underbrace{c(n)}_{\text{Teilen und Rekombinieren}} \quad \text{mit } c(n) \in \mathcal{O}(n^d)$$

wobei  $a \in \mathbb{N}^{\geq 1}$ ,  $b \in \mathbb{N}^{\geq 2}$ ,  $d \in \mathbb{R}^{\geq 0}$ . Dann gilt:

- 1**  $a < b^d \Rightarrow f(n) \in \mathcal{O}(n^d)$
- 2**  $a = b^d \Rightarrow f(n) \in \mathcal{O}(\log_b n \cdot n^d)$
- 3**  $a > b^d \Rightarrow f(n) \in \mathcal{O}(n^{\log_b a})$

Zur Vorlesung 9

# Zusammenfassung

- ▶ Die Geschwister von  $\mathcal{O}$ :  $\Omega$ ,  $\Theta$  und  $\sim$
- ▶ Arrays
  - ▶ Zugriff per Index ist  $\mathcal{O}(1)$ !
  - ▶ Einfügen/Ausfügen mit/ohne Beibehaltung der Reihenfolge hat verschiedene Komplexitäten
  - ▶ Als Konsequenz: Sortierte/unsortierte Arrays haben sehr verschiedene Stärken und Schwächen

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 10

- ▶ Rückblick
- ▶ Listen
  - ▶ Einfach verkettet
  - ▶ Doppelt verkettet

# Rückblick/Wiederholung

- ▶  $\mathcal{O}, \Omega, \Theta, \sim$ 
  - ▶ Irgendwann im wesentlichen nicht schneller -  $\mathcal{O}$
  - ▶ Irgendwann im wesentlichen nicht langsamer -  $\Omega$
  - ▶ Irgendwann im wesentlichen genau so schnell -  $\Theta$
  - ▶ Irgendwann genau so schnell -  $\sim$
- ▶ Arrays
  - ▶ Definition
  - ▶ Organisation im Speicher
  - ▶ Verwendung/Datenhaltung in Arrays
  - ▶ Standard-Operationen

Zur Vorlesung 10

# Zusammenfassung

- ▶ Listen
  - ▶ Einfach verkettet
    - ▶ Struktur
    - ▶ Zyklenerkennung (Hase/Igel)
  - ▶ Doppelt verkettet
    - ▶ Struktur (Tipp: Ankerzelle)
    - ▶ Einfügen/Ausfügen
  - ▶ Scorecard

# Hausaufgabe

Bestimmen Sie mit dem Master-Theorem Abschätzungen für die folgenden Rekurrenzen, oder geben Sie an, warum das Theorem nicht anwendbar ist. Finden Sie in diesen Fällen eine andere Lösung?

- ▶  $T(n) = 3T\left(\frac{n}{2}\right) + n^2$
- ▶  $T(n) = 7T\left(\frac{n}{2}\right) + n^2$
- ▶  $T(n) = 4T\left(\frac{n}{2}\right) + n \log n$
- ▶  $T(n) = 4T\left(\frac{n}{2}\right) + \log n$
- ▶  $T(n) = T(n - 1) + n$
- ▶  $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + n^2$
- ▶  $T(n) = 2T\left(\frac{n}{4}\right) + \log n$

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 11

- ▶ Rückblick
- ▶ Grundlagen des Sortierens
  - ▶ Klassifikation
- ▶ Einfache Sortierverfahren
  - ▶ Selection Sort
  - ▶ Insertion Sort
  - ▶ Bubble Sort

# Rückblick/Wiederholung

- ▶ Einfach verkettete Listen
  - ▶ Struktur
  - ▶ Zyklenerkennung mit Hase und Igel
- ▶ Doppelt verkettete Listen
  - ▶ Struktur
  - ▶ Einfügen/Ausfügen
- ▶ Eigenschaften
  - ▶ Listen und Arrays
  - ▶ Operationen mit Komplexitäten

Zur Vorlesung 11

# Zusammenfassung

- ▶ Grundlagen des Sortierens
  - ▶ Klassifikation
- ▶ Einfache Sortierverfahren
  - ▶ Selection Sort
  - ▶ Insertion Sort
  - ▶ Bubble Sort

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 12

- ▶ Rückblick
- ▶ Diskussion der einfachen Verfahren
- ▶ Sortieren mit Divide-and-Conquer (1)
  - ▶ Quicksort

# Rückblick/Wiederholung

- ▶ Klassifikationskriterien
  - ▶ Verwendete Datenstrukturen
  - ▶ Verhältnis der benötigten Operationen
  - ▶ Benötigter zusätzlicher Speicher (in-place vs. out-of-place)
  - ▶ Stabilität: Auswirkung auf Elemente mit gleichem Schlüssel
- ▶ Einfache Sortierverfahren
  - ▶ Selection Sort
  - ▶ Insertion Sort
  - ▶ Bubble Sort
  - ▶ <https://www.youtube.com/watch?v=kPRA0W1kECg>

Zur Vorlesung 12

# Zusammenfassung

- ▶ Diskussion der einfachen Verfahren
- ▶ Sortieren mit Divide-and-Conquer (1)
  - ▶ Quicksort (Teile nach Größe per Pivot)
    - ▶ Komplexität:  $\mathcal{O}(n \log n)$  im Durchschnitt,  $\mathcal{O}(n^2)$  im schlechtesten Fall
  - ▶ Alternative: Teilung nach Position (Mergesort → nächstes Mal)

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 13

- ▶ Rückblick
- ▶ Sortieren mit Divide-and-Conquer (2)
  - ▶ Mergesort - Algorithmus
  - ▶ Mergesort - Komplexität
- ▶ Bottom-up Mergesort

# Rückblick/Wiederholung

- ▶ Sortieren mit Divide-and-Conquer 1: Quicksort
  - ▶ Rate Pivot  $p$
  - ▶ Teile Array in Teil  $< p$ ,  $p$ , Teil  $\geq p$
  - ▶ Sortiere die Teile rekursiv
- ▶ Analyse Quicksort - Best case: Pivot zerlegt in Hälften
  - ▶ Master-Theorem:  $q(n) = 2q(\lfloor \frac{n}{2} \rfloor) + kn$
  - ▶  $a = 2, b = 2, d = 1$ : Fall 2:  $q(n) \in \mathcal{O}(\log_2 n \cdot n^1) = \mathcal{O}(n \log n)$
- ▶ Analyse Quicksort - Worst-case: Pivot ist kleinstes (analog größtes)
  - ▶  $q(n) = kn + q(n - 1)$  (ohne Master-Theorem)
  - ▶ Also:  $q(n) \in \mathcal{O}(n^2)$

Zur Vorlesung 13

# Zusammenfassung

- ▶ Sortieren mit Divide-and-Conquer (2)
  - ▶ Mergesort - Algorithmus
  - ▶ Mergesort - Komplexität
- ▶ Bottom-up Mergesort

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 14

- ▶ Rückblick
- ▶ Heaps
  - ▶ Partiell geordnete Bäume
  - ▶ Effiziente Implementierung von Prioritäten
  - ▶ Einbettbar in Arrays

# Rückblick/Wiederholung

- ▶ Sortieren mit Divide-and-Conquer (2)
  - ▶ Mergesort - Algorithmus
  - ▶ Mergesort - Komplexität
- ▶ Bottom-up Mergesort

**Hinweis:** Vorträge der ersten *International Research Conference On The History Of Computing* (1976), u.a. Stanislaw M. Ulam, Edsger W. Dijkstra, Donald Knuth, John Bakkus, Konrad Zuse, Friedrich L. Bauer

[https://computerhistory.org/playlists/  
international-research-conference-on-the-history-of-computing/](https://computerhistory.org/playlists/international-research-conference-on-the-history-of-computing/)

Zur Vorlesung 14

# Zusammenfassung

- ▶ Heaps
  - ▶ Fast vollständige Binärbäume
  - ▶ Einbettbar in Arrays
  - ▶ Eltern sind  $\geq$  (bzw.  $\leq$ ) als die Kinder
  - ▶ Operationen: Bubble-up, Bubble-down, Heapify, ...

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 15

- ▶ Rückblick
- ▶ Heapsort
  - ▶ Heap-Basiertes Selection-Sort
  - ▶ Effizient
  - ▶ In-Place
- ▶ Sortieren: Abschluss
  - ▶ *As good as it gets*
- ▶ Einführung: Suchen und Finden

# Rückblick/Wiederholung

- ▶ (Binäre) Heaps
  - ▶ Fast vollständige Binärbäume
  - ▶ Eltern sind  $\geq$  (bei Maxheaps) bzw.  $\leq$  (bei Minheaps) als die Kinder
  - ▶ Heap Eigenschaften
    - ▶ Shape
    - ▶ Heap
    - ▶ Heaps als Arrays
- ▶ Operationen auf Heaps
  - ▶ Einfügen und bubble-up
  - ▶ Ausfügen (der Wurzel) und bubble-down
  - ▶ Heapify
- ▶ Operationen: Bubble-up, Bubble-down, Heapify, ...

Zur Vorlesung 15

# Zusammenfassung

- ▶ Heapsort
  - ▶ Heapify
  - ▶ Wiederholtes Select-Maxi
  - ▶  $\Theta(n \log n)$
- ▶ Sortieren: Abschluss
  - ▶ Viel besser als  $\mathcal{O}(n \log n)$  geht es nicht
  - ▶ Mergesort ist *nahezu optimal*
- ▶ Anforderungen an Schlüsselverwaltungen

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 16

- ▶ Rückblick
- ▶ Binäre Suchbäume
  - ▶ Definition/Struktur
  - ▶ Einfügen
  - ▶ Sortierte Ausgabe/Iteration
  - ▶ Löschen

# Rückblick/Wiederholung

- ▶ Heapsort
  - ▶ Heapify Array
  - ▶ Tausche Größtes gegen Letztes
  - ▶ Verkleinere Heap um ein Element
  - ▶ Bubble-Down der Wurzel
  - ▶ ... bis das Array sortiert ist
- ▶ Abschluss Sortieren
  - ▶ Selektieren einer Permutation aus  $n!$
  - ▶ Erfordert  $\Theta(\log(n!))$  Bits
  - ▶ Also  $\Theta(n \log n)$  (mit Stirling)
- ▶ Dictionaries
  - ▶ Verwalten Schlüssel/Wert-Paare
  - ▶ Uns interessiert meist nur der Schlüssel - den Wert denken wir uns ;-)
  - ▶ Operationen:
    - ▶ Anlegen, Einfügen, Suchen, Löschen, Ausgabe, ...

# Zusammenfassung

- ▶ Binäre Suchbäume
  - ▶ Definition
  - ▶ Suche
  - ▶ Einfügen
  - ▶ Analyse
    - ▶ Best case:  $\mathcal{O}(\log n)$
    - ▶ Worst case:  $\mathcal{O}(n)$
- ▶ Löschen
  - ▶ Blatt/Knoten mit einem Nachfolger
  - ▶ 2 Nachfolger: Suche und tausche Ersatzknoten

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 17

- ▶ Rückblick
- ▶ Balancierte binäre Suchbäume
  - ▶ Definitionen
  - ▶ Rotationen
  - ▶ AVL-Bäume

# Rückblick/Wiederholung

- ▶ Binäre Suchbäume
  - ▶ Definition
  - ▶ Suche
  - ▶ Einfügen
  - ▶ Analyse
    - ▶ Best case:  $\mathcal{O}(\log n)$
    - ▶ Worst case:  $\mathcal{O}(n)$
- ▶ Löschen
  - ▶ Blatt/Knoten mit einem Nachfolger
  - ▶ 2 Nachfolger: Suche und tausche Ersatzknoten

Zur Vorlesung 17

# Zusammenfassung

- ▶ Balancierte binäre Suchbäume
  - ▶ Definitionen
  - ▶ Rotationen
- ▶ AVL-Bäume
  - ▶ Höhenbalancierte Binärbäume mit maximalem Höhenunterschied 1
  - ▶ Einfügen: Normal, lokale Rotation, wenn AVL-Eigenschaft verletzt ist
  - ▶ Löschen: Normal, Rotationen auf dem Pfad
  - ▶ <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 18

- ▶ Rückblick
- ▶ Hashing
  - ▶ Idee
  - ▶ Hash-Funktionen
  - ▶ Kollisionsbehandlung
  - ▶ Analyse

# Rückblick/Wiederholung

- ▶ Balancierte binäre Suchbäume
  - ▶ Definitionen
  - ▶ Rotationen
- ▶ AVL-Bäume
  - ▶ Höhenbalancierte Binäräbäume mit maximalem Höhenunterschied 1
  - ▶ Einfügen: Normal, lokale Rotation, wenn AVL-Eigenschaft verletzt ist
  - ▶ Löschen: Normal, Rotationen auf dem Pfad

Zur Vorlesung 18

# Zusammenfassung

- ▶ Hashing
  - ▶ Idee
  - ▶ Hash-Funktionen
    - ▶ Oft basierend auf Modulus
  - ▶ Kollisionsbehandlung
  - ▶ Analyse

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 19

- ▶ Rückblick: Hashing
- ▶ Graphalgorithmen (1)
  - ▶ Graphen
    - ▶ Definitionen
    - ▶ Anwendungen
    - ▶ Repräsentation
  - ▶ Problem: Minimaler Spannbaum
    - ▶ Prim's Algorithmus

- ▶ Hashing
  - ▶ Idee: Arrays sind  $\mathcal{O}(1)$  für Zugriffe über den Index
  - ▶ Bilde große/unendliche Schlüsselmenge auf kleine Menge von Hashes ab
  - ▶ Verwende Hashes als Indices in ein Array
  - ▶ Kollisionsbehandlung
  - ▶ Performance:  $\mathcal{O}(1)$  für die meisten Operationen

Zur Vorlesung 19

# Zusammenfassung

- ▶ Rückblick: Hashes
- ▶ Graphalgorithmen
  - ▶ Graphen
    - ▶ Definitionen
    - ▶ Anwendungen
    - ▶ Repräsentation
  - ▶ Problem: Minimaler Spannbaum
    - ▶ Prim's Algorithmus

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 20

- ▶ Rückblick: Graphen, Prim
- ▶ Komplexität:
  - ▶ Prim naiv vs. Prim optimiert
- ▶ Routing: Dijkstra
  - ▶ Algorithmus
  - ▶ Implementierung
  - ▶ Komplexität

# Rückblick/Wiederholung

- ▶ Graphen:  $V, E$  (Knoten und Kanten)
  - ▶ Gerichtet, ungerichtet
  - ▶ Pfade, Zyklen, Verbundenheit, Bäume
- ▶ Gewichtete Graphen:
  - ▶ Beschriftungsfunktion (z.B.:  $e : E \rightarrow \mathbb{N}$ )
- ▶ Anwendungen
- ▶ Repräsentationen
  - ▶ Adjazenzmatrix
  - ▶ Adjazenzlisten
- ▶ Minimaler Spannbaum
  - ▶ Algorithmus von Prim (Greedy)
  - ▶ Naiv:  $\mathcal{O}(|V| \cdot |E|)$
  - ▶ Geht es besser?

Zur Vorlesung 20

# Zusammenfassung

- ▶ Komplexität:
  - ▶ Prim naiv vs. Prim optimiert
- ▶ Routing: Dijkstra
  - ▶ Algorithmus
  - ▶ Implementierung
  - ▶ Komplexität

# Feedback

- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

# Ziele Vorlesung 21

- ▶ Rückblick
- ▶ Dijkstra Teil 2
  - ▶ Optimierungen
- ▶ Greedy-Algorithmen und ihre Grenzen
- ▶ Zusammenfassung der Vorlesung

# Rückblick/Wiederholung

- ▶ Komplexität Prim
  - ▶ Prim naiv:  $\mathcal{O}(|V| \cdot |E|)$
  - ▶ Prim optimiert:  $\mathcal{O}(|E| \cdot \log |V|)$
- ▶ Routing: Dijkstra
  - ▶ Algorithmus
  - ▶ Implementierung
  - ▶ Komplexität

Zur Vorlesung 21

# Zusammenfassung

- ▶ Rückblick
- ▶ Dijkstra Teil 2
  - ▶ Optimierungen
- ▶ Greedy-Algorithmen und ihre Grenzen
- ▶ Zusammenfassung der Vorlesung

# Feedback

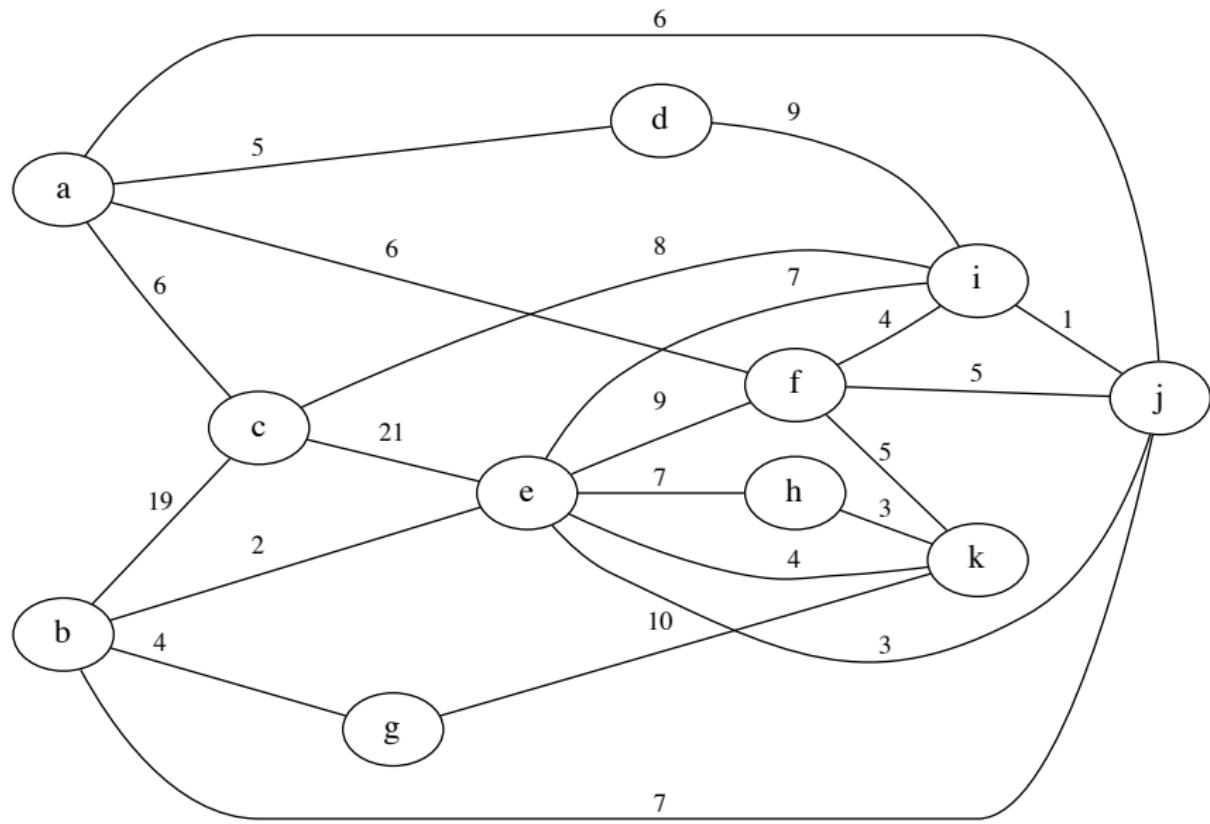
- ▶ Was war der beste Teil der heutigen Vorlesung?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?

**Klausurvorbereitung: Übungsklausur**

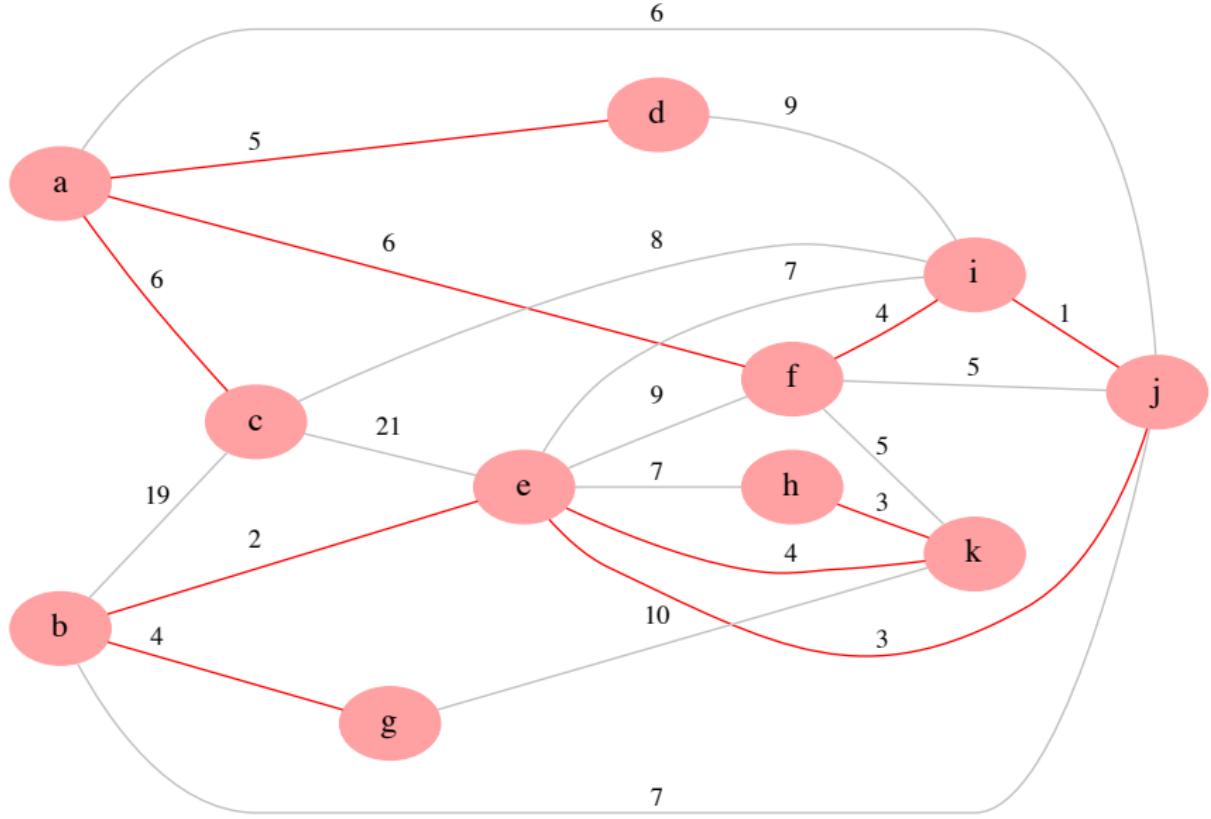
# Feedback

- ▶ Wie war die Vorlesung insgesamt?
- ▶ Was kann verbessert werden?
  - ▶ Optional: Wie?
- ▶ Feedback auch gerne über das Online-System
  - ▶ Einladung kommt (hoffentlich) per Email!

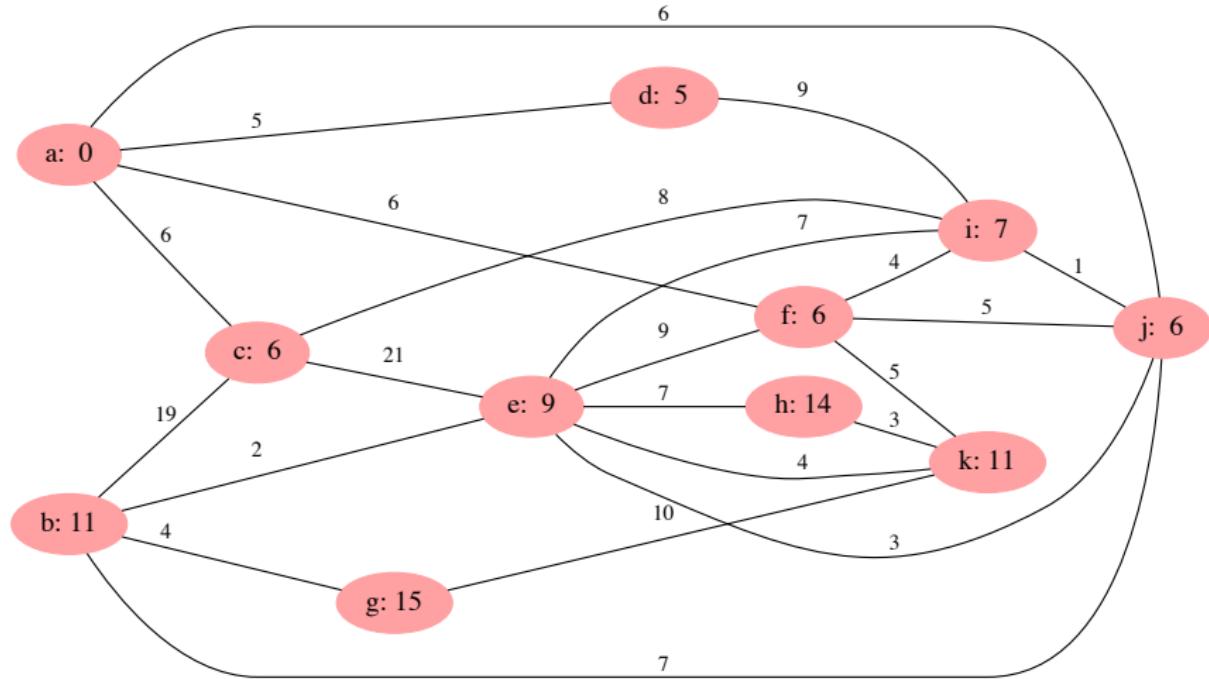
# Übung: Zusatzgraph 1



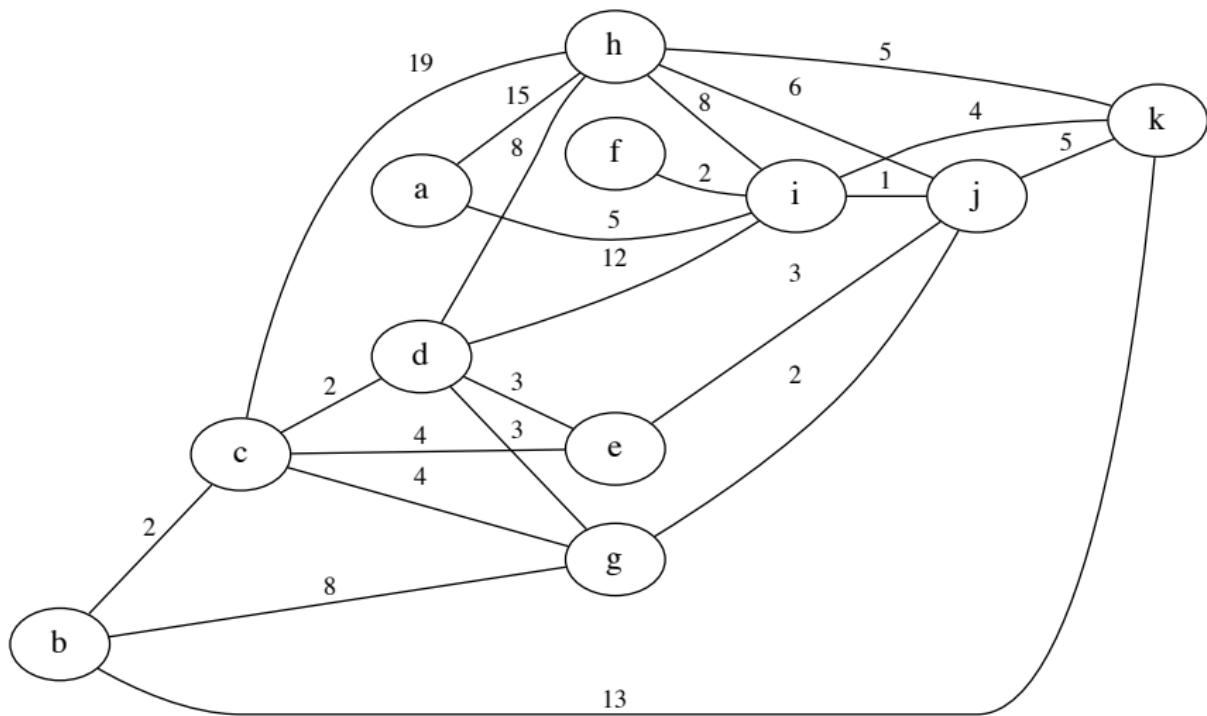
# Lösung: Zusatzgraph 1 - Prim



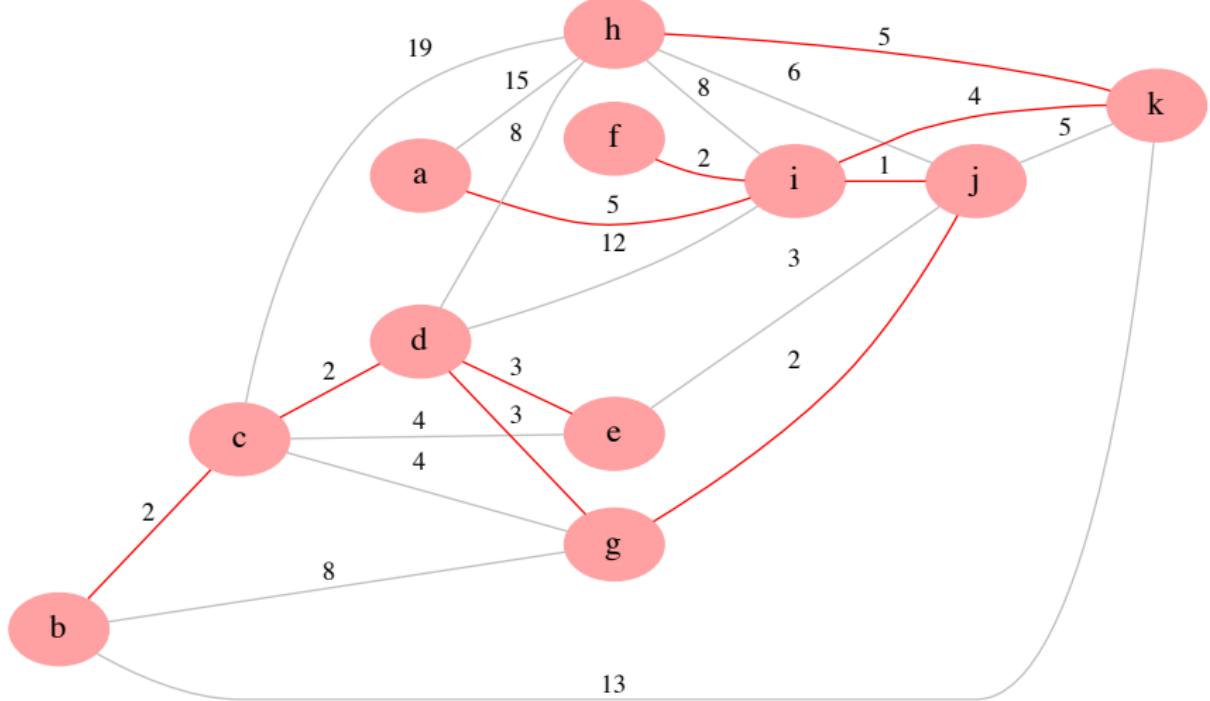
# Lösung: Zusatzgraph 1 - Dijkstra



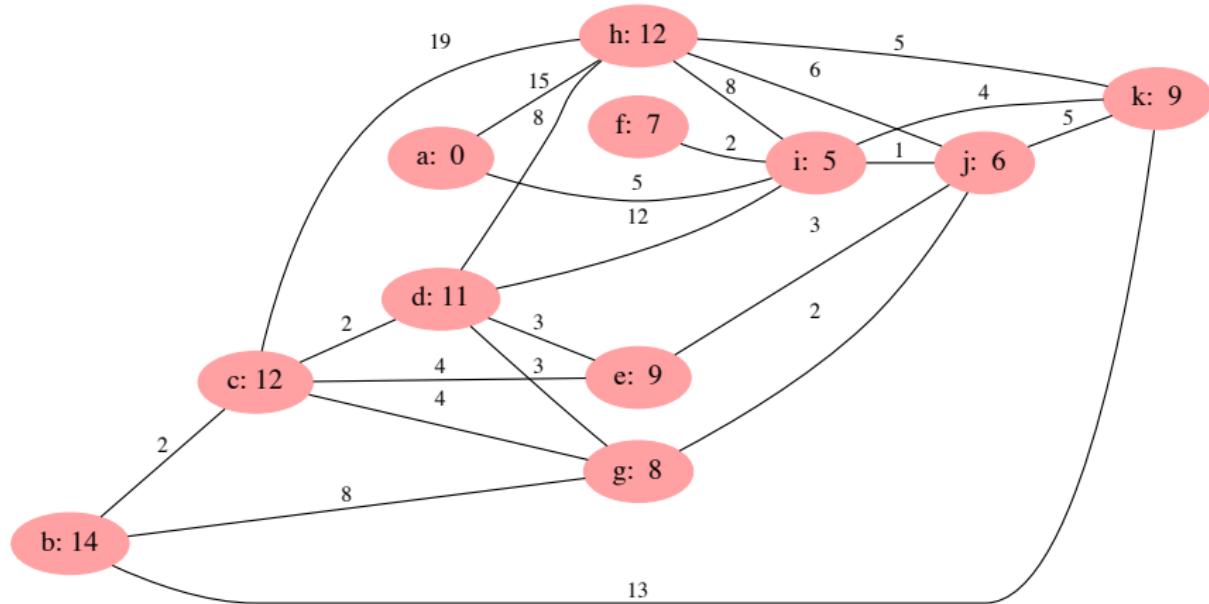
# Übung: Zusatzgraph 2



# Lösung: Zusatzgraph 1 - Prim



# Lösung: Zusatzgraph 1 - Dijkstra



Zurück (Prim)

Zurück (Dijkstra)