Algorithmen und Datenstrukturen
Sommersemester 2020

Ullrich Köthe
Heidelberg Collaboratory for Image Processing (HCI)
Interdisciplinary Center for Scientific Computing (IWR)
Universität Heidelberg
Mathematikon B (Berliner Str. 43), 69120 Heidelberg
ullrich.koethe@iwr.uni-heidelberg.de

Version 1.0 — Johanna Riedel

Erstellt: 11. Oktober 2020

URL zur Vorlesung:
https://hci.iwr.uni-heidelberg.de/teaching/iad_2020

Wiki zur Vorlesung:
http://alda.iwr.uni-heidelberg.de/index.php/Main_Page

Inhaltsverzeichnis

1 Einführung

1.1 Algorithmus, Problem, elementare Schritte

Algorithmus

- löst ein bestimmtes Problem
- braucht dafür endlich viele Schritte
- alle Schritte sind elementar

Problem

- formal beschrieben: Spezifikation

1. Vorbedingung: in welchem Zustand muss die „Welt“sein, damit der Algorithmus ausgeführt werden kann?
   → falls Vorbedingung nicht erfüllt: Fehlermeldung (nicht stillschweigend falsches Ergebnis)
2. Nachbedingungen: in welchem Zustand ist die „Welt“nach Ende des Algorithmus
   → wie kann man feststellen, dass der Algorithmus korrekt durchgelaufen ist?

Beispiel: $y = \sqrt{x}$

Vorbedingungen :

- $x \in \mathbb{R}$ oder $x \in \mathbb{N}$
- $x \geq 0$

Nachbedingung:

- $y * y = y^2 = x$

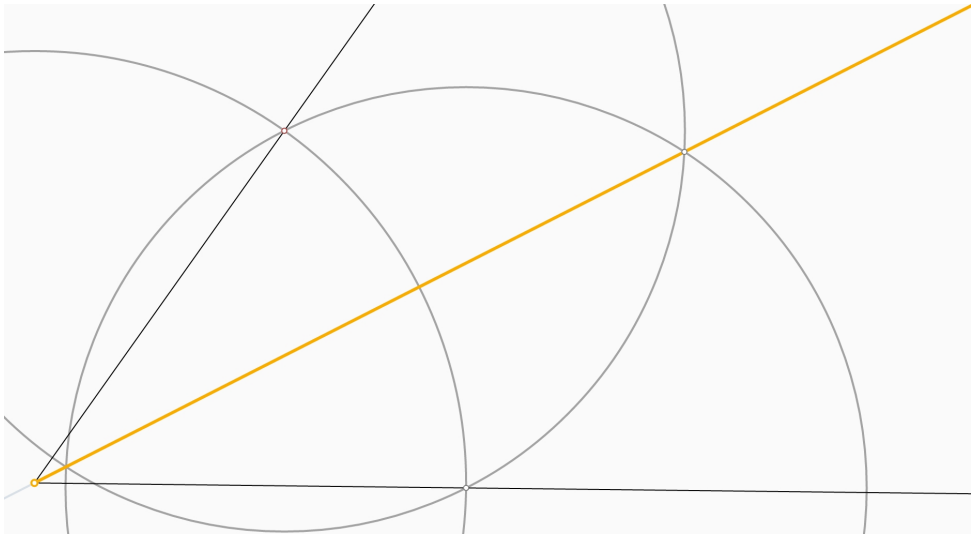
Varianten:

- falls $x < 0 \rightarrow$ Alg. gibt Fehlermeldung
- falls $x < 0 \rightarrow y = NaN$ (“not a number”: spezieller Zahlenwert für genau diesen Zweck, wenn etwas nicht berechnet werden kann) Vorteil: nicht direkt Programmabbruch weil Fehlermeldung

Elementare Schritte

charakterisieren das Gerät (bzw. Menschen), der den Algorithmus ausführen soll ("Spielregeln")

- Beispiel: Geometrie mit Zirkel und Lineal (alte Griechen)



Winkelhalbierende eines Winkels: elementare Schritte:

1. einen Punkt definieren (-beliebig oder - Schnittpunkt von Linien)
2. mit Zirkel Abstand von 2 Punkten abgreifen
3. mit Zirkel einen Kreis um einen bestimmten Punkt zeichnen
4. Zwei Punkte mit dem Lineal verbinden

- Abakisten vs. Algorithmiker (1200 bis 1500)
  - Abakisten rechnen mit römischen Zahlen und Abakus
  - Algorithmiker (al Quarismi: Rechnen mit indischen / arabischen Ziffern (ursprünglich aus Indien) schriftliches Rechnen wie in der Grundschule) 800
  - 1200 lateinische Übersetzung "Dixit Algorismi" ← Herkunft Wort
  - Vereinigung um 15000 : Adam Riese (Rechnen auf den Federn und Linien)

- elementare Schritte im modernen Computer:
  - λ-Kalkül
  - rekursive Funktionen (Gödel)
  - while-Programme
  - Turing-Maschinen

  } ganz untersch. Sammlungen von elementaren Schritten

  Aber: die Menge der Algorithmen, die man damit implementieren kann, sind identisch! "Menge der berechenbaren Funktionen" → worüber die Informatik spricht

- while-Programme:
  in verbesserter Form von allen CPUs implementiert

- vier Grundoperationen:
  * Addition einer Konstanten : `x[j] = x[i] +c`
    `x[j]` = Inhalt der Speicherstelle j; `c` = Konstante
  * Subtraktion einer Konstanten: `x[j] = x [i] - c`
    " =" → Zuweisung
  * Nacheinanderausführung von Programmen P und Q → `P ; Q` (erst `P`, dann `Q`)
  * Schleife: `WHILE x[i] != 0 DO P DONE`
    Programm `P` sollte `x[i]` irgendwann auf `0` setzen, sonst Endlosschleife $\widehat{=}$ kein Algorithmus

– erstaunlicher Fakt: alle berechenbaren Funktionen können mit nur 4 elementaren Operationen ausgedrückt werden:
in Backus-Naur-Notation:

```
Programm::=     x[i] = x[j] + c                                  #Addition einer Konstanten
                |       x[i] = x[j] - c                          #Subtraktion einer Konstanten
                |       Programm ; Programm                      #Nacheinanderausfuehren
                |       WHILE x[i] != 0 DO Programm DONE         #Wiederholtes Ausfuehren
```

Beispiel: Addition von zwei Speicherzellen `x[i]` und `x[j]`
Spezifikation des Algorithmus:

- Vorbedingung: `x[j] >= 0`
- Nachbedingung: `x[i]' = x[i] + x[j]` (`x[i]' = x[i]` am Ende des Alg.)
- Algorithmus:

```
WHILE x[j] != 0 DO
x[i] = x[i] + 1 ;
x[j] = x[j] - 1
DONE
```

in der Praxis ist das sehr ineffizient: Anzahl der Schritte $O(x[j])$ es geht auch mit $O(log(x[j]))$ Schritten

⇒ pragmatische Definition von elementaren Schritten

- Hardware-orientierte Definition: elementare Schritte sind alle Operationen, die die jeweilige CPU anbietet ("Assembler") ("Maschinensprache")
- Software-orientierte Definition: elementare Schritte sind alle Operationen, die die Programmiersprache, inklusive ihrer Standardbibliothek, anbietet.

1.2 Was ist eine Datenstruktur?

- Daten sind Folgen von Bits d.h. 0/1 - Folgen

$$1101, 0110, 0110, 1100$$

Bitfolgen allein haben keine Bedeutung, man braucht zusätzlich eine Interpretations-Vorschrift
$\widehat{=}$ "Datenformat"

- Beispiele:
  - Interpretation als "unsigned integer 16"

1	1	0	1	0	1	1	0	0	1	1	0	1	1	0	0
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
=33768		...												=2	=1

    33768 + ... + 8 + 4 + 0 + 0 = 54892

  - Interpretation als "signed integer 16 im 2er-Komplement"
    Regel:
    - wenn das linke Bit 0 ist ⇒ unsigned int 15
    - wenn das linke Bit 1 ist ⇒ negative Zahl

$$-10644 = \begin{cases} \bullet \text{ alle Bits negieren und 1 addieren} \Leftrightarrow \text{interpretiere Erg. als "unsigned int 15"} \\ 0010100110010011 \\ \underline{\qquad\qquad\qquad +1} \\ 0101001100101 00 \end{cases}$$

    Ausnahme:

    negieren 0111111111111111

    $+$ 1

    1000000000000000 = definiert als -2^{15}

    (die Zahl $+2^{15}$ ist in signed int 16 nicht darstellbar)

  - Interpretation als Windows-Zeichensatz, zwei 8-bit Zeichencodes

$$\underbrace{11010110}_{\text{"Ö"}} \quad \underbrace{01101100}_{\text{"l"}} \Rightarrow \text{Öl}$$

  - Interpretation als Gleitkommazahl "float 16" nach dem Standard IEEE 754

$$\underbrace{1}_{\text{sign = s}} \quad \underbrace{10101}_{\text{exponent = e}} \quad \underbrace{1001101100}_{\text{mantisse = m}}$$

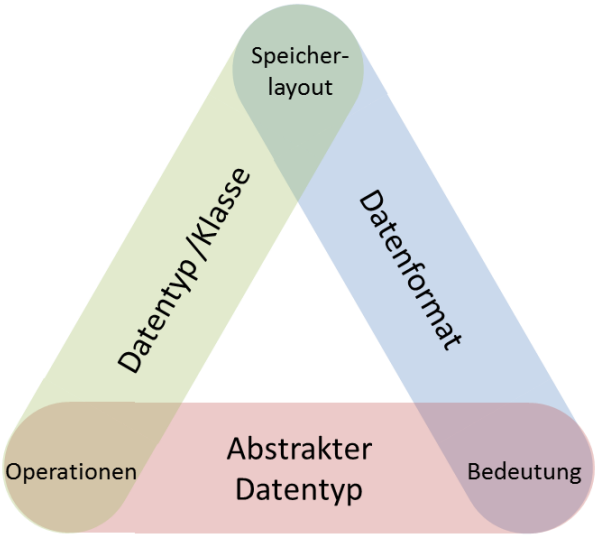
$$z = (1 - 2 * s) * 2^{e-15} * (1 + m * 2^{-10})$$
$$= -102.75$$

    usw. (unendlich viele Interpretationen)

- wichtig bei Daten in Dateien: Dateien sind Bitfolgen auf Festplatte,

  → man braucht Interpretation:
  - nach Ende des Filename: .jpg

- im Internet: Mime-types (Multipurpose Internet Mail Extension): mit den Daten verknüpfte Typ-Info
- magic numbers am Fileanfang: 255 216 255 (sonst kein .jpg, selbst wenn Endung vorhanden → evtl. Virus)
- alternative Möglichkeiten, Datenstrukturen zu definieren



Datenformat ← Files
Datentyp/Klasse ← Programmiersprache
Speicherlayout = Bitfolge
Operationen = erlaubte Operationen
Bedeutung = Interpretation
String:

- Bytefolge der Zeichen
- Operationen: print, append, to_lower_case

ADT ≙ abstract data type: Datentypen werden definiert, ohne eine spezielle Implementation als Bitfolge

⇒ Theorie, abstrakte Spezifikation von Alg. (Pseudocode)
⇒ Vorteil: Programmierer hat große Freiheiten für Implementation

1.3 Fundamentale Algorithmen

stehen für (fast) alle Typen zur Verfügung

- Konstruktor:
  1. weise einer bestimmten Speicherstelle (Bitfolge) eine Interpretation zu
  2. Initialisiere die Bitfolge mit einem festen Anfangswert (oft 0)

Beispiel: in Python heißt der Konstruktor-Algorithmus wie der Datentyp

```
1        i = int()       # ganze Zahl
2        f = float()     # Gleitkommazahl
3        l = list()      # leeres Array
```

alternative Konstruktoren für andere Anfangswerte:

```
        j = int(2)      # ganze Zahl 2 statt 0
        a = [i, j]      # Array mit den Zahlen [0,2]
```

- Vergleiche auf Gleichheit und Identität

↓	↘
zwei Variablen des gleichen Typs enthalten die gleiche Bitfolge	zwei Bezeichner (= Variablenname) referenzieren die gleiche Variable
↓	↓
==	`is`
Negation: ! =	`is not`
a = [1,2], b = [1,2] } "a == b"'ist wahr; "a! = b"ist falsch	"a `is` b" ist falsch; "a `is not` b" ist wahr

  Es gilt stets:

  - "a `is` b" wahr → "a ==b" wahr
  - "a `is` a" und "a == a" immer wahr

- swap-Operation: Vertauschen der Bitfolgen von zwei Variablen des gleichen Typs
  andere Programmiersprachen: swap(a,b)
  Python:

    Mehrfachzuweisung: a, b = b, a

    Beispiel-Verwendung: Sortieren

Referenzsemantik vs. Wertsemantik

- Wofür steht ein Variablenname in einer Programmiersprache?
- Was genau bewirkt die Zuweisung an einen Variablennamen?
- Analogie:

Bookmarks im Internet	vs.	Download einer Seite
⇓		⇓
URL, mit der man eine Seite wieder aufrufen kann aber: es könnte auch eine neue sein		lokale Kopie der Seite kann wieder aufgerufen werden aber: könnte veraltet sein
URL ≙ Referenz auf die Seite (ähnlich der Adresse einer Wohnung) ≙		Kopie ≙
Referenzsemantik		Wertsemantik
Python: für alle anderen Typen		Zahlen(int, float, boolean, string)

```
i = [1,2]
j = i           #j=[1,2]
i = [0] = 3     #i=[3,2] j=[3,2]
```

```
i=int(2)
j=i             # j==2
i=3             # j==2,
```

i und j sind nur alternative Namen für die selbe Speicherstelle	i und j verweisen auf verschiedene Speicherstellen

Freiheitsgrade bei der Datenstruktur-Definition

- Datenformat: u int8 (8 bit, als unsigned integer interpret.)
  - Speicher & Interpretation festgelegt
  - Operation: Addition: -Funktionsname “add”, “plus”, “+”
  - Implementation:

    00000001
    + 00000001
    00000010

    11111111 ≙ 255
    + 00000001
    100000000 ≙ 256 ⇒ 9 bit ??

  - Konvention:
    - es passiert *nichts*, bleibt 255
      Keine gute Idee, z.B. Kommutativität :(
      Beispiel: `i = j + k` wenn j und k vertauscht werden: Was bleibt dann erhalten?
    - Fehlermeldung: “Zahl zu groß”
      suboptimal, weil es evtl. zu oft zu Programmabbrüchen kommt

- Berechnung “modulo 256”
  $(255 + 1) mod 256 = 0$
  man rechnet zyklisch
  funktioniert ebenso im negativen Bereich (CPU: Übertrag der letzten Addition wird gelöscht)

2 Container

- Datenstrukturen, die andere Datenstrukturen enthalten
- wichtig, weil Computer benutzt werden, wenn man dieselbe Operation sehr häufig ausführen will

Beispiele:

- RGB Pixel: 3 Helligkeitswerte für Rot, Grün, Blau
- Bild: Container von RGB Pixeln (1024x1024)
- Video: Folge von Bildern (20 fps)

Grundprinzip: DS können geschachtelt werden, um mächtigere DS zu erzeugen

2.1 Abstrakte Container – Datentypen

Interpretation & Operation (Bit-Repräsentation ist dem Programmierer überlassen)

Notation: Operationen werden durch Vor- und Nachbedingung spezifiziert, dabei ist `x` die DS vor der Operation, `x'` nach der Operation

3 Arten von Operationen:

- Konstruktoren: erzeugen eine neue DS mit definiertem Anfangszustand
- Accessoren: aktuellen Zustand (=Inhalt des Containers) abfragen
- Modifizierer: aktuellen Zustand ändern

2.2 Grundlegende Container

2.2.1 Array

speichert Objekte im Speicher hintereinander ab

Ob1	Ob2	Ob3	...						ObN

Zugriff auf Objekte erfolgt über den Index $\widehat{=}$ laufende Nummer Position im Array

Denkschulen:

- zero-band indexing (C, C++, Python)
  0-based: 0, 1, ..., N-1 (N = Arraygröße)

- one-based indexing (Fortran, Matlab, Julia)
  1-based: 1, 2, ..., N

ADT:

Op	Bedeutung	Axiom

`a = new_array(size, initial)`	Konstruktor	`len(a) == size`
		$\forall i \in [0, size-1]$:
		`get(a,i) == initial`

`v=get(a,i)`	Zugriff auf Element i	Vorbedingung: $i \in [0, size-1]$ Nachbedingung: `v` == i-tes El. `a' == a`
`set(a, i, v)`	Setzen des i-ten Elements	Vorbed: $i \in [0, size-1]$ Nachbed.: `get(a',i) == v` $\forall k \neq i$: `get(a',k) == get(a,k)`

in Python:

get ⇒ `__getitem__`

set ⇒ `__setitem__`

- Punktsyntax: `get(a,i)` ⇒ `a.__getitem__(i)`
  "__getitem__" ist Methode der Klasse
- Index-Notation
  `get(a,i)` ⇒ `v = a[i]` (falls auf der rechten Seite der Zuweisung)
  `set(a,i,v)` ⇒ `a[i] = v`
- Konstruktor: `new_array` ⇒ list
  (nicht mit verketteter Liste verwechseln)

2.2.2 Stack

≙ Stapel (z.B. von Bierkästen)
nur der oberste Kasten ist leicht zugreifbar

Op	Bedeutung	Axiome
`s = new_stack()`	Konstruktor	`len(s) == 0`
`len(s)`	Abfrage der Größe	(= aktuelle Anzahl der Elemente)
`push(s,v)`	Element v am Ende anhängen (oben drauf stapeln)	`len(s') == len(s) + 1` `top(s') == v`
`top(s)`	Abfragen des obersten Elements	
`pop(s)`	Entfernen des obersten/letzten Elements	`s'= pop(s, push(s,v))` `s' == s`

in Python: die DS "list" ist auch ein Stack

- `push(s,v)` ⇒ `s.append(v)`
- `pop(s)` ⇒ `s.pop()`
- `top(s)` ⇒ `s[-1]`

Konventionen in Python: negative Indizes vom Ende gerechnet

`s[i]` (i positiv): normaler Zugriff

`s[i]` (i negativ): `s[len(s)+i]`

`s[-1]` `s[len(s)-1]`

Stackverhalten: LIFO "last in – first out"

2.2.3 Queue

≙ Warteschlange (z.B. Eisdiele)

FIFO "first in – first out"
"first come - first serve"

Op	Bedeutung	Axiome
`push(q,v)`	v am Ende anhängen (wie beim Stack)	
`first(q)`	das erste Element (gegensatz Stack: `top(s)` ⇒ letztes Element)	
`pop(q)`	entferne das erste Element (geg. Stack: das letzte El.)	
in Python: `first(q)` ⇒ `q[0]` `pop[q]` ⇒ `q.pop(0)`	Funktion von "list"	

2.2.4 Deque

$\widehat{=}$ double ended Queue $\widehat{=}$ DS gleichzeitig Stack und Queue

`pop_front()` $\widehat{=}$ erstes Element entfernen $\widehat{=}$ Queue
`pop_back()` $\widehat{=}$ letztes Element entfernen $\widehat{=}$ Stack

2.2.5 Assoziatives Array

$\widehat{=}$ Dictionary $\widehat{=}$ in Python "dict"

statt Indizes aus $\mathbb{N}_0$ (natürliche Zahlen) sind beliebige Schlüssel erlaubt.

typische Fälle:

- natürliche Zahlen, die nicht in `[0, N-1]` liegen
  z.B. Matrikelnummern
  `a[1359742]` $\Rightarrow$ "Fritz Schulze"
- strings, z.B. Namen
  `alda_noten["Fritz Schulze"]` $\Rightarrow$ 1.0

die Schlüsselworte werden automatisch (versteckt vor Programmierer) in die eigentlichen Speicherindizes umgerechnet

2.2.6 Ausblick: Prioritätswartescchlangen

`a.top(), a.pop()` greifen zu / entfernen das Element mit höchster Priorität

Stack und Queue sind Spezialfälle:
neuestes bzw. ältestes Element haben höchste Priorität

Anwendungen von Queue und Stack

- Queue: Drucken-Warteschlange
- Stack:Undo-Funktionalität in Textprogrammen

Aktion des Benutzers	undo-stack u	redo-Stack v
a_1	`u.push(a1)`	
a_2	`u.push(a2)`	
a_3	`u.push(a3)`	
undo	`undo(u.top())` $\Rightarrow a_3$	
	rückgängig	
	`u.pop()`	`v.push(u.top())`
undo	`undo(u.top())` $\Rightarrow a_2$	
	rückgängig	
	`u.pop()`	`v.push(u.top())`
redo		`do (v.top())` $\Rightarrow a_2$
		wiederherstellen
	`u.push(v.top())`	`v.pop()`
a_4	`u.push(a4)`	`v.clear()`
$\vdots$	$\vdots$	$\vdots$

3 Sortieren

Warum?

- viele Konzepte des Algorithmen-Design und -Vergleichs werden sehr anschaulich
- sortierte Daten braucht man oft in der Praxis, z.B. zum schnellen Suchen
- aber: man muss sortieren heute selten selbst implementieren, weil alle Programmiersprachen das schon anbieten

  `sort(a)`

Spielregeln:

1. Die Daten liegen in einem Array: `a`
   ⇒ Der Alg. darf aufrufen:

```
N = len(a)              # Laenge von a
v = a[i]                # Lesen von El.i
a[i] = v                # Schreiben von El.i
```

mit $i \in [0, N-1]$
⇒ Der zu sortierende Datentyp ($\widehat{=}$ Elemente des Arrays) unterstützen in Vergleichsfunktion, meist "$<$"
$a[i] < a[k] \Rightarrow$ `True oder False`

Der Vergleich muss die mathematischen Anforderungen einer *totalen Ordnung* erfüllen

Eine totale Ordnung ist antisymmetrisch, transitiv, reflexiv, total
Elemente $a, b, c, ...$ und die Relation "$\leq$": $a \leq b \rightarrow$ `t, f`

- total: man kann beliebige Elementpaare vergleichen
  $a \leq b$ liefert immer `t oder f`
  (Gegenteil: Halbordnung: manche Elemente nicht vergleichbar $a \leq b$ liefert `t oder f oder "unknown"`)
- antisymmetrisch: $a \leq b \wedge b \leq a \Rightarrow a == b$
- transitiv: $a \leq b \wedge b \leq c \Rightarrow a \leq c$
- reflexiv: (folgt aus den anderen): $a \leq a$ immer `true`

Frage: Angenommen $a \leq b$ ist definiert, aber der Sortieralgorithmus braucht $a < b$.
Kann man $a < b$ implementieren, indem man nur logische Operationen $\wedge \vee \neg$ sowie $\leq$ verwendet?

Antwort: $a < b \Leftrightarrow \neg(b \leq a)$

Hausaufgabe: Wie bekommt man $>, \geq, ==, !=$

3.1 Selection Sort

```python
def selection_sort(a):
    N = len(a)
    for i in range(N-1):        # i ist die Arrayposition, die wir sortieren wollen
        m = i                   # m ist unsere aktuelle Meinung,
                                # wo das kleinste rechts von i steht
        for k in range(i+1, N):
            if a[k] < a[m]:
                m = k           # Meinung korrigieren
                                # a[m] ist jetzt das kleinste Element rechts von i
        a[i], a[m] = a[m], a[i] # vertauschen
```

- Datenobjekte haben oft mehrere Eigenschaften, nach denen man sortieren kann.
  Studenten: Sortieren nach Alter, Alda-Noten, ...
  hier: nach Zahl oder Farbe
  sortiere jetzt nach Farbe: orange < rot< blau < schwarz

  Stabiliät der Sortierung:

  – Anfangs ist das Array nach Kriterium 1 sortiert (“Zahl”)

  – Wir sortieren nun nach Kriterium 2 (“Farbe”), aber es gibt Elemente, die dabei den gleichen Wert haben

  – Sortieralgorithmus ist *stabil*, wenn die Ordnung 1 erhalten bleibt, über Elementen mit identischem Wert 2

⇒ Selection Sort nicht stabil

3.2 Insertion Sort

ähnlich einfach wie Selection Sort, aber *stabil*

- Beobachtung: Insertion Sort ist für *kleine Arrays* (N < 30) der schnellste Sortieralgorithmus

```python
def insertion_sort(a):
    N = len(a)
    for i in range(N):
        current = a[i]
        k = i
        while k > 0:
            if current < a[k - 1]:
                a[k] = a[k - 1]
            else:
                break
```

```
            k = k - 1
    a[k] = current
```

Welche Laufzeit benötigen Selection und Insertion Sort?

- Wie misst man das unabhängig davon, ob man einen schnellen oder langsamen Computer hat, oder wie groß N ist?
- 2 Lösungen: Zähle a Anzahl der Vergleiche $a < b$, b Anzahl der Vertauschungen

i	Anzahl der Schritte in der Schleife $\widehat{=}$ Anzahl Vergleiche
0	$k \in [1, N-1] \widehat{=} N-1$ S.
1	$[2, N-1] \widehat{=} N-2$ S.
2	$\widehat{=} N-3$ S.
$\vdots$	
$N-2$	$[N-1, N-1] \widehat{=} 1$ S.

$\Rightarrow$ die totale Anzahl Vergleiche:
$T = (N-1) + (N-2) + (N-3) + ... + 2 + 1$

$\boxed{= \frac{N(N-1)}{2} \approx \frac{N^2}{2}}$

Anzahl Vertauschungen: $V = N - 1 < T$

Elementare Sortierverfahren

iteriere mit i über alle Arrayelemente

- Selection Sort: finde das kleinste Element rechts von i und bringe es auf Position i
- Insertion Sort: Finde die passende Lücke links von i, wo Element a[i] einsortiert werden muss

3.2.2 Aufwand beim Sortieren

Aufwand beim Sortieren: Anzahl der Vergleiche $a[i] < a[k]$ und/oder Anzahl der Vertauschungen $a[i], a[k] = a[k], a[i]$

- bei Selection Sort: Anzahl Vergleiche $V = \frac{N(N-1)}{2}$
- bei Insertion Sort: Unterscheide drei Fälle:
  - günstigster Fall: Array schon sortiert
  - ungünstigster Fall: Array umgekehrt sortiert (absteigend statt aufsteigend)
  - typischer Fall: Array zufällig angeordnet
- alle drei Fälle haben unterschiedliches V!
  (bei Selection Sort: V immer gleich)

```python
def insertion_sort_1(a):
    N = len(a)
    for i in range(1, N):
        k = i
        while k > 0:
            if a[k] < a[k - 1]:
                a[k - 1], a[k] = a[k], a[k - 1]
            else:
                break
            k = k - 1
```

⇒ für kleine N der schnellste Algorithmus

1. günstigster Fall: Array ist sortiert, d.h. der Vergleich (*) liefert immer sofort "False"⇒ while-Schleife hat nur 1 Iteration
   ⇒ ein Vergleich pro i ⇒ $\boxed{V = N - 1}$
2. ungünstigster Fall: Array umgekehrt sortiert ⇒ Vergleich (*) liefert immer "True" ⇒ while-Schleife muss immer bis zum Ende (k=0) durchlaufen werden ⇒ i Vergleiche für jedes i
   $V = 1 + 2 + 3 + ... + (N - 1) = \boxed{\frac{N(N-1)}{2} = V} \approx \boxed{\frac{N^2}{2}}$
3. typischer Fall: Array zufällig ⇒ im Mittel wird die while-Schleife zur Hälfte durchlaufen ⇒ $\boxed{V = \frac{N(N-1)}{4}} \approx \boxed{\frac{N^2}{4}}$

Einwurf:
swap braucht mindestens 3 Zuweisungen:
swap(a,b) : tmp = a, a = b, b = tmp
in Python: sogar 4 a,b = b,a $\widehat{=}$ t1=a, t2=b, b=t1, a=t2

Sortieren nach dem Teile-und-Herrsche-Prinzip

- elementare Alg. brauchen im typischen Fall $V = c * N^2$ Vergleiche für eine Konstante c ⇒ sie sind für große N langsam
- bessere Alg.: teilen das Sortierproblem in Unterprobleme, die getrennt sortiert und dann effizient zusammengesetzt werden.

3.3 Merge Sort

- Operation merge: Setze ein großes Array aus zwei sortierten Teilarrays zusammengesetzt

```python
def merge(l, v):  # l,r: sortierte Teilarrays
    a = []  # leeres Array fuer das Ergebnis
    i, k = 0, 0
    Nl, Nr = len(l), len(r)
    while i < Nl and k < Nr:
        if l[i] <= r[k]:
```

```
            a.append(l[i])
            i = i + 1
        else:
            a.append(r[k])
            k = k + 1

    a = a + l[i:Nl] + r[k:Nr]  # Rest von l bzw. r an a anhaengen
    return a
```

`r[k : Nr]` entspricht:

```
    while k < Nr:
        n.append(r[k])
        k = k+1
    while i < Nl:
        n.append(l[i])
        i = i+1
```

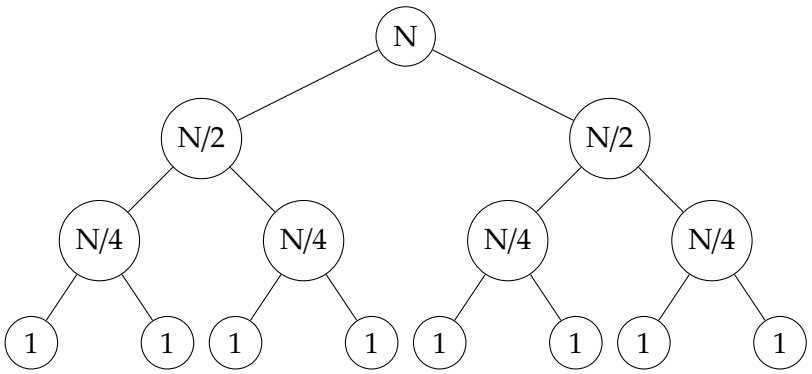
- um l und r zu sortieren, wendet man das gleiche Prinzip rekursiv auf die linke bzw. rechte Hälfte des Arrays an:

```
def merge_sort(a):
    N = len(a)
    if N <= 1:                          # leeres Array oder mit 1 Element
        return a                        # ist automatisch sortiert
    else:
        l = a[0:N//2]                   # N//2 = "floor division", rundet ab
        r = a[N//2 : N]
        l_sorted = merge_sort(l)        # teile-und-herrsche
        r_sorted = merge_sort(r)
        a_sorted = merge(l_sorted, r_sorted)
        return a_sorted
```

Laufzeit von merge sort:

- Wie tief ist der Baum, der bei der Ausführung entsteht?

Beispiel für $N = 8$



$8 = 2^3 \Rightarrow$: $\boxed{M + 1 = \text{Tiefe} = \lceil log_2 N \rceil}$

Wie viele Vergleiche braucht man pro Ebene?

→ Ebene 1: Zwei Arrays der Länge N/2

⇒ merge vergleicht immer die ersten Elemente von l und r

ungünstigster Fall: das kleinste Element ist abwechselnd links und rechts ⇒ (N-1) Vergleiche ≈ N Vergleiche

dazu kommen die Vergleiche für das Sortieren der Teilarrays l und r

$$
\begin{aligned}
\Rightarrow V(N) &= \underbrace{N}_{\approx N-1} + \underbrace{2}_{\text{2 Arrays}} * \underbrace{(N/2)}_{\text{der Länge N/2 sortieren}} \\
&= N + \underbrace{[\frac{N}{2} + 2 * V(\frac{N}{4})]}_{\text{l sortieren}} + \underbrace{[\frac{N}{2} + 2 * V(\frac{N}{4})]}_{\text{r sortieren}} \\
&= N + (\tfrac{N}{2} + \tfrac{N}{2}) + 4 * [\tfrac{N}{4} + 2 * V(\tfrac{N}{8})] \\
&= N + \underbrace{(\frac{N}{2} + \frac{N}{2})}_{N} + \underbrace{4 * (\frac{N}{4})}_{N} + \cdots \\
&= N + N + N + \cdots + N
\end{aligned}
$$

⇒ pro Zerlegungsebene N Vergleiche (bzw. Zuweisungen)

⇒ da es $\lceil log_2 N \rceil$ Ebenen gibt, ist die Gesamtzahl der Vergleiche

$\boxed{V = N * \lceil log_2 N \rceil}$ << $\frac{N^2}{4}$ für große N

günstiger Fall: Array bereits sortiert

⇒ bei merge werden zuerst alle Elemente von l gewählt, dann r an das Ergebnis-Array angehängt

⇒ man braucht nur halb so viele Vergleiche wie im ungünstigen Fall

$V = \frac{N}{2}\lceil log_2(N)\rceil = cN * \lceil log_2(N)\rceil$

unterscheidet sich nur durch Konstante c $\widehat{=}$ unwichtig

- Merge Sort ist stabil, wenn bei Gleichheit l[i] == r[k] immer das linke Element gewählt wird
- Python's list.sort()-Funktion verwendet merge sort wegen der Stabilität
  aber: hochoptimiert, d.h. häufige Spezialfälle (bereits sortiert, umgekehrt sortiert) werden abgefangen und schneller implementiert

3.4 Quick Sort

- Standard-Algorithmus für Sortieren, z.B. in vielen Implementationen von C++: std::sort() (optimierte Kombination von Quick Sort mit heap sort & insertion sort)
- Nachteil von merge sort: es braucht temporären Speicher zum Anlegen der gemergten Arrays
- besser: in-place $\widehat{=}$ sortieren erfolgt im Originalarray
  $\Rightarrow$ quick sort
- Idee: Funktion "partition": wähle Pivot-Element und sortiere es an die korrekte Stelle $\Rightarrow$ alle linken sind kleiner, alle rechten größer, aber nicht untereinander sortiert

```
def partition(a, l, r):       # l: linke Grenze des zu sortierenden Bereichs
    pivot = a[r]              # r: rechte Grenze
    i = l, k = r - 1
    while True:               # Endlosschleife -> wird unten per "break" verlassen
        while i < r and a[i] <= pivot: # suche Element > pivot
            i = i+1
        while k > l and a[k] >= pivot: # suche Element < pivot
            k = k - 1
        if i < k :
            a[i], a[k] = a[k}, a[i]    # tausche oder beende Schleife
        else:
            break
    a[r] = a[i]
    a[i] = pivot              # bringe pivot an richtige Pos.
    return i                  # pivot-Position fuer Rekursion

def quick_sort(a):
    quick_sort_impl(a, 0, len(a)-1)
    (return a)

def quick_sort_impl(a, l, r):
    if r <= l :           # a[l : r + 1] hat hoechstens ein Element -> schon sortiert
        return (None)
    k = partition(a, l, r) # k ist korrekte Position des Pivot
```

```
25      quick_sort_impl(a, l, k-1)  # rekursiv links
26      quick_sort_impl(a, k+1, r)  # rekursiv rechts
```

Laufzeit von Quick Sort

- günstiger Fall: das Pivot ist bei jedem Aufruf immer der Median des Teilarrays ($\widehat{=}$ Element in der Mitte, nach partition())
  $\Rightarrow$ die verbleibenden Teilarrays sind ungefäht gleichgroß

  Rekursionsformel: allg. Prinzip: $\underbrace{C(N)}_{\text{totale Laufzeit für Größe N}} = \underbrace{A(N)}_{\text{Laufzeit im aktuellen Teilarray}} + \underbrace{R(N)}_{\text{Laufzeit für rekursive Aufrufe}}$

  `C(quick_sort_imp(N)) = C(partition(N)) + C(qsi(links)) + C(qsi(rechts))`

  $C_g(N) \approx (N+1) + C_g(\frac{N}{2}-1) + C_g(\frac{N}{2}-1)$

  $\approx N + 2 + C_g(\frac{N}{2}) = N * \lceil log_2(N) \rceil$

- ungünstiger Fall: Pivot ist immer am Rand, d.h. partition() verändert die Position des Pivots nicht $\widehat{=}$ Array war schon sortiert
  $C_u(N) = \underbrace{(N+1)}_{\text{partition}} + \underbrace{C_u(N-1) + C_u(0)}_{\text{Rekursion}} \qquad C_u(0) = 0$
  $= (N+1) + [N + C_u(N-2)]$
  $= (N+1) + N + \Big[(N-1) + C_u(N-3)\Big]$
  $= (N+1) + N + ... + 1 = \frac{(n+2)(N+1)}{2} \approx \frac{N^2}{2}$
  $\Rightarrow$ so langsam wie selection sort $\Rightarrow$ wir müssen den ungünstigen Fall verhindern
- typischer Fall: Array ist zufällig sortiert
  $\Rightarrow$ jede Position zwischen l und r ist mit gleicher Wahrscheinlichkeit die Position des Pivot nach partition()
  $\Rightarrow$ wir verwenden den Mittelwert des Aufwands in der Rekursionsformel

$$C_t(N) = (N+1) + \underbrace{\frac{1}{N} \underbrace{\sum_{k=1}^{N}}_{\text{mögliche Pos. des Pivots}} [C_t(k-1) + C_t(N-k)]}_{=2\sum_{k=1}^{N} C_t(k-1)}$$

$$N * C_t(N) = (N+1) * N + 2\sum_{k=1}^{N} C_t(k-1)$$

$$\left.\begin{aligned} (N-1)C_t(N-1) &= N*(N-1)+2 \\ N * C_t(N) &= (N+1)*N + 2\textstyle\sum_{k=1}^{N-1} C_t(k-1) \end{aligned}\right\} -$$

$$N * C_t(N) - (N-1)C_t(N-1) = (N+1)N - N(N-1) + 2C_t(N-1)$$

$$N * C_t(N) = (N+1) * C_t(N-1) + 2N \quad \Big| : N$$

$$C_t(N) = \frac{N+1}{N} C_t(N-1) + 2 * \frac{N+1}{N+1}$$

$$(N+1)\left[\frac{1}{N} C_t(N-1) + \frac{2}{N+1}\right]$$

$C_t(N-1)$ sukzessive expandieren

$$C_t(N) = (N+1)\left[\frac{1}{\cancel{N}} \frac{\cancel{N}}{N-1} C_t(N-2) + 2\frac{N}{N} + \frac{2}{N+1}\right]$$

$$= \frac{N+1}{N-1} C_t(N-2) + 2(N+1)\left[\frac{1}{N} + \frac{1}{N+1}\right]$$

$$= \frac{N+1}{N-2} C_t(N-3) + 2(N+1)\left[\frac{1}{N} + \frac{1}{N+1} + \frac{1}{N+1}\right]$$

$$= \frac{N+1}{1} \underbrace{C_t(0)}_{=0} + 2(N+1) \underbrace{\left[\frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{N+1}\right]}_{< \frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\cdots+\frac{1}{N+1}}$$

$$\leq 2(N+1) \underbrace{\sum_{k=1}^{N+1} \frac{1}{k}}_{\text{„harmonische Reihe“}} \qquad \sum_{k=1}^{N+1} \frac{1}{k} \approx \int_1^{N+1} \frac{1}{k} dk = ln(N+1)$$

$$\boxed{C_t(N) \leq 2(N+1)\, ln(N+1)} \boxed{\approx 1.38N * log_2(N)}$$

typisch: schneller als merge sort

- wie verhindert man, dass der ungünstige Fall eintritt?
  - viele komplizierte Ideen
  - einfache Idee: wähle das Pivot-Element (bzw. seinen Index) zufällig
    statt: `pivot = a[r] # immer das rechte Element`
    schreibe:

```
1  p = random.randint(l,r) # zufaelliges p aus [l,r], Gleichverteilung
2                          # randint: Zufallszahlenmodul in Python
3  a[p],a[r] = a[r],a[p]   # Pivot jetzt rechts
4  pivot = a[r]            # Rest von partition() wie gehabt
```

    $\Rightarrow$ da die Aufteilung für die Rekursion nur von der finalen Pivot-Position abhängt, garantiert dies, dass der typische Fall zutrifft:
    typischer Fall : Array zufällig sortiert, insbes. a[r] ein zufälliges Element
    Zufallswahl: bringe zufälliges Element nach $a[r] \Rightarrow$ wie typischer Fall
    heutige Zufallszahlengeneratoren sind sehr schnell
    (zur Zeit der Erfindung von quick sort war das noch anders)
  - für kleine Array verwendet man insertion_sort (auch in der Rekursoin)
  - quick sort ist nicht stabil

4 Korrektheit

Ein Algorithmus besteht aus zwei Teilen:

- Spezifikation: Was soll der Algorithmus tun? (Vor- und Nachbedingungen)
- Implementation: Wie geht der Algorithmus vor?

⇒ Validierung: Prüfe, ob die Spezifikation das beschreibt, was wir wirklich wollen.
⇒ Verifikation: Prüfe, ob die Implementation die Spezifikation erfüllt.

Bemerkungen:

1. Wenn ein Algorithmus nicht korrekt ist, sind alle anderen Qualitäten (Effizienz, Lesbarkeit, ...) irrelevant.
2. Manche Algorithmen liefern nie oder fast nie exakte Ergebnisse, z.B. Numerik:
   - Mathematik: reelle Zahlen (unendlich viele Bits)
   - Informatik: Gleitkommazahlen (endlich viele Bits)

   ⇒ dann muss die Spezifikation die gewünschte Genauigkeit angeben.
   bei float64 typisch: relativer Fehler zw. $10^{-10} \dots 10^{-15}$
3. Manche Alg. liefern nie zweimal das gleiche Ergebnis (z.B. Training neuronaler Netze) oder nur mit sehr hoher Wahrscheinlichkeit das richtige Ergebnis (randomisierte Primzahltests)
   → typisch für Alg., die Zufallszahlen verwenden
   ⇒ schwierig zu testen

4.1 3 Wege zum korrekten Code

1. Programmiersprachen finden bestimmte Fehler automatisch:
   - Syntaxprüfung:

   ```
   if a = 2:   # sollte ein "==" sein
       a += 1  # -> Python: Syntax Error
   ```

   in C: es ist erlaubt, aber es gibt eine Warnung
   - Typprüfung: jede Datenstruktur in der Programmiersprache hat einen Typ
   ⇒ Programmiersprache weist Operation zurück, wenn die Typen nicht passen

   ```
   a = 3
   b = None
   a + b       # -> Python: TypeError
   ```

- in C/C++: statische Typprüfung: der Fehler wird bereits bei der Kompilierung signalisiert
- Python: dynamische Typprüfung: Der Fehler wird erst bei der Ausführung signalisiert

- Prüfen der Vorbedingung:
  - in manchen Programmiersprachen (z.B. Eiffel) kann man Vorbedingungen für jede Funktion explizit formulieren ⇒ sie werden bei jedem Aufruf automatisch geprüft
  - in Python: händisch prüfen (mit `if(Vorbedingung == False): raise exception(``Message'')`)
    (ähnlich in den meisten anderen Sprachen)

2. formaler Korrektheitsbeweis
   ⇒ Bugs werden mathematisch ausgeschlossen
   ⇒ sehr aufwändig, nur bei sicherheitskritischer Software

3. Software-Tests: heute die wichtigste Methode

Alg.	Test	
korrekt	korrekt	⇒ alles okay
bug	korrekt	⇒ ok: Test findet das Bug
korrekt	bug	⇒ ärgerlich, aber leicht zu reparieren
bug	bug/nicht mächtig genug	⇒ Test versagt

   sehr gute Tests garantieren zwar keine Bugfreiheit, aber können die Wahrscheinlichkeit dafür sehr hoch machen.

4.2 Wie testet man in Python?

Regel: Tests werden in Testdateien gesammelt und sind **Teil des Projekts**

⇒ werden als **Regressionstests** nach jeder Programmänderung erneut ausgeführt

⇒ dadurch erkennt man sofort, wenn eine Änderung etwas zerstört

Testframeworks: doctest, unit test (eingebaute Module)
pytest, nose (muss man installieren)
unterstützen das systematische Schreiben und Ausführen von Tests

Beispiel: $x = \sqrt{y}$ berechne Wurzel

- Prinzip: iterativer Algorithmus
  0. initial guess $x^{(o)}$, so dass $(x^{(o)})^2 \approx y$
  1. while maximum number of iteration must not reached:
     a) compute new guess from old one
     $x^{(t)} = f(x^{(t-1)}, y)$

b) if $x^{(t)}$ is good enough:
return $x^{(t)}$

return $x^{(T_{max})}$ and/or error message warning

- Babylonischer Alg:

$$x^{(t)} = \frac{x^{(t-1)} + y/x^{(t-1)}}{2}$$

Begründung: Spezialfall von Newtons Algorithmus zur Berechnung von Nullstellen

Aufgabe: finde x, so dass $g(x) = 0$

Speziell: $g(x) = y - x^2 = 0$ falls $x = \sqrt{y}$

Iteration:

$$x^{(t)} = x^{(t-1)} - \frac{g(x^{(t-1)})}{g'(x^{(t-1)})}$$

$$\left[g'(x) = -2x\right] = x^{(t-1)} - \frac{y - (x^{(t-1)})^2}{-2x^{(t-1)}}$$

$$= \frac{x^{(t-1)} + y/x^{(t-1)}}{2}$$

```python
import pytest
import doctest

def mysqrt(y):
    '''
    :param y: the value to take the square root of
    :return: the square root of y

    Example:
    >>> mysqrt(9)
    3.0

    The argument must be non-negative:
    >>> mysqrt(-1)
    Traceback (most recent call last):
    ...
    Value Error: mysqrt(): argument must be non-negative

    '''
    #docstring, sollte jede Python-Funktion haben
    if y < 0:
        raise ValueError("mysqrt(): argument must be non-negative.")
    x = y / 2           #nicht mehr floor division
```

Ducktyping: Objekt wird nicht aufgrund des Datentyps sondern aufgrund der möglichen auf ihm anwendbaren Operationen behandelt

```
    #in Python 2 war normale Division 'y/2' eine floor division,
    #wenn y vom Typ 'int' war.
    #in Python 3 liefert 'y/2' immer float, auch wenn y 'int' ist
    # => Fehlerquelle beseitigt, aber schwierige Portierung
    while abs(x**2 - y) > 1e-15*x**2:
        x = (x+y / x) / 2
    return x


def test_mysqrt():      #Alle Testfunktionen muessen mit test_ anfangen
    assert mysqrt(0) == 0
    with pytest.raises(ValueError):
        mysqrt(-1)
    assert mysqrt(9) == 3
    assert mysqrt(1) == 1
    assert mysqrt(4) == 2
    assert mysqrt(1.21) == pytest.approx(1.1)


#Ausfuehrung mit pytest
#man kann doctest mit pytest ausfuehren, indem man --doctest-modules beim Ausfuehren
↪  hinzufuegt
#erst den Test schreiben und dann die Funktion: test-driven development
```

4.2.1 3 Arten von Tests

- black box: die Implementation ist dem Tester unbekannt
  ⇒ Domainexperten überlegen, welche Systemeigenschaften erfüllt sein müssen
- gray box: der Tester kennt die Implementation und kann den Test geziehlt auswählen, z.B. Randworttest
- white box: der Tester kann die Implementation modifizieren
  - explizite Tests für Nachbedingungen einfügen
    ⇒ diese Tests werden in der Releaseversion deaktiviert
    z.B. assert(test):
    - in Debugcode: führe den Test aus
    - in Releasecode: ignoriere Test

    einige Tests bleiben im Releasecode ⇒ Entwickler über Abstürze informieren
  - Code coverage messen: automatisch überprüfen, welche Teile des Quellcodes während des Tests ausgeführt werden : > 80%, möglichst 100%
  - absichtliche Bugs einbauen: teste die Mächtigkeit der Tests
    ⇒ Testprogramm verbessern “fault injection”

4.2.2 Wie definiert man gute Tests?

- Prinzip des Regression-Testing: implementierte Tests werden nicht gelöscht, sondern nach jeder Code-Änderung erneut ausgeführt
- Prinzip der Reproduktion von Bugs: Bug report ⇒ implementiere zuerst einen neuen Test, der Bug reproduziert ⇒ danach korrigieren des Codes
- Prinzip der äquivalenten Eingaben: meist gilt: für ähnliche Eingaben verhält sich der Algorithmus gleich (d.h. die gleichen Codeteile, z.B. if-Zweige, for-Iterationen, while-Iterationen, Unterprogrammaufrufe werden ausgeführt)

  ⇒ teste wenige repräsentative Eingaben einer Äquivalenzklasse

  ⇒ teste Repräsentanten von allen Aquivalenzklassen ⇒ Code Coverage

  [für einfache, aber kritische Codeteile: vollständiger Test aller Eingaben]
- Prinzip des Randwerttests: Eingaben an der Grenze der Äquivalenzklassen haben besonders häufig Bugs ⇒ teste diese besonders sorgfältig
  Bsp: `mysqrt()`:
  - Äquivalenzklassen:
    - $y < 0$
    - $0 \leq y < 1$
    - $1 \leq y$
    - $y \in \text{int} \cap x \in \text{int}, y \in \text{int} \cap x \in \text{real}, y \in \text{real} \cap x \in \text{real}$
  - Randwerte: y = 0, y = 1
- Prinzip der „beliebten Fehler“:
  - off-by-one: eine Variable (oft: Schleifen- oder Arrayindex) liegt um 1 daneben: z.B.
    - falscher Vergleich: if $i < k$: statt if $i <= k$:
    - falsche Indizes: a[i] < a[i+1] statt a[i-1] < a[i]
  - Rundungsfehler bei Gleitkommazahlen: ($sin(\pi) \neq 0$, mysqrt(2)**2 ≠ 2
    ⇒ numerische Analyse: Feld, dass diese Fehler systematisch untersucht
  - Spezialfall: loss of precision
    die Differenz von zwei fast gleichen Gleitkommazahlen hat viel weniger gültige Stellen als die Originalzahlen

    ```
      100.00001
    -
      100.00002
    ---------------
    -   0.00001
    ```

    Bsp: p-q-Formel für quadratische Gleichungen:

    $$x_{1,2} = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$$

Trick: ersetze Wurzel durch hypot$(a, b) = \sqrt{a^2 + b^2}$ (hypot → Hypothenuse)

if $abs(a) > abs(b)$:

return $a\sqrt{1 + \frac{b^2}{a^2}}$

else :

return $b\sqrt{\frac{a^2}{b^2} + 1}$

Trick 2: verwende mehr Bits (double(64-bit) statt float(32-bit))

Bsp.: (Hausaufgabe) Algorithmus von Archimedes zur Berechnung von π
(Resultat von Archimedes: $\pi \approx \frac{22}{7}$

- Generieren von Testdaten $\widehat{=}$ vorberechnetes korrektes Ergebnis, mit dem man den Output vergleichen kann
  - händisch für einfache Eingaben berechnen, z.B. Sortieren für N klein
  - verwende alternatives Verfahren: langsamen, aber einfachen Algorithmus oder anderes Programm
  - teste die inverse Funktion $y = sqrt(x) \Rightarrow y * y = x$
  - teste nicht das ganze Ergebnis, sondern seine Eigenschaften (Nachbedingung oder andere besondere Eigenschaften, die bei Bugs verletzt sein können)

4.3 Korrektheitsbeweise

- auf abstrakter Ebene (Pseudocode): beweise, dass der neue Algorithmus das Ziel erreicht, ⇒ wissenschaftl. Literatur über neue Alg., Lehrbücher, besonders Cormen, et.al.
- auf Implementationsebene: beweise, dass die Implementation keinen Bug hat
  - autonome U-Bahn 14 in Paris
  - Flugzeugbetriebssystem INTEGRITY 178B
  - Sicherheitsmerkmale von Chipkarten, allg. von Verschlüsselung

Beispiel für Korrektheitsbeweis des Algorithmus Selection Sort

- Prinzip der Induktion:
  1. Induktionsanfang: beweise, dass best. Invarianten am Anfang des Alg. gelten
  2. Induktionsschritt: beweise, dass die Invarianten nach Iteration t gelten, wenn sie nach Iteration (t-1) wahr waren
  3. Induktionsschluss: beweise, dass aus der Gültigkeit der Invarianten nach der letzten Iteration die Gültigkeit der Nachbedingung folgt ⇒ Algorithmus korrekt
- Beispiel:

Nachbedingung: für alle i < k gilt: a[i] <= a[k]

Invarianten (i) nach Iteration i ist das linke Teilarray a[0 : i+1] sortiert

(ii) alle Elemente im rechten Teilarray sind mindestens so groß wie im linken Teilarray: max(a[0:i+1]) = a[i] <= min(a[i+1:N])

Konventionen: leeres Teilarray ist sortiert und max([]) = min([]) = $-\infty$

```
1  def selection_sort(a):
2      N = len(a)                      #(1)
3      for i in range (N-1):
4          m = i                       #(2)
5          for k in range(i+1, N):     #(2)
6              if a[k] < a[m]:         #(2)
7                  m = k               #(2)
8          a[i], a[m] = a[m], a[i]     #(2)
```

1. Induktionsanfang: Invariante:

   (i): das Array links von i ist sortiert: $\widehat{=}$ [] ist per Definition sortiert

   (ii): $\underbrace{max(\text{links von i})}_{-\infty} \quad \leq \quad \underbrace{min(\text{rechts von i})}_{\text{Fall 1: } N=0 \rightarrow -\infty | \text{Fall 2: } N>0 \rightarrow \text{ ganze Zahl}}$

2. Schritt: in Iteration i: nimm an, dass a[0:i] sortiert ist (i) und max(a[0:i]) ≤ min (a[i:N]) (ii) beweise, dass das am Ende der Iteration für i+1 gilt
   - nach innerer Schleife wissen wir: a[m] = min(a[i:N])
   - nach tauschen wissen wir: a[i] = min(a[i:N]) (*)
   - Aus Induktionsannahme (ii) wissen wir: max(a[0:i]) ≤ a[i]
     ⇒ a[0:i+1] ist sortiert ⇒ Invariante (i) danach
   - aus (*) folgt außerdem: a[i] ≤ min([i+1:N]) ⇒ max(a[0:i+1]) ≤ min(a[i+1:N]), max([a[0:i])

3. Induktionsende: nach der Schleife `for i in range(N)`: hat i formal den Wert N
   (in C/C++ hat i tatsächlich diesen Wert: int i = 0; for (i = 0; i < N-1; i++) //hier hat i den Wert N
   ⇒ das Teilarray ist nach der letzten Iteration: a[0:N] == a, also das gesamte Array.
   wegen Invariante (i) ist das linnke Teilarray immer sortiert
   ⇒ das ganze Array ist sortiert □

5 Effizienz

- eine der zentralen Fragen bei Algorithmen
- Laufzeitmessung („wall clock time“): für den Endnutzer relevant
- Komplexität (auf abstrakt algorithmischer Ebene): für Programmierer & Theoretiker

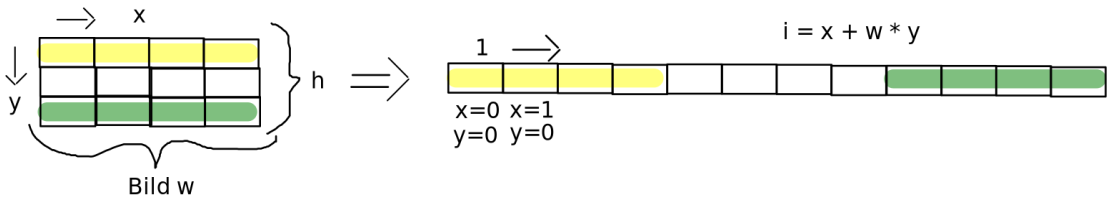
5.1 Laufzeit

„der Algorithmus kommt rechtzeitig zum Ergebnis“

- bei Steuerprogrammen (Maschinen, Fahrzeuge, Flugzeuge):
  - mind. < 1/1000s Antwortzeit muss garantiert werden (nicht im Mittel, sondern immer) „Echtzeit-Anforderung“
  - Video: 1/25s pro Bild für Dekompression
  - Interaktion: nach 1/2 s muss etwas passieren, sonst klickt der Benutzer nochmals, zumindest Fortschrittsbalken
  - Wettervorhersage: sollte fertig sein, bevor das reale Wetter passiert
- Messung z.B. mit timeit-Modul (Python) Google benchmark (C/C++)
- wie erreicht man eine schnelle Laufzeit (außer: besseren Alg. wählen)
  - in Python (langsame Sprache): in schnellere Sprachen auslagern (C, C++, Fortran)
    - C-API (viele eingebaute Python-Module)
    - Cython: schreibe (mit Python-Syntax ⇒ wird automatisch kompiliert und eingebunden)
    - pybind11 (Vorgänger: boost.python): Glue code generieren C/C++ (Glue code für Kommunikation zwischen Programmiersprachen notwendig)
- in kompilierten Sprachen (C, C++, Fortran) oder Sprachen mit Just-in-time Compiler (Java) kann man durch gute Code-Struktur eine Beschleunigung von 2x bis 10x erreichen.
  - moderne Sprachen haben „Optimierer“, der den Code automatisch besser strukturiert
  - Beispiele: „common subexpression elimination“: Doppelberechnungen vermeiden

0. $\underbrace{x_1 = -\frac{p}{2} + \sqrt{\frac{p^2}{4} - q} \quad x_2 = -\frac{p}{2} - \sqrt{\frac{p^2}{4} - q}}_{\text{nicht doppelt berechnen}}$
1. $d = \sqrt{\frac{p^2}{4} - q}, \quad x_1 = -\frac{p}{2} + d, \quad x_2 = -\frac{p}{2} - d$
2. $p2 = -\frac{p}{2}, \quad d = \sqrt{p2 * *2 - q}, \quad x_1 = p2 + d, \quad x_2 = p2 - d$

– „loop invariant elimination“: Ausdrücke, die in jeder Iteration dasselbe Ergebnis haben, aus der Schleife ziehen: Bsp.: Umrechnung von 2D Bildkoordinaten in 1D Speicherindizes




```python
        for y in range(h):
            for x in range(w):
                data[x+w*y] = 0 #Initialisierung
        #Optimierung:
        for y in range(h):
            wy = w * y
            for x in range(w):
                data[x+wy]=0
```

Faustregel: „Vereinfachung der inneren Schleife“(wird am meisten ausgeführt)
Ausnutzen der Hardware:

– Prozessor Cache: Kommunikation zwischen CPU und RAM ist langsam

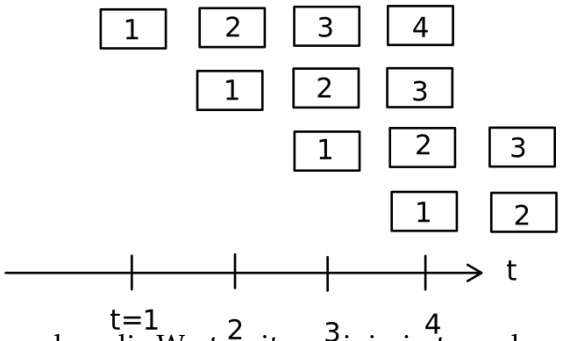
⇒ naiver Code: CPU wartet häufig auf RAM-Zugriff

⇒ Cache Zugriffe sind schnell ⇒ sorge dafür, dass benötigte Daten bereits im Cache sind
$\widehat{=}$ Daten, die im Speicher in der Nähe der bereits bearbeiteten Daten liegen „cache locality“

⇒ strukturiere Schleifen so, dass in der inneren Schleife auf aufeinanderfolgende Daten zugegriffen wird

– Prozessor Pipeline: jeder Befehl besteht aus Phasen, z.B.

1. Dekodieren des Befehls
2. Beschaffen der Inputdaten
3. Ausführen des Befehls
4. Abspeichern des Ergebnisses



Moderne Optimierer ordnen die Befehle so um, dass die Wartezeiten minimiert werden
aber:

- wenn ein Befehl auf Daten wartet, müssen die Nachfolgenden auch warten
- if-Anweisungen: je nach Ergebnis ist der nächste Befehl anders

⇒ moderne Prozessoren „spekulative Ausführung“

⇒ komplexe if-Verschachtelungen vermeiden (auch viel lesbarer)

- unnötige Typkonvertierungen vermeiden: int ⇔ double

5.2 Komplexität

- beschreibt den Aufwand eines Algorithmus auf abstrakter Ebene
- macht Algorithmen vergleichbar, unabhängig von Implementation und Hardware

Idee: beschreibe Anzahl der benötigten Schritte als Funktion der Problemgröße N

$f(N)$ (kompliziert) und vereinfache sie zu einer Näherungsfunktion $g(N)$ (einfach), so dass das essentielle Verhalten von $f(N)$ und $g(N)$ gleich ist.

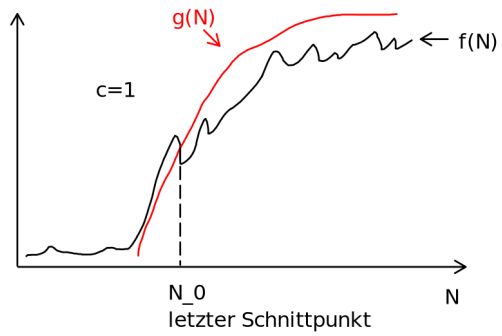
Anschaulich: behalte dominierende Terme $\widehat{=}$ die am schnellsten wachsen

Mathematisch: Landau-Symbole, „O-Notation“

$$f(N) \in O(g(N)) \qquad O(g(N)) = \text{'Komplexitätsklasse'}$$

Definition: Es gibt ein N_0 (Mindestproblemgröße) und eine Konstante c, sodass gilt:

$$\forall N \geq N_0 : f(N) \leq cg(N)$$



$$O(g(N)) := \left\{ f(N), \text{ so dass } \exists N_0, c \text{ mit } f(N) \leq c * g(N) \text{ für } N \geq N_0 \right\}$$

Eigenschaften:

- transitiv: $f(N) \in O(g(N)) \wedge g(N) \in O(h(N)) \Rightarrow f(N) \in O(h(N))$
- additiv: $f(N) \in O(h(N)) \wedge g(N) \in O(h(N)) \Rightarrow f(N) + g(N) \in O(h(N))$
- skalare Multiplikation: $f(N) \in O(g(N)) \Rightarrow c * f(N) \in O(g(N))$

$\Rightarrow$ für Monome und Polynome gilt:

$$x^k \in O(x^{k+j}) \qquad a_0 + a_1 x^1 + a_2 x^2 + \cdots + a_k x^k \in O(x^k)$$

für alle $j \geq 0$
Logarithmus: die Basis ist egal, d.h. $log_a(x) \in O(log_b(x))$ für alle $a, b > 1$

$$\text{weil} \qquad log_a(x) = \frac{log_b(x)}{log_b(a)} \Rightarrow c = \frac{1}{log_b(a)}$$

Aus Multiplikation folgt auch $\quad c * O(f(N)) \in O(f(N))$
$\qquad\qquad\qquad\qquad\qquad\qquad\quad O(c * f(N)) \in O(f(N))$

mehrere O's verschwinden: $O(O(f(N)) \in O(f(N))$

Spezialfall von Polynom: additive Konstanten dominieren nie
$\Rightarrow$ können weggelassen werden: $O(f(N)) + p \in O(f(N))$

5.3 Landau-Symbole, O-Notation

groß-O: $f(N)$ „$\leq$“ $g(N) : \exists N_0, c$, so dass $\forall N \geq N_0$ gilt : $f(N) \leq c * g(N)$
$\Leftrightarrow f(N) \in O(g(N))$

klein-o: $f(N)$ „$<$“ $g(N)$: für alle $c > 0 : \exists N_0$, so dass $\forall N \geq N_0$ gilt : $f(N) < c * g(N)$
$\Leftrightarrow f(N) \in o(g(N))$

Ω: $f(N)$ „$\geq$“ $g(N) : \exists N_0, c$, so dass $\forall N \geq N_0$ gilt : $c * f(N) \geq g(N)$
$\Leftrightarrow f(N) \in \Omega(g(N))$

ω: $f(N)$ „$>$“ $g(N)$: für alle $\infty > c > 0 : \exists N_0$, so dass $\forall N \geq N_0$ gilt : $c * f(N) > g(N)$
$\Leftrightarrow f(N) \in \omega(g(N))$

Θ: $f(N)$ „$=$“ $g(N) : \exists N_0, c_1, c_2$, so dass $\forall N \geq N_0$ gilt : $c_1 * g(N) \leq f(N) \leq c_2 * g(N)$
$\Leftrightarrow f(N) \in \Theta(g(N)) \Leftrightarrow f(N) \in O(g(N)) \wedge f(N) \in \Omega(g(N))$

$\Rightarrow$ wenn $f(N) \in o(g(N)) \Rightarrow f(N) \in O(g(N))$
Beispiele: $N \in o(N^2) \qquad \Rightarrow N \in O(N^2)$
aber: $N \notin o(N) \qquad$, aber $N \in O(N)$

Anwendung auf Sortieren:

Anzahl der Vergleiche bei :

- insertion sort:
  - typischer Fall: $V(N) = \frac{N(N-1)}{4} = \frac{N^2}{4} - \frac{N}{4} \in O(N^2)$
  - schlechter Fall: $V(N) = \frac{N(N-1)}{2} = \frac{N^2}{2} - \frac{N}{2} \in O(N^2)$
- quick sort:

- typischer Fall: $V(N) = 2(N+1)ln(N+1) \in O(N * logN)$
- schlechter Fall: $V(N) = V(N) = \frac{(N+1)(N+2)}{2} \in O(N^2)$

Rechenregeln

siehe vorige VL (oder Skript)

Regeln zur Anwendung auf Algorithmen

- Sequenzregel $\widehat{=}$ Nacheinander-Ausführung von Befehlen im Algorithmus
  anschauliche Bedeutung: der langsamste Befehl bestimmt die Komplexität

  $$\left.\begin{array}{ll} \text{Befehl 1} & O(f(N)) \\ \text{Befehl 2} & O(g(N)) \end{array}\right\} O(f(N)) + O(g(N)) \begin{cases} \in O(f(N)) & \text{wenn } g(N) \in O(f(N)) \\ \in O(g(N)) & \text{wenn } f(N) \in O(g(N)) \end{cases}$$
  $$\widehat{=} \max(O(f(N)), O(g(N)))$$

- Schachtelungsregel $\widehat{=}$ Ausführung eines Befehls (oder einer Sequenz) in einer Schleife
  anschauliche Bedeutung: Komplexität erhöht sich um die Anzahl der Durchläufe

  $$\left.\begin{array}{lll} \text{for k in range(N) :} & O(N) & O(f(N)) \\ \text{Befehl} & O(g(N)) & \end{array}\right\} O(N * g(N)) \quad O(f(N) * g(N))$$

- Wie berechnet man die Komplexität?
  - vollständige Induktion:
    1. Induktionsanfang: suche N_0 und c, so dass $f(N_0) \leq c * g(N_0)$
    2. Induktionsschritt: falls $f(N) \leq c * g(N)$
       $$\Rightarrow f(N+1) \leq c * g(N+1)$$
  - Grenzwertbildung $n \to \infty$ ($\widehat{=}$ alternative Definition von O, o, etc.)

    $f(N) \in O(g(N)) \Leftrightarrow \lim_{N\to\infty} \frac{f(N)}{c*g(N)} = 1$ für ein geeignetes $c \Leftrightarrow \lim_{N\to\infty} \frac{f(N)}{g(N)} = c$

    $f(N) \in o(g(N)) \Leftrightarrow \lim_{N\to\infty} \frac{f(N)}{g(N)} = 0$

Beispiel: ist log(N) $\in O(N)$, Grenzwertmethode:

$$\lim_{N\to\infty} \frac{lnN}{N} = \frac{\infty}{\infty} \text{ ??}$$

Regel von l'Hospital: $\lim_{x\to x_0} \frac{f(x)}{g(x)} = \lim_{x\to x_0} \frac{f'(x)}{g'(x)}$ falls $\frac{f(x_0)}{g(x_0)}$ unbestimmt

$$f(N) = ln(N) \Rightarrow f'(N) = \frac{1}{N} \qquad g(N) = N \Rightarrow g'(N) = 1$$

$$\lim_{N\to\infty} \frac{lnN}{N} = \lim_{N\to\infty} \frac{\frac{1}{N}}{1} = \lim_{N\to\infty} \frac{1}{N} = \frac{''1''}{\infty} = 0$$
$$\Rightarrow lnN \in o(N)$$

5.4 Analyse von Algorithmen für den gleitenden Mittelwert

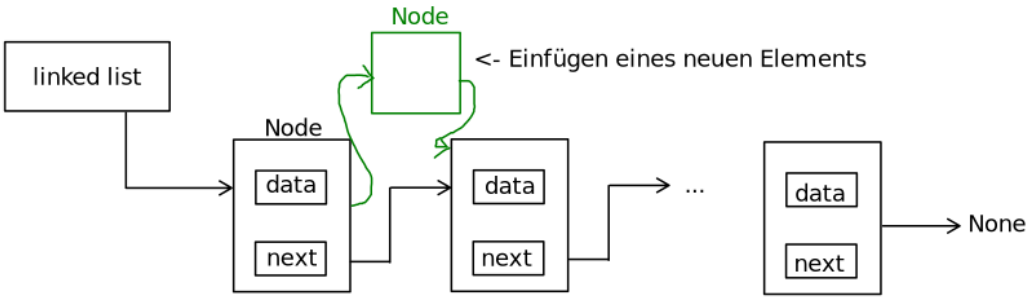
```
1  def running_mean(a, k):              #N=len(a)
2      r = [0] * len(a)                 #O(N)
3      if k > len(a):                   #O(1)
4          raise RuntimeError("k too large")
5      for j in range (k-1, len(a)):    #O(N-k+1) = O(N)
6          for i in range(j-k+1, j+1): #O(k)
7              r[j] += a[i]             #O(1) + O(1) + O(1) = O(1)
8          r[j] /= float(k)             #O(1) + O(1) + O(1) = O(1)
9      return r
```

Schachtelung: $O(k) * O(1) = O(k)$
Schachtelung: $O(N) * O(k) = O(N * k)$
Sequenz: $O(N) + O(1) + O(N * k) + O(1) = O(N * k)$

Verwende ungünstige Datenstruktur für a:
verkettete Liste: a[i] → get_element(a,i) O(i)

Verkettete Liste

für jedes Element: Node-Datenstruktur, enthält das Datenelement und einen zeiger/Referenz auf den nächsten Node



Zugriff auf Element i: Durchlaufen der Kette bis Node i $\in O(i)$ Schritte

Zugriff auf Element i: Durchlaufen der Kette bis Node i $\in O(i)$ Schritte

Optimierte Version des gleitenden Mittels:

```
1  def running_mean_2(a, k):            #N=len(a)
2      r = [0] * len(a)                 #O(N)
3      if k > len(a): ...               #O(1)
4      for i in range(k):               #O(k)
5          r([k-1] += a[i]              #O(1) -> O(k*1)
6      for j in range(k, len(a)):       #O(N)
7          r[j] = r[j-1] - a[j-k] + a[j] #O(1) -> O(N)
```

```
8       for j in range(k-1, len(a)):        #O(N)
9           r[j] /= float(k)                #O(1) -> O(N)
10      return r                            #Sequenz: maximum-> O(N) (statt O(N*k))
```

Rechenregel: $a_0 + a_1 * N + a_2 * N^2 + \cdots + a_k * N^k \in O(N^k)$

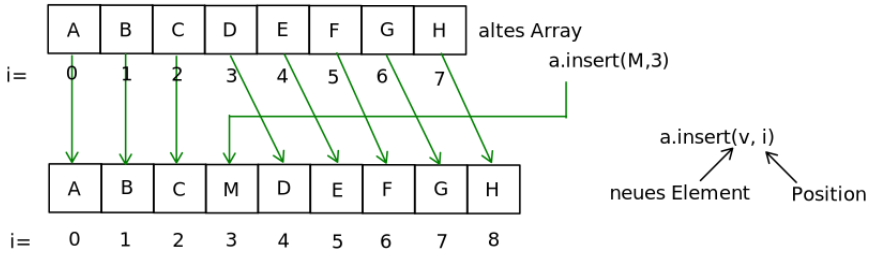
$$O(N - k) = O(N)$$

5.5 Amortisierte Komplexität

- falls dieselbe Operation manchmal schnell und manchmal langsam ist:
  - wie ist das Verhalten im Mittel?
  - Verteilt („amortisiert “) sich der Aufwand von wenigen langsamen Aufrufen über viele schnelle Aufrufe?

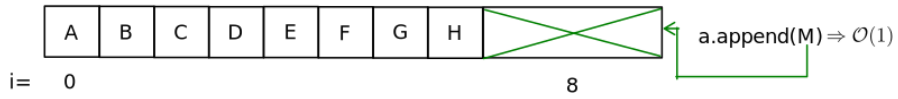
Anwendung: **dynamisches Array**

- **Problem**: Einfügung neuer Elemente in ein statisches Array teuer:

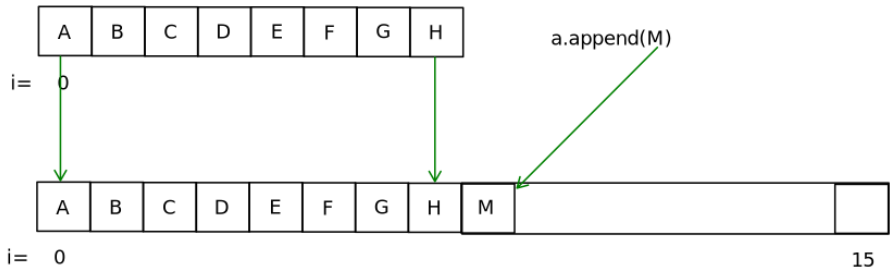


1. neues Array mit einer zusätzlichen Speicherstelle allozieren
2. die alten Daten kopieren $\Rightarrow O(N)$ teuer!
3. das neue Element einfügen

- **Lösung**: oft genügt es, wenn neue Elemente immer am Ende angehängt werden
- **Trick**: man alloziert **mehr** Speicher als man aktuell benötigt (z.B. 2x)
- **Fall 1**: Array hat noch unbenutzten Speicher:



- **Fall 2**: Array ist voll $\Rightarrow$ kopiere in ein neues Array mit **doppelter** Größe
  (nicht: in ein neues Array mit einem zusätzlichen Element, wie in Beispiel 1)



1. alloziere Array der doppelten Größe
2. kopiere die alten Daten $O(N)$
3. kopiere neues Element $O(1)$
   $\Rightarrow O(N)$

$\Rightarrow$ jetzt sind noch N-1 Speicherstellen frei (N-1 $\widehat{=}$ alte Arraygröße)
$\Rightarrow$ jetzt bekommen wir (N-1)-mal Fall 1 mot $O(1)$ $\Rightarrow$ selten $O(N)$, häufig $O(1)$$\Rightarrow$ Was bedeutet das im Durchschnitt?

- Was ist die amortisierte Komplexität des „Verdopplungs-Trick“?
- Accounting-Methode: definieren „Guthaben“, das wir während des billigen Anfügens ansparen und bei teuren Operationen verbrauchen
  $\varphi_i = \text{size}_i - \text{capacity}_i \leq 0$ weil size $\in$ capacity

Kosten einer Einfügung $c_i = \tilde{c}_i + \varphi_i - \varphi_{i-1}$

- $c_i \rightarrow$ tatsächliche Kosten der Einfügung
- $\varphi_i - \varphi_{i-1} \rightarrow$ verbrauchtes Guthaben
- **Fall 1:** vor der i-ten Einfügung gilt $\text{size}_{i-1} < \text{capacity}_{i-1} \Rightarrow$ noch Platz

  $\Rightarrow \tilde{c}_i = 1$ für das Kopieren des neuen Elements, $\text{capacity}_i = \text{capacity}_{i-1}$

  $c_i = 1 + (\underbrace{\text{size}_i}_{\text{size}_{i-1}+1} - \text{capacity}_i) - (\text{size}_{i-1} - \text{capacity}_{i-1}) = 2$
- **Fall 2:** vor der i-ten Einfügung gilt: $\text{size}_{i-1} = \text{capacity}_{i-1} \Rightarrow$ Array voll
  $\text{capacity}_i = 2 * \text{capacity}_{i-1}$

  $\Rightarrow \tilde{c}_i = \underbrace{\text{size}_{i-1}}_{\text{alte Elemente in den neuen Speicher kopieren}} + 1 \leftarrow$ neues Element kopieren

  $c_i = \text{size}_{i-1} + 1 + (\underbrace{\text{size}_i}_{\text{size}_{i-1}+1} - \underbrace{\text{capacity}_i}_{2*\text{capacity}_{i-1}=2*\text{size}_{i-1}}) - (\text{size}_{i-1} - \underbrace{\text{capacity}_{i-1}}_{\text{size}_{i-1}}) = 2$

  in beiden Fällen: $c_i = 2 \Rightarrow$ amortisierte Komplexität $O(1)$

Array	size	capacity	aktuelle Kosten $\tilde{c}_i$	totale Kosten	Durchschnittskosten	φ_i	$c_i = \tilde{c}_i + \varphi_i - \varphi_{i-}$
`[None]`	0	1				-1	
`[a]`	1	1	1	1	1	0	2
`[a,b]`	2	2	1+1	3	3/2	0	2
`[a,b,c, None]`	3	4	2+1	6	6/3	-1	2
`[a,b,c,d]`	4	4	0+1	7	7/4	0	2
`[a,b,c,d,e,None,None,None]`	5	8	4+1	12	12/5	-3	2
`[a,b,c,d,e,f,None,None]`						-2	2
`[a,b,c,d,e,f,g,None]`						-1	2

6 Suchen

Viele Arten von Suchen:

- **Schlüsselsuche**: suche Elemente, deren Schlüssel einem vorgegebenen Schlüssel entspricht
- Bereichssuche: ein Intervall von gültigen Werten wird gesucht
- Nachbarschaftssuche / Ähnlichkeitssuche: Elemente, die einem gegebenen Element ähnlich sind (z.B. Internetsuche)
- Graphensuche: suche einen Weg von einem gegebenen Knoten zu einem anderen (z.B. Navigationsprogramm)

6.1 Schlüsselsuche

naive Methode: sequentielle Suche

Vorbedingungen:

- Daten liegen in einem Array
- Datenelemente können auf Identität (==) des Schlüssels verglichen werden

```
def sequential_search(a, target_key, key_func):
    for i in range(len(a)):
        if key_func(a[i]) == target_key:
            return i        # gefunden bei Index i
    return None             # nichts gefunden
                            # oder "return -1"
```

Vereinfachung: die Datenelemente sind der Schlüssel

```
def sequential_search(a, target_key):   #Komplexitaet:
    for i in range(len(a)):             #O(N)
        if a[i] == target_key:          #O(1)
            return i
    return None                         #O(1)
```

- Fall 1: nichts gefunden: $O(N) * O(1) + O(1) = O(N)$
- Fall 2: gefunden: im Schnitt: $\frac{N}{2}$ Schritte $\Rightarrow O(N)$

$\Rightarrow$ „lineare Suche“

Schnellere Suche: binäre Suche

Vorbedingungen:

- Elemente sind in sortiertem Array
- auf den Elementen gibt es eine totale Ordnung
  ⇒ impliziert Identität: a==b ⇔ not(a<b) and not (b<a)

⇒ Suche nach devide-and-conquer Prinzip

```
a = [...]
a.sort()
found = binary_search(a, target_key)

def binary_search(a, target_key):
    return binary_search_impl(a, target_key, 0, len(a))

def binary_search_impl(a, target_key, start, end):
    size = end - start
    if size <= 0:
        return None                    # nichts gefunden
    center = (start+end) // 2          # floor-Division
    if a[center] == target_key:
        return center                  # gefunden bei Index center
    elif a[center] < target_key:
        return binary_search_impl(a, target_key, center+1, end)
    else:
        return binary_search_impl(a, target_key, start, center)
```

Komplexität der binären Suche

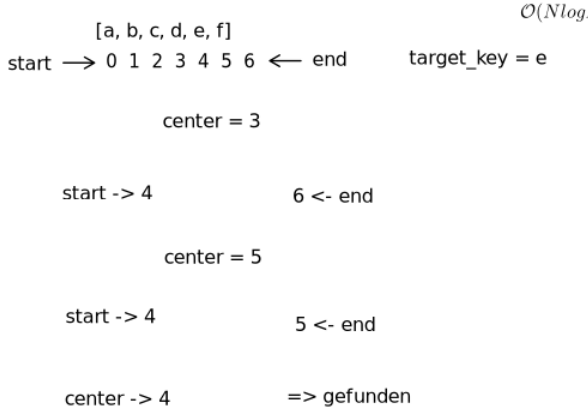
- vorab: Sortieren O(N log N)
- Alg. selbst: V(N) = 2 + V(N/2) = 2 + 2 + V(N/4) = ...

$$= \underbrace{2 + 2 + ... + 2}_{\lfloor log_2(N) \rfloor \text{Summanden}} = 2 * \lceil log_2(N) \rceil = O(logN)$$

⇒ wesentlich schneller als sequentielle Suche, **aber** man muss vorab sortieren

⇒ Sortieraufwand amortisiert sich über viele Suchanfragen, wenn man mindestens $\Omega(logN)$ Suchen ausführt

$$\mathcal{O}(NlogN + \mathcal{O}(logN)^2) = \mathcal{O}(NlogN)$$

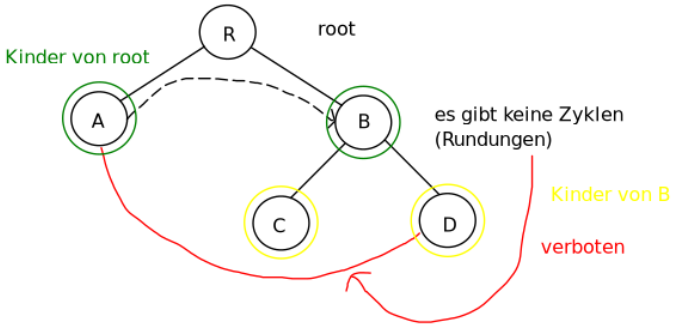


Anwendung: Daten werden vorab gesammelt und ändern sich dann nicht mehr
⇒ einmal sortieren, viele Suchvorgänge

ungünstig: Datenarray ändert sich häufig (Einfügen oder Löschen von Elementen)
⇒ Sortieraufwand amortisiert sich nicht ⇒ Suchbäume oder Hashtabelle verwenden

6.2 Suchbäume

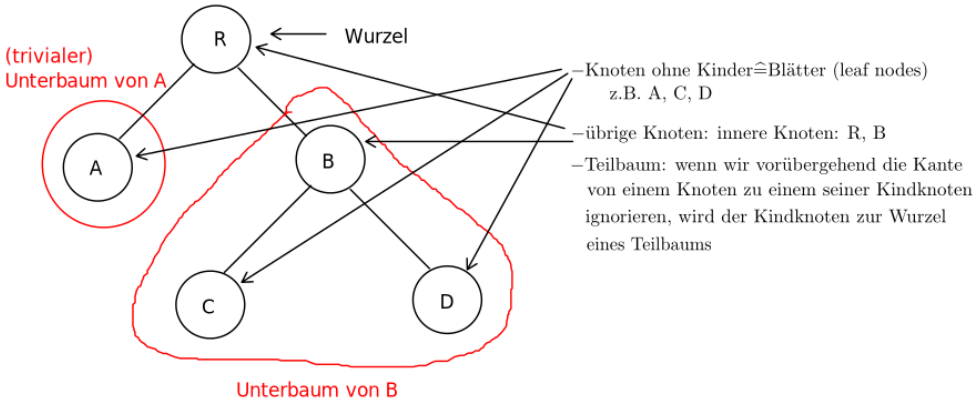
- effiziente Suche, wenn der Datenbestand sich häufig ändert
- „Bäume mit Wurzel" = rooted trees
  - spezielle Graphen, wo es von jedem Knoten zu jedem anderen genau einen Weg gibt
  - ein ausgezeichneter Knoten heißt „Wurzel", die zeichnet man oben :-)



  - speziell: Binärbäume – jeder Knoten hat maximal zwei Kinder
  - Kinder sind Nachbarknoten, die weiter von der Wurzel entfernt sind

in Python: Hilfsklasse für Knoten, die ein Datenelement und maximal 2 Kinder speichert

```
1  class Node:
2      def __init__(self, key):
3          self.key = key
4          self.left = self.right = None        #anfangs keine Kinder
5  root = Node("R")
6  root.left = Node("H"); root.right = Node("B"); root.right.left = Node("C")
```



Suchbaum:

1. auf den Schlüsseln ist eine Ordnung definiert „<“(wie Sortieren)
2. für jeden Knoten gilt:
   a) die Schlüssel im linken Teilbaum sind alle kleiner als der Schlüssel des Knotens
   b) die Schlüsse im rechten Teilbaum sind alle größer als der Schlüssel des Knotens

   (vgl.: analoges Verhalten beim Pivot in quick sort)

- zur Erinnerung:

```
class Node:
    def __init__(self, key):    #bei echten Anw: , data)
        self.key = key
        self.left = self.right = None
```

- Idee der Suche:
  - wenn der Schlüssel gefunden: gib den betreffenden Node zurück
  - sonst suche im linken oder rechten Teilbaum weiter ⇒ „teile und herrsche“
  - wenn ein Blatt und key nicht gefunden: gib None zurück

```
def tree_search(node, key):
    if node is None:                        # nichts gefunden, Rekursionsabschluss 1
        return None
    if key == node.key:                     # gefunden! Rekursionsabschluss 2
        return node
    if key < node.key:
        return tree_search(node.left, key)  # links weitersuchen
    else:
        return tree_search(node.right, key) # rechts weitersuchen
```

Verwendung:

```
root = ...          #Wurzel des Suchbaums
```

result = tree_search(root, key)

[in einer Programmbibliothek würde man all das noch „kapseln“, so dass der Nutzer nur ein interface sieht, aber nicht die interne Funktionalität, z.B. Klasse Node]

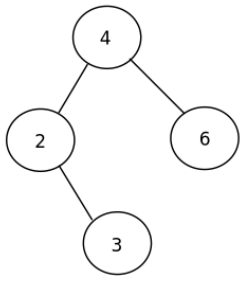
Einen neuen Schlüssel in den Baum einfügen

- muss auch funktionieren, wenn Baum leer ≙ root = None
- muss auch funktionieren, wenn der Schlüssel schon vorhanden ist
  Möglichkeiten:
  - Fehlermeldung (Exception)
  - besser: nur die mit dem Schlüssel verknüpften Daten überschreiben `a["Fritz Mueller"] = 1.0`

```
def tree_insert(node, key):                          # in der Praxis: ,data)
    if node is None:                                 # richtigen Platz gefunden
        return Node(key)                             # Konstruktor
    if key == node.key:                              # Schluessel schon vorhanden
        return node                                  # in Praxis: Daten ersetzen vorher
    if key < node.key:                               # key gehoert in den linken Teilbaum
        node.left = tree_insert(node.left, key)      #Rekursion
    else:
        node.right = tree_insert(node.right, key)    #Rekursion
    return node
```

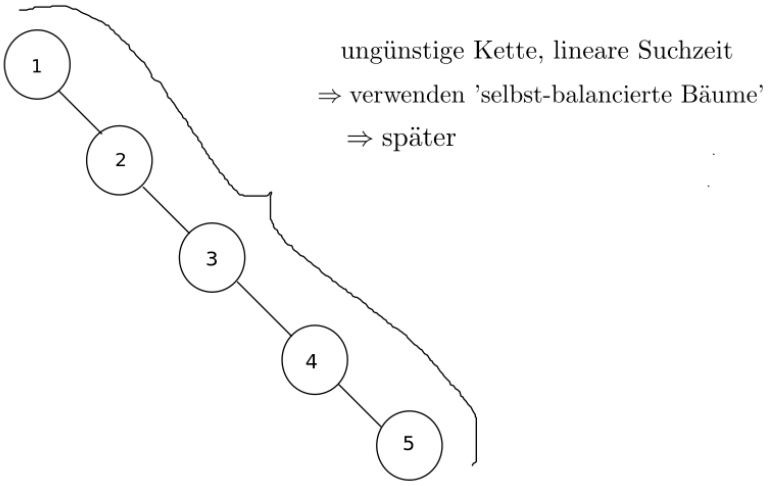
Verwendung:

```
root = None                     # leerer Baum
root = tree_insert(root, 4)
root = tree_insert(root, 2)
root = tree_insert(root, 3)
root = tree_insert(root, 6)
```



Wenn die Schlüssel in ungünstiger Reihenfolge eingefügt werden (z.B. sortiert oder umgekehrt sortiert), erhält man statt eines Baums eine Kette (≙ verkettete Liste) mit sehr langsamem Suchverhalten

Einfügen in der Reihenfolge 1, 2, 3, 4, 5



Entfernen eines Schlüssels aus dem Baum
vier Fälle:

1. der Schlüssel ist nicht vorhanden:
   - Fehlermeldung
   - remove gibt None zurück $\widehat{=}$ nichts entfernt
2. Schlüssel ist in einem Blatt ⇒ kann ihn einfach entfernen
3. Schlüsselknoten hat ein Kind ⇒ kann ihn durch Teilbaum des Kindes ersetzen
4. Schlüsselknoten hat zwei Kinder ⇒ ersetze nur den Schlüssel (plus evtl. die Daten) durch den Schlüssel eines geeigneten Blatts (→ der Vorgänger des zu entfernenden Schlüssels) und entferne dann dieses Blatt

```
def tree_predecessor(node):
    node = node.left                # suche größten Key im linken Teilbaum
    while node.right is not None:
        node = node.right
    return node

def tree_remove(node, key):
    if node is None:  # nicht gefunden
        return None
    if key < node.key:
        node.left = tree_remove(node.left, key)
    elif key > node.key:
        node.right = tree_remove(node.right, key)
    else:  # key gefunden
        if node.left is None and node.right is None:  #Blatt
            node = None
        elif node.left is None:  # node hat nur ein Kind
            node = node.right
        elif node.right is None:  # node hat nur ein Kind
```

```
            node = node.left
        else:  # node hat 2 Kinder
            pred = tree_predecessor(node)
            node.key = pred.key  # recycle node fuer seinen Vorgaenger
            node.left = tree_remove(node.left, pred.key)
    return node
```

Verwendung: `root = tree_remove(root, key)`

Komplexitätsanalyse der Suchbaumfunktion (ungünstiger Fall)

zwei Möglichkeiten:

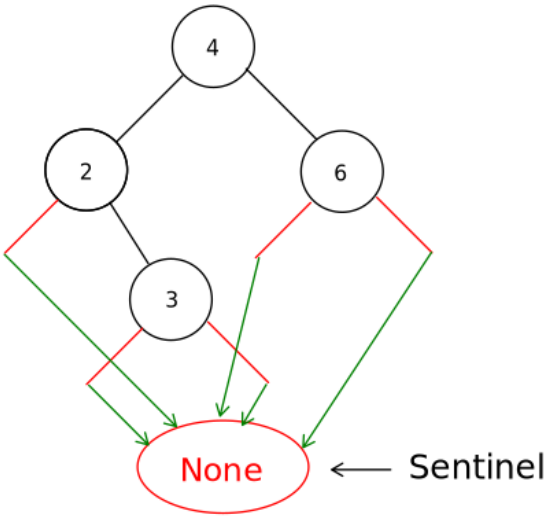
- key gefunden
- stelle fest, dass key nicht vorhanden

intuitiv: wenn Entscheidung viele Vergleiche erfordert ⇒ langsam
Begriffe:

- **Pfade**: Folge von Knoten $n_1, n_2, \cdots, n_k$
  so dass:
  - n_i und n_{i-1} sind Nachbarn
  - n_1, n_n sind zwei Knoten, zwischen denen wir den Pfad suchen
- **Länge des Pfades**: Anzahl der Kanten, k-1
- **Tiefe eines Knotens im Baum**: Länge des Pfades bis root
- **Tiefe des Baums**: maximale Tiefe eines Knotens

⇒ ungünstiger Fall: key wird am Ende des längsten Pfades gefunden bzw. nicht-vorhandensein festgestellt

- **Balance eines Baumes**: die Pfade von der Wurzel zum „Entscheiden" sollten möglichst ähnliche Länge haben
- **Trick**: zusätzlicher *virtueller* Knoten $\widehat{=}$ **Sentinel**
  darauf zeigen alle Knoten, die den Wert None (als linkes oder rechtes Kind) speichern
  in Python: das None-Objekt



⇒ RS-Pfade: von root zum Sentinel

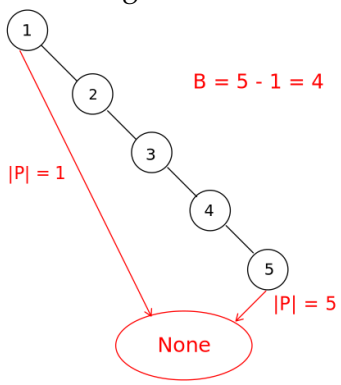
6.2.1 Balance des Baumes:

Differenz zwischen kürzestem und längstem RS-Pfad

$B = max|P| - min|P|$

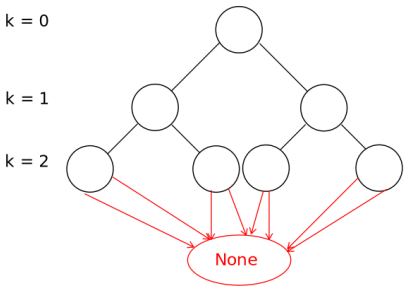
$P \in \{RS\} \quad P \in \{RS\}$

alle Pfade von root ⇒ Sentinel



6.2.2 Vollständiger Baum:

B = 0 → alle Knoten haben entweder 2 oder 0 Kinder



Anzahl der Knoten im vollständigen Baum: bei Tiefe k gibt es immer 2^k Blätter

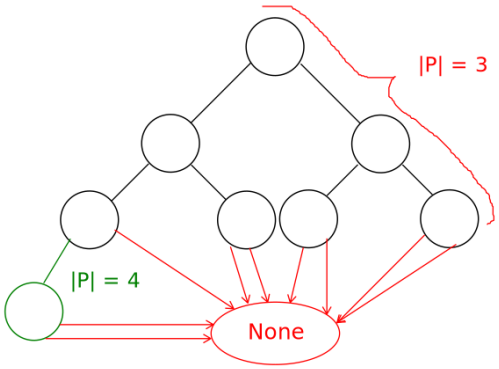
$N = 2^0 + 2^1 + \cdots + 2^{D+1} - 1$

$D \widehat{=}$ Tiefe

6.2.3 Perfekt balancierter Baum

B ≤ 1

⇒ für jeden Knoten gilt:

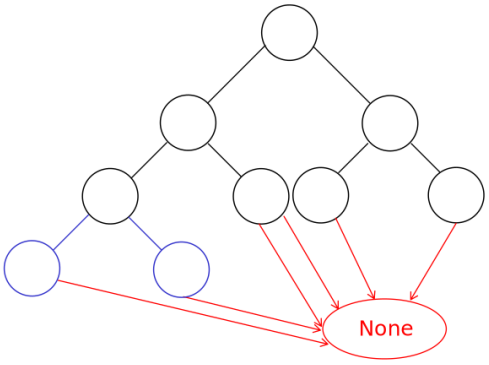


- linker und rechter Unterbaum sind auch perfekt balanciert
- Tiefen von linkem und rechtem Unterbaum unterscheiden sich höchstens um 1

6.3 Balance von Suchbäumen

B = (längster Pfad Wurzel → Sentinel) - (kürzester Pfad Wurzel → Sentinel)

Wieviele Elemente hat ein perfekt balancierter Baum der Tiefe d?



- vollständiger Baum, Tiefe 2
- perfekt balancierter Baum der Tiefe 3, d.h. B ≤ 1

- für Tiefe d muss es mindestens ein Element mit Abstand d von der Wurzel geben
  ⇒ mindestens ein Element mehr als der vollständige Baum der Tiefe (d-1): $N_v = 2^d - 1$
  $N_p \geq 2^d$
- um in die Tiefe d zu passen, darf der perfekt balancierte Baum höchstens ein vollständiger Baum sein ⇒ $N_p \leq 2^{d+1} - 1$

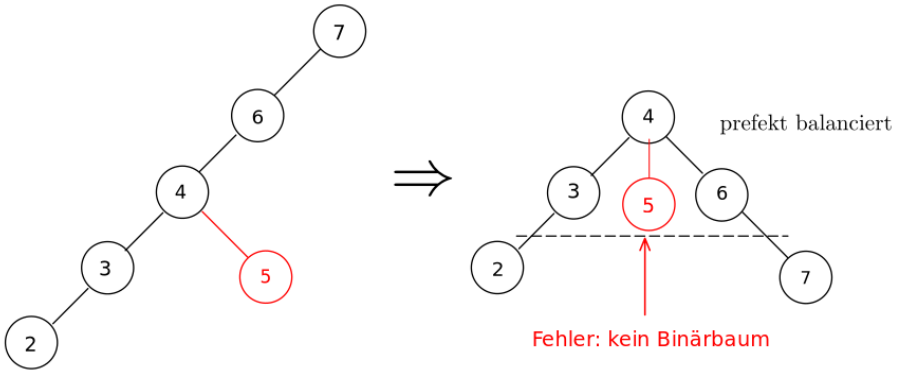
⇒ Komplexität der Suchoperationen: Anzahl der Vergleiche $\widehat{=}$ Länge des längsten RS-Pfades im ungünstigsten Fall, $V \in O(d)$ beim perfekt balancierten Baum der Tiefe d

$$2^d \leq N_p \leq 2^{d+1} - 1 \qquad log_2(2^d) \leq log_2(N_p) \leq log_2(2^{d+1} - 1) < log_2(2^{d+1})$$

⇒ $d \leq log_2(N_p) < d + 1 \Rightarrow V \in O(d) \Leftrightarrow O(logN_p)$
wie bei binärer Suche, also schnell.

⇒ Folgerung:

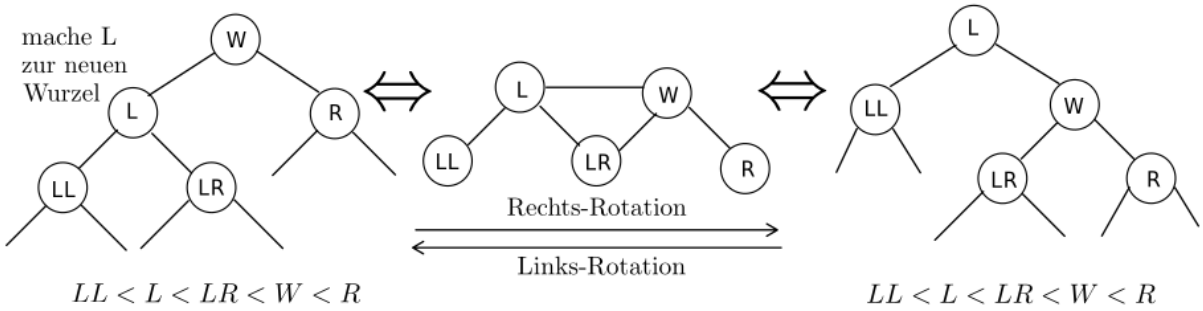
wenn Baumoperationen schnell sein sollen, müssen wie sicherstellen, dass der Baum balanciert ist
⇒ nach jedem insert oder remove wird die Balance „repariert“



3 Regeln für die Reparatur:

1. Der Baum muss ein Binärbaum bleiben
2. Die Suchbaumbedingung muss weiter gelten
3. Die Reparatur soll lokal erfolgen (in der Nähe des aktuellen Knotens) ⇒ Effizienz

Grundlegende Operation: **Baumrotation**: ändert die Wurzel eines Teilbaums nach obigen Regeln, so dass Balance besser wird



in Python:

```python
def tree_rotate_right(node):
    new_root = node.left
    node.left = new_root.right
    new_root.right = node
    return new_root
```

```python
def tree_rotate_left(node):
    new_root = node.right
    node.right = new_root.left
    new_root.left = node
    return new_root
```

Beispiel: AVL-Bäume

- transformieren den Baum immer in perfekte Balance
- aber: das ist schwierig und unnötig,

⇒ „einfache" Balance reicht aus:

$V \in O(d)$

$V \leq c - d$ für geeignete Konstante (unabhängig von d bzw. N)

⇒ wenn die Tiefe nur um einen konstanten Faktor schlechter ist als beim perfekt balancierten Baum, verschwindet die Konstante in $O(d)$ ⇒ gleiche Komplexität

⇒ viel einfachere Implementation
Beispiele:

- Rot-Schwarz-Baum (typische Implementation)
- Treap (⇒ siehe Hausaufgabe)
- Anderson-Baum (Vereinfachung des Rot-Schwarz-Baum)

6.4 Anderson-Bäume

Idee: jeder Knoten hat ein *level* $\widehat{=}$ Abstand (kürzester Pfad) zum None-Sentinel

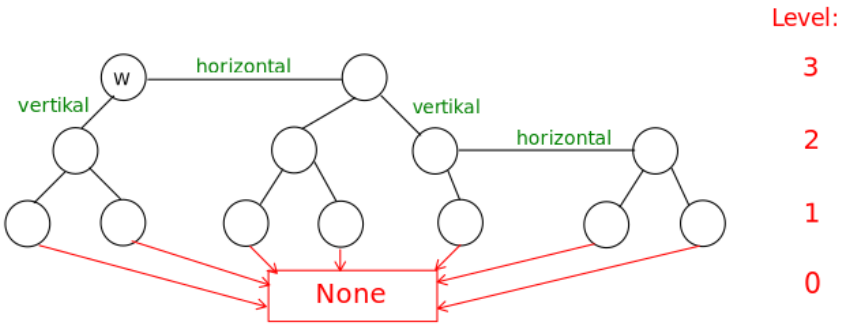
```python
class AndersonNode:
    def __init__(self, key):
        self.key = key
        self.left = self.right = None
        self.level = 1  #distance to None in left and right
```

Idee: Im Anderson-Baum gibt es

- horizontale Kanten (parent.level == child.level)
- vertikale Kanten (parent.level == child.level + 1)

Regeln:

- die reduzierte Länge eines Pfades r: nur die vertikalen Kanten gezählt
- Sentinel hat level == 0 und alle Kanten zum Sentinel sind vertikal
  $\Rightarrow$ echte Knoten haben level ≥ 1
- reduzierte Höhe (h_r) eines Knotens entspricht der reduzierten Pfadlänge (r)
- Alle RS-Pfade haben die **gleiche** reduzierte Länge. $\Rightarrow$ jeder Knoten, insbesondere die Wurzel werden auf allen Pfaden von None nach der gleichen Anzahl vertikaler Kanten erreicht (*aber*: die Anzahl horizontaler Kanten kann verschieden sein)
- kein Pfad darf zwei horizontale Kanten hintereinander haben
- Vereinfachung durch Anderson: nur Kanten zum rechten Kind dürfen horizontal sein (sonst: Rot-Schwarz-Baum)



Satz: jeder Anderson-Baum ist balanciert (höchstens einen konstanten Faktor schlechter als der perfekt balancierte Baum), Beweis:

1. h_r = reduzierte Höhe des Baums (Abstand Wurzel $\rightarrow$ Sentinel *ohne* horizontale Kanten)
   - hat der Baum *keine* horizontalen Kanten $\Rightarrow$ vollständiger Baum Tiefe $d_v = h_r - 1$
   - hat der Baum horizontale Kanten, hat er *mehr* Konten als der vollständige Baum

     $$N \geq 2^{d_v+1} - 1 = 2^{h_r} - 1$$

2. Da nie zwei horizontale Linien aufeinander folgen, ist die tatsächliche Tiefe höchstens zweimal die reduzierte Tiefe: $d \leq 2 * h_r$

3. Zusammenfassen

$$N \geq 2^{d/2} - 1 \qquad N + 1 \geq 2^{1/2}$$

$$log_2(N+1) \geq d/2 \qquad d \leq 2log_2(N+1))$$

Da der schlechteste Fall des Suche: $V \in O(d) = O(2log_2(N+1))$

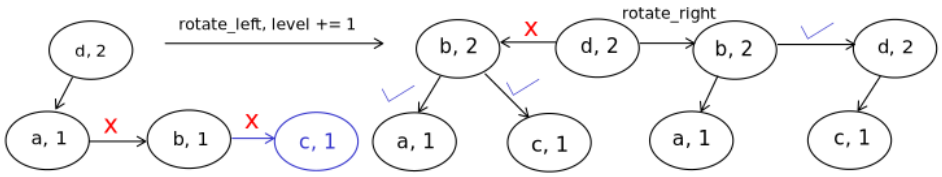
$$\Rightarrow \boxed{V = O(logN)} \qquad \square$$

Prinzip, den Baum balanciert zu halten bei *insert*:

- füge den neuen Knoten normal ein (rekursive Aufrufe von `tree_insert`)
- prüfe, ob die Anderson-Bedingungen beim parent des neuen Knotens verletzt sind ⇒ repariere den Unterbaum von parent
- auf dem Rückweg der Rekursion: prüfe und repariere auch alle Vorfahren auf dem Pfad bis zur Wurzel

in Python:

```
def anderson_tree_insert(node, key):
    if node is None:
        return AndersonNode(key)
    if key == node.key:
        #Daten aktualisieren
        return node
    if key < node.key:
        node.left = anderson_tree_insert(node.left, key)
    else:
        node.right = anderson_tree_insert(node.right, key)
    # bis hier wie bei tree_insert()
    # ab hier Balance reparieren
    if node.left is not None and node.level == node.left.level:
        node = tree_rotate_right(node)      #turn left horizontal into right horizontal
    if node.right is not None and node.right.right is not None and node.level ==
    ↪  node.right.right.level:    #zwei horizontal hintereinander
    # node.right -> anheben und zur Wurzel machen
    node = tree_rotate_left(node)
    node.level += 1
    return node
```



6.5 Prioritätssuche

- Variante der Suche: statt nach einem konstanten Schlüssel suchen wir den
  - größten Schlüssel (max-priority search)
  - kleinsten Schlüssel (min-priority search)
- andere Interpretation als Verallgemeinerung von Stack und Queue, anschaulich: Elemente mit hoher Priorität dürfen in der Schlange vordrängeln
  - Annahme: beim push in Stack oder Queue wird für das betreffende Element der Zeitstempel mitgespeichert
  - definiere Priorität als $|now - timestamp_i|$ $\Rightarrow$ höchste Priorität $\widehat{=}$ am längsten im Container
    $\Rightarrow$ first-come – first-served Verhalten $\widehat{=}$ Queue

    $\Rightarrow$ kleinste Priorität $\widehat{=}$ am kürzesten im Container $\Rightarrow$ $\left.\begin{array}{l}\text{last-come – first-served}\\ \text{last in - first out}\end{array}\right\}$ Stack
- elementare Eigenschaft: max priority search $\Leftrightarrow$ zu min priority search mit negierten Schlüsseln
- naive Implementation: sequentielle priority search:

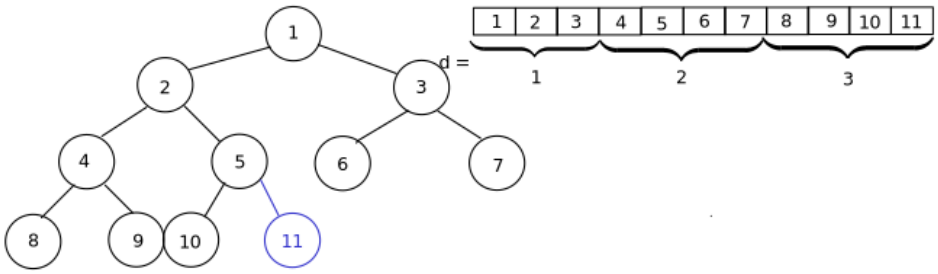
```python
def sequential_priority_search(a):  #max priority
    m = 0
    for i in range(1, len(a)):
        if a[i] > a[m]:
            m = i
    return m        # Index des groessten Elements
    # => innere Schleife von selection sort (aber min priority) => O(N)
```

- Implementation mittels Suchbaum: das größte Element ist das „rechteste Kind“

```python
def tree_priority_search(node):
    if node is None:
        raise KeyError("empty tree")
    while node.right is not None:
        node = node.right
    return node
    # => sehr aehnlich zu tree_predecessor (aber min pr.) => O(d) = O(log N) im
    ↪ balancierten Baum
```

- Beobachtung: $O(logN)$ kann man viel einfacher erreichen als mit einem Suchbaum ⇒ Heap („priority search tree“)
  - ersetze Suchbaumbedingung durch Heapbedingung:
    „kein Schlüssel im linken oder rechten Teilbaum hat höhere (niedrigere) Priorität als die Wurzel jedes Teilbaums im max heap (min heap)“
  - das erweist sich als viel einfacher, weil man
    1. immer einen perfekt balancierten Baum garantieren kann (ohne komplexe Umstrukturierungen wie beim Andersson-Baum)
    2. den Baum als Array (= sehr effizient) speichern kann

  ⇒ linkslastigeR perfekt balancierteR Baum kann eindeutig „geflattet“ werden, d.h. als Array „flachgedrückt“



  - Adressierung der Knoten im geflatteten Baum a
    - a[0] ⇒ Wurzel
    - a[1], a[2] linkes und rechtes Kind der Wurzel
    - a[3], a[4] linkes und rechtes Kind von a[1] usw.
    - generell gilt:
      - die Kinder von a[i] sind a[2*i + 1] linkes Kind
        a[2*i + 2] rechtes Kind
      - der Parent von a[i] ist a[(i-1) // 2] (floor division → abrunden)
    - die Umrechnungen ersetzen die Zugriffe `node.left` und `node.right` im Suchbaum

- aus der Heap-Bedingung folgt: das Element mit größter Priorität ist immer die Wurzel des ganzen Baums: a[0]

```
1  def heap_largest(a):
2      return a[0]          #Komplexitaet O(1)
```

- Einfügen in den Heap: Nutze die Eigenschaft des dynamischen Arrays, das Einfügen am Ende billig ist (amortisiertes $O(1)$):
  - füge neue Elemente am Ende an (dann bleibt der Baum linkslastig perfekt balanciert)
  - repariere eventuell die Heap-Bedingung, falls das neue Element größer als sein parent ist.

in Python:

```python
def heap_insert(a, key):        #max heap
    a.append(key)               #am Ende anhaengen
    upheap(a, len(a)-1)     #reparieren des aktuell gueltigen Bereichs

def upheap(a, k):
    v = a[k]        #zwischenspeichern
    while True:     #Endlosschleife (durch "break" verlassen)
        if k == 0:  #a[k] Wurzel
            break   #Heap-Bedingung auf jeden Fall erfuellt
        parent = (k-1) // 2
        if a[parent] > v:   #Heap-Bedingung erfuellt
            break
        a[k] = a[parent]    #Heap-Bed. reparieren: parent eine Ebene nach unten
        ↪ schieben
        k = parent
    a[k] = v    #Element v korrekt einsortieren
```

Komplexität

$\widehat{=}$ Anzahl der Durchläufe durch while-Schleife

$\leq$ der ursprünglichen Tiefe des Knotens $k \in O(logN)$

- analog funktioniert das Entfernen: `pop()` $\Leftrightarrow$ `remove_largest()`
  Idee:
  - tausche Wurzel mit dem letzten Element des Arrays
  - Entferne letztes Element (amortisiert $O(1)$)
  - repariere die Heap-Bedingung, wenn neue Wurzel nicht größtes Element

```python
def heap_remove_largest(a):
    a[0] = a[len(a) - 1]        #largest ersetzen
    del a[len(a) - 1]           #letztes loeschen
    # = a[-1]
    downheap(a)             #Heap-Bedingung reparieren

def downheap(a, last = None):
    if last is None: last = len(a) - 1
    k = 0   #Wurzel ist eventuell nicht das groesste Element
    v = a[k]
    while True:     #ab hier insgesamt: O(d) = O(log N)
        child = 2 * k + 1   #Index des linken Kinds
        if child > last:    #Kind existiert nicht => Heap-Bedingung erfuellt
            break
        if child + 1 <= last and a[child] < a[child + 1]:
        # rechtes Kind existiert & rechtes Kind hat hoehere Prioritaet als linkes
            child = child + 1
        if v >= a[child]:   #Heap-Bedinung erfuellt
```

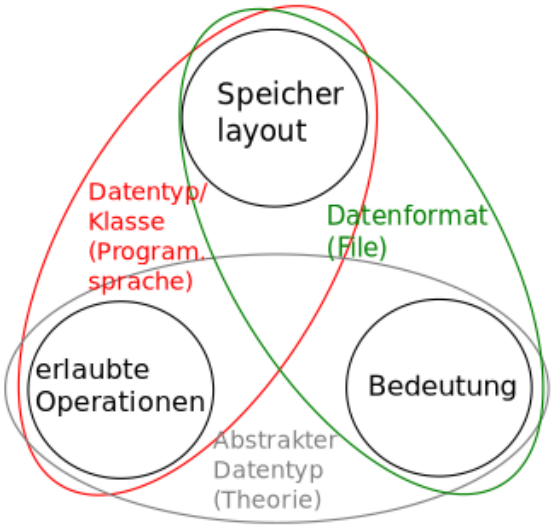
```
            break
        a[k] = a[child]     #child eine Ebene hoch
        k = child
    a[k] = v                # Element v korrekt einsortieren
```

- aus dem Heap-Verhalten kann man einen effizienten Sortieralgorithmus ableiten:
  - Idee: unsortiert in den Heap einfügen, immer das größte wieder entnehmen
    $\Rightarrow$ man bekommt die Elemente in absteigender Sortierung
  - Komplexität: $O(NlogN)$ wie Merge sort
  - überraschende Beobachtung: das geht in-place: man braucht nur das Array für den geflatteten Heap, wie Quick Sort
  - in Python:

```
a = ... # unsortiertes Array, benutze gleichzeitig als Heap
heap_sort(a)
def heap_sort(a):
    N = len(a)
    for k in range(1, N):  #Einfuegephase
        upheap(a, k)
    # jetzt ist a ein Heap, d.h. so sortiert, dass die Heap-Bedingung gilt
    for k in range(N - 1, 0, -1):  # Entfernungsphase
        #groesstes Element an die sortierte Position bringen
        a[0], a[k] = a[k], a[0]
        downheap(a, k - 1)  #repariere Heap bis Indes k - 1 ("Restheap")
```

7 Assoziative Arrays

7.1 Datenstrukturdreieck am Beispiel Assoziativer Arrays



Abstrakter Datentyp		**Datentyp in Python**
Operationen	Bedeutung	Welche Fkt. implementiert man?
Schlüssel k , Werte v, Array a		Hilfsklasse Node
`a[k] = v`	Speichere v unter dem Schlüssel k oder ersetze Daten, falls k vorhanden	`__setitem__(self, k, v)`
`v = a[k]`	frage Daten vom Schlüssel k ab, Fehlermeldung falls k nicht vorhanden	`__getitem__(self, k)`
`a.has_key(k)`	True, wenn k vorhanden	`has_key(self, k`
`del a[k]`	Lösche Schlüssel k und seine Daten, oder Fehlermeldung, wenn k nicht vorhanden	`__delitem__(self, k)`
`len(a)`	aktuelle Anzahl der Schlüssel bzw. Schlüssel/Wert-Paare	`__len__(self)`

k im Prinzip beliebig (Anforderungen später)

Fileformate für assoziative Arrays:

- lesbar für Menschen und Maschinen (als Textfiles): XML, JSON, YAML
- nur lesbar für Maschinen (als Binärfile): HDF5

7.2 JSON-Format

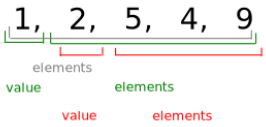
7.2.1 Spezifikation des JSON-Formats

in Backus-Naur-Notation:

JSON-file	:=	array |
	|	dictionary
array	:=	'[' elements ']'
	|	'[' ']' #leeres Array
elements	:=	value
	|	value ',' elements
dictionary	:=	'{' pairs '}'
	|	'{' '}' #leeres Dictionary
pairs	:=	string ':' value
	|	string ':' value ',' pairs
string	:=	'"' '"' #leerer String
	|	'"' characters '"'
value	:=	number | string | boolean | null | array | dictionary

Schlüssel

werden als Zeichen im UTF-8 Zeichensatz codiert



Beispiel für JSON: Studenten-Datenbank

```
Students = {
    "Fritz Mueller": {
        "Mathematik": [2.0, 1.7, 1.3],
        "ALDA":       1.3
    },
    "Anna Weise": {
        "Mathematik":  [1.0, 1.0, 1.0];
        "Philosophie": 1.3
    }
}
```

Einlesen von JSON in Python: json-Modul

```
import json
students = json.load(file("students.json").read().decode("utf_8"))
```

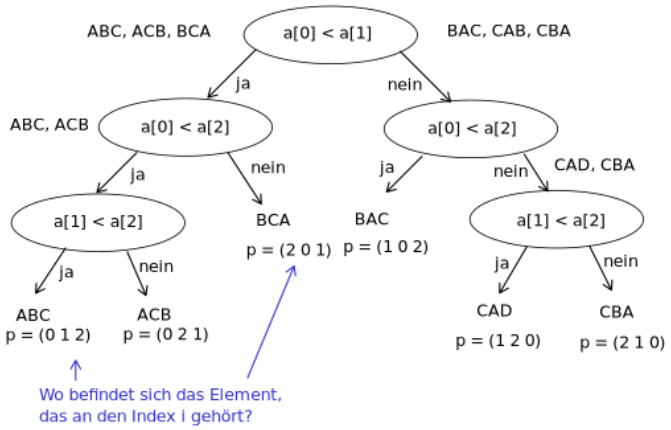
Anforderungen an die Schlüssel :	Implementation
`key1 == key2`	sequentielle Suche $O(N)$
Ordnung `key1 < key2` (impliziert `key1 == key2`)	binäre Suchbäume, binäre Suche $O(logN)$
`key1 == key2` `hash(key1) => integer`	Hashtabelle $O(1)$

8 Effizientes Sortieren und Suchen

in $O(N)$ (Sortieren) bzw. $O(1)$ (Suchen)

Beweis wenn man nur die Operation `key1 < key2` (Ordnung) zur Verfügung hat, ist Sortieren nicht besser als $\Omega(NlogN)$ möglich

Beispiel Array der Länge 3: kann in 6 = 3! verschiedenen Sortierungen vorliegen: ABC, ACB, BAC, BCA, CAB, CBA
Ratespiel: mit möglichst wenigen Fragen herausfinden, welche Permutation vorliegt ⇒ Suchbaum, Fragebaum



wenn man p kennt, kann man in $O(N)$ sortieren

```
def index_sort(a, p):
    v = [None] * len(a)
    for k in range(len(a)): v[k] = a[p[k]] #O(N)
    return v
```

Wie groß ist der Fragebaum mindestens, wenn wir die Frage so geschickt wie möglich stellen?

Um N! Permutationen zu repräsentieren, brauchen wir N! Blätter

Um N! Blätter zu haben, muss der Baum die Tiefe $\Omega(logN!)$ haben

Vereinfachen mit Hilfe der Stirlingschen Formel für große N:

$$N! \approx \sqrt{2\pi N} * \left(\frac{N}{e}\right)^N$$

$$log\ N! \approx log\ \sqrt{2\pi N} + log\ N^N - log\ e^N \geq log\ \sqrt{2\pi N} + log\ N^N$$

$$\Omega(log\ N!) = \Omega(log\ \sqrt{2\pi N}) + \Omega(log\ N^N) = \boxed{\Omega(N\ log\ N)}$$

⇒ Sortieren allein mit paarweisen Vergleichen "key1 < key2" braucht immer mindestens $\Omega(N \log N)$ Zeit

⇒ Merge Sort, Quick Sort, Heap Sort sind nicht zu verbessern, ohne zusätzliche Anforderungen in die Schlüssel

8.1 Sortieren in linearer Zeit $\mathcal{O}$(N)

- ist einfach, wenn die Schlüssel ganze Zahlen im Bereich $[0, \cdots, M-1]$
  so dass $M \in \mathcal{O}(N) \Rightarrow$ benutze Array(∞) von Arrays $\rightarrow$ ein Array pro Schlüssel (genauer: Queue)

```
1  def integer_sort(a, M):
2      bucket = [ [] for k in range(M)] #M leere Queues ([])
3      #M = Anzahl der erlaubten Schluessel
4      #verteile Raten auf Buckets
5      for k in range(len(a)):
6          bucket[a[k]._key].append(a[k])  #a[k]._key Element von [0, M-1]
7
8      # sortiertes Array aus den Queues zusammensetzen
9      start = 0
10     for k in range(M):
11         end = start + len(bucket[k])
12         [a[start:end] = bucket[k]   #alle Elemente mit Schluessel k in a sortiert
           ↪  einfuegen
13         start = end
```

Komplexität

- line 1-3: $\mathcal{O}(M)$
- line 4-6: $\mathcal{O}(N)$
- line 10-13: $\mathcal{O}(N)$ $\begin{cases} \text{line } 11 : \mathcal{O}(k) \\ \text{line } 12 : \mathcal{O}(N_k) \\ \text{line } 13 : \mathcal{O}(1) \end{cases}$

Schleife über Buckets hat Komplexität:

$$\sum_{\Psi=0}^{M-1} \mathcal{O}(N_k) = \mathcal{O}\underbrace{\left(\sum_{k=0}^{M-1} N_k\right)}_{=N} = \mathcal{O}(N)$$

⇒ der gesamte Algorithmus ist $\mathcal{O}(N)$, falls $M \in \mathcal{O}(N)$

Wie kann man das auf beliebige Schlüssel verallgemeinern?

- Quantisierung: `quantize(key, M)` $\Rightarrow$ `[0, M-1]` $\widehat{=} q_{\text{key}}$
  Ordnungserhaltend: falls `key1` $<$ `key2` $\Rightarrow q_{\text{key1}} \leq q_{\text{key2}}$
  wenn eine solche Funktion für keys definiert ist $\Rightarrow$ bucket sort $\in O(N)$

8.2 Bucket Sort

```
1  def bucket_sort(a, quantize, d):
2      N = len(a)
3      M = int(N / float(d)) #Python3 N//d
4      # M = Anzahl der Buckets, d = mittlere Zahl von Elementen in jedem Bucket
5
6      bucket = [[] for k in range(M)]
7      #Daten auf buckets verteilen
8      for k in range(N):
9          index = quantize(a[k]._key, M)
10         bucket[index].append(a[k])
11     #Daten sortiert einfuegen
12     start = 0
13     for k in range(M):
14         insertion_sort(bucket[k])   #sortiere Schluessel mit gleicher Quantisierung
15         end = start + len(bucket[k])
16         a[start:end] = bucket[k]
17         start = end
```

Komplexität:

- line 2: $O(1)$
- line 3: $O(1)$
- line 6: $O(M)$
- line 8-10 : $O(N)$ $\begin{cases} \text{line 8: } O(N) & \\ \text{line 9: } O(1) & ! \\ \text{line 10: } O(1) & \text{amortisiert} \end{cases}$
- line 12: $O(1)$
- line 13 - 17: $\sum_{k=0}^{M-1} O(N_k^2)$ $\begin{cases} \text{line 13:} & O(M) \\ \text{line 14:} & O(N_k^2) \\ \text{line 15:} & O(1) \\ \text{line 16:} & O(N_k) \\ \text{line 17:} & O(1) \end{cases}$

zum Vergleich: integer_sort:

$$\sum_{k=0}^{M-1} O(N_k) = O\left(\sum_{k=0}^{M-1} N_k\right) = O(N)$$

jetzt:

$$\sum_{k=0}^{M-1} O(N_k^2) = \sum_{k=0}^{M-1} O(1) = O(M) \quad \underline{\text{falls}}\ N_k \in O(1), \text{ d.h. unabh. von N und M}$$

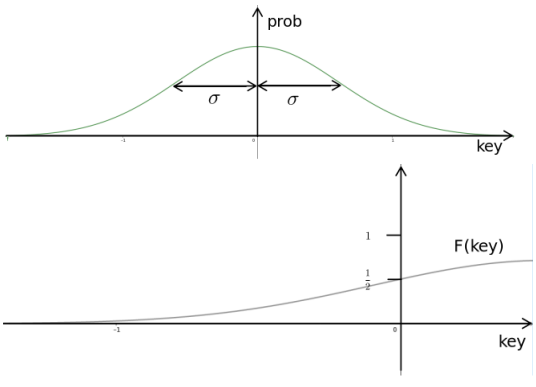
Wie erreicht man, dass $N_k \in O(1)$?

1. Lege M so fest, dass $M \in O(N)$: externer Parameter $d \approx 10$ gibt den gewünschten Füllstand jedes Buckets an
$$M = \lfloor \frac{N}{d} \rfloor \in O(N)$$
2. Teile Schlüssel so auf die Buckets auf, dass $N_k \approx d$ für alle k
⇒ Komplexität für Bucketsort:
$$\sum_{k=0}^{M-1} O(N_k^2) = \sum_{k=0}^{M-1} O(d^2) = \sum_{k=0}^{M-1} O(1) = O(M) = O(N)$$
⇒ brauchen quantize(), die das sicherstellt
   - einfachster Fall: alle Schlüssel sind reelle Zahlen $\in [0, 1)$, gleichverteilt

```
1  def quantize_uniform(key, M):
2      return int(key * M)
```

   ⇒ $\mathbb{E}_k[N_k] = \frac{N}{M} = O(1)$
   - für andere Wahrscheinlichkeitsverteilungen der Schlüssel muss man die Verteilung kennen, um eine gute quantize-Fkt. zu definieren

a) Formel $p(\text{key}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\frac{\text{key}^2}{\sigma^2}}$



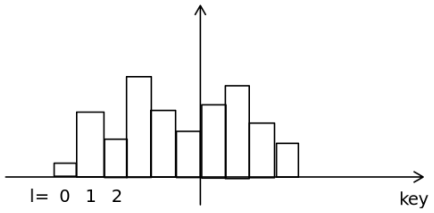
kumulative Verteilungsfunktion:

$$F(\text{key}) = \int_{-\infty}^{\text{key}} p\,(\text{key}')\; d\,\text{key}$$

⇒ optimale Quantisierung: int(F(key)*M)

b) keine Formel: empirische Wahrscheinlichkeit $\widehat{=}$ Histogramm

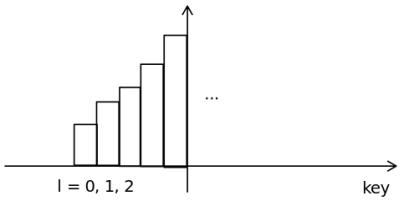
⇒ optimale Quantisierung: l=raw_quantize(key), k = int($F_e * M$) F[l]



Quantisiere Schlüssel in L Bins
zähle Häufigkeit der Schlüssel für jeden Bin:

$h_l \in [0, 1]$

kumulatives Histogramm $F_e = \sum_{l'=0}^{l} h_{e'}$



Hausaufgabe: optimale Quantisierung für Fall a) finden & bucket_sort implementieren
Die Quantize-Funktion sollte möglichst billig sein.

9 Hashtabellen

Suchen in $O(1)$ pro Query (d.h. $O(N)$ zum Aufbau der Tabelle)

Trick wie bei bucket_sort: i = hash(key)⇒ ganze Zahl $\in [0, M)$
Unterschied zu quantize: hash(key) muss **nicht** die Ordnung erhalten
⇒ mehr Freiheit bei der Auswahl von hash(key)
die selbe Hash-Funktion funktioniert für viele verschiedene Wahrscheinlichkeitsverteilungen der keys.

Wertebereich von hash(): U $\widehat{=}$ „Universum“ des Schlüssels
z.B. wenn key ein String der Länge 9, 60 erlaubte Zeichen

$$|U| = 60^9 \approx 10^{16}$$

dagegen: $M \in O(N) << |U| \Rightarrow$ Viele Schlüssel werden durch hash(key) auf den gleichen Wert abgebildet $\widehat{=}$ **Kollision**
gute Hashfunktion: Kollision für alle Schlüssel gleich wahrscheinlich
($\widehat{=}$ bucket_sort: alle Buckets sind gleich voll)

Beispiel: Monatsnummern als Schlüssel `"Januar", "Februar", ..., "Dezember"`
hash(key) ⇒ key[0] Anfangsbuchstabe:

J F M A M J J A S O N D viele Kollisionen

klassisch (Papierformulare) : hash(key)⇒ key[0:3] erste 3 Buchstaben
Jan, Feb, ... keine Kollision, aber M sehr groß ($60^3 = 216000 >> N$)

perfekte Hashfunktion: hash(key) ⇒ [0,11] [1,12]

9.1 Bewährte Hashfunktionen

Idee: Interpretiere jeden Schlüssel als Bytefolge

Bernstein:

```
def b_hash(u):          #u: Array of Bytes
    h = 0
    for i in u:
        h = 33*h + i    # 33 wurde experimentell bestimmt fuer wenig Kollision
    return h
```

$h \in [0, \cdots, 2^{32} - 1]$ int32
$h \in [0, \cdots, 2^{64} - 1]$ int64

modifizierte Bernstein:

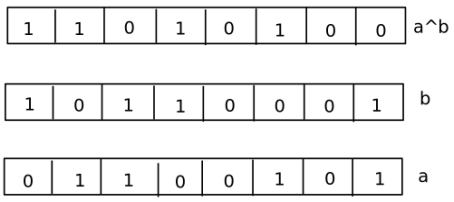
```
def mb_hash(u):          #u: Array of Bytes
    h = 0
    for i in u:
        h = 33*h^i       # ^: bitweise XOR
    return h
```

shift-Add-XOR:

```
def sax_hash(u):
    h = 0
    for i in u:
        h^= (h << 5) + (h >> 2) + i
    return h
```

Fowler | Noll | Vo:

```
def FNV_hash(u):
    h = 2166136261
    for i in u:
        h = (16777619 * h)^i
    return h
```



9.2 Prinzip der Hashtabelle

- intern: Array der Länge M (dynamisches Array: vergrößere M falls N zu groß wird ⇒ später)
- speichere Schlüssel key beim Index i = $\underbrace{\text{hash(key)}}_{0\ldots(2^{32}-1)} \,\%\, \underbrace{M}_{>>O(N)} \in [0, M-1]$
- mache etwas trickreiches bei Kollision
  - lineare Verkettung „offenes Hashing“ „geschlossene Adressierung“
  - offene Adressierung
  - doppeltes Hashing

9.3 Hashtabelle mit linearer Verkettung

- wie bei bucket_sort: für jeden Index hat man ein Array oder eine verkettete List, wo alle Schlüssel mit demselben Hash landen:

```python
class HashNode:
    def __init__(self, key, value, next):
        self.key = key
        self.value = value
        self.next = next

class HashTabelle:
    def __init__(self):
        self.capacity = ...             # initiales M
        self.size = 0
        self.data = [None] * self_capacity

    def __setitem__(self, key, value):
        i = hash(key) % self.capacity   #bucket finden
        bucket = self.data[i]           #bucket finden
        while bucket is not None:
            if bucket.key == key:       #key war schon vorhanden
                bucket.value = value    # => value ersetzen
                return
            else:
                bucket = bucket.next     #durch verkettete Liste iterieren

        # wenn wir hier landen, war key noch nicht  vorhanden
        # => neues Element einfuegen

        new_node = HashNode(key, value, self.data[i])
        self.data[i] = new_node
        self.size += 1
        ... # hier eventuell die capacity vergroessern

    def __getitem__(self, key):
        i = hash(key) % self.capacity
        bucket = self.data[i]
        while bucket is not None:
            if bucket.key == key:
                return bucket.value
            bucket = bucket.next
        raise keyError(key)
```

Komplexität der Hash-Tabelle mit linearer Verkettung

- Ausrechnen des Hashs und finden des Buckets: $O(1)$
- Schlüssel im Bucket suchen: (N_i: Länge Bucket i)

- vorhanden $O\left(\frac{N_i}{2}\right)$
- nicht vorhanden $O(N_i)$

- mittlere Suchzeit für viele Aufrufe:

$$\mathbb{E}[t] = \sum_{i=0}^{M-1} p_i O(N_i) \qquad M : \text{capacity}$$

Annahme: gute Hash-Funktion ⇒ Schlüssel gleichmäßig auf die Buckets verteilt
$\Rightarrow p_i = \frac{1}{M}, \qquad N_i = \frac{N}{M}$

$$\mathbb{E}[t] = \sum_{i=0}^{M-1} \frac{1}{M} O\left(\frac{N}{M}\right) = O\left(\frac{N}{M^2}\right) \underbrace{\sum_{i=0}^{M-1} 1}_{=M} = O\ \underbrace{\left(\frac{N}{M}\right)}_{\text{Füllstand jedes Buckets}}$$

- Gesamtzeit soll $O(1)$ sein:

$$O(1) \stackrel{!}{=} O(1) + \underbrace{O\left(\frac{N}{M}\right)}_{O(1)} \Leftrightarrow M \in O(N)$$

⇒ wenn Hashtabelle voll ⇒ verdoppele Kapazität M
Trick: setze M als Primzahl ⇒ keys werden gleichmäßiger verteilt bei hash(key)%M

$$M \in \{11, 23, 47, 97, 199, 409, 823, \cdots\}$$

z.B. häufig verwendet für C++ Klasse std::unordered_map

9.4 Hashtabellen mit offener Adressierung

- HT mit linearer Verkettung: pessimistisch $\widehat{=}$ es wird viele Kollisionen geben
  ⇒ jeder Index enthält von vornherein eine Liste von Keys

- HT mit offener Adressierung: optimistisch $\widehat{=}$ Kollisionen sind so selten, dass wir keine Liste für jeden Index brauchen
  $\widehat{=}$ jeder Index enthält kein oder ein Element

- einfachste Variante: **sequentielles Sondieren**: wenn der gewünschte Index bereits belegt ist
  ⇒ probiere den nächsten bis man einen freien Index findet
  Konsequenzen:
  - capacity > size notwendig, sonst Endlosschleife
  - Entfernen von Elementen nicht mit naivem Vorgehen (Element auf None setzen) möglich

statt dessen: gelöschte Elemente als *gelöschte* markieren, damit die Suchkette bei Kollisionen nicht vorzeitig abbricht

Nachteil: Clusterbildung

- Bereiche, wo alle Indizes belegt sind
- Bereiche, wo alle Indizes frei sind

⇒ Suchzeit linear in der Länge der Cluster, nicht immer $O(1)$

- verbesserte Variante: doppeltes Hashing: verwende 2. Hashfunktion, um den Ersatzindex festzulegen ⇒ gleichmäßige Verteilung der Belegung

Doppeltes Hashing a la Python:

```python
def __setitem__(self, key, value):
    i = hash(key)%self.capacity
    h = hash(key)
    while True:
        if self.data[i] is None:    #freies Feld gefunden => key nicht vorhanden
            break
        if self.data[i].key == key: #key gefunden => Daten aktualisieren
            self.data[i].value = value
            return
        #wenn wir hier landen: Kollision(Index i mit anderem key belegt)
        i = (5 * i + 1 + h) % self.capacity
        h //= 32        # h = h // 32

        #wenn wir hier landen wurde der Key nicht gefunden
        h = hash(key)
        i = h % self.capacity
        #zweite Schleife: neues Element einfuegen
        while True:
            if self.data[i] is None or self.data[i].key is None:
                #index ist frei(1. Bedingung) oder als geloescht markiert(2. Bedingung)
                #=> hier gehoert key hin
                self.data[i] = HashNode(key, value)
                self.size += 1
                ... #eventuell muss hier die Kapazitaet optimiert werden
                return
            # index ist schon belegt => neuen Index durch 2. Hashfunktion berechnen
            i = (5 * i + 1 + h) % self.capacity
            h = h // 32
```

```python
def __getitem__(self, key):
    h = hash(key)
    i = h % self.capacity
```

```
    while True:
        if self.data[i] is None:    raise keyError(key)
        if self.data[i].key == key: return self.data[i].value
        i = (5 * i + 1 + h) % self.capacity
        h = h // 32
```

Komplexität des doppelten Hashings

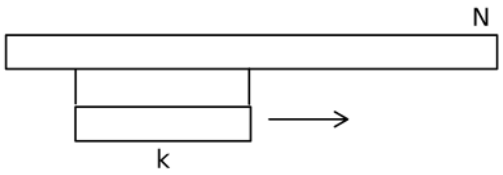
- mittlerer Füllstand $\alpha = \frac{N}{M} = \frac{\text{size}}{\text{capacity}}$
  - erfolglose Suche (Schlüssel nicht vorhanden) $\Omega(\frac{1}{1-\alpha})$
  - erfolgreiche Suche (Schlüssel vorhanden) $\Omega(\frac{1}{\alpha} ln(\frac{1}{1-\alpha}))$

α	0,5	0,9
erfolglos	2	10
erfolgreich	1,4	2,6

$\Rightarrow$ verdopple die Kapazität, wenn $\alpha = \frac{2}{3}$
(in Python: $M \in \{4, 8, 16, 32, ...\}$)
2. Hashfunktion sorgt für gleichmäßige Verteilung

Anwendung von Hashing zur Suche in Textdateien: Rabin-Karp-Algorithmus

- ähnlich zu gleitendem Mittelwert
  - naiv: für jede Fensterposition Mittelwert berechnen $\Rightarrow O(N * k)$
  - elegant: bei jeder Fensterposition rechts ein neues Element hinzufügen, links eins entfernen



- Suchwort der Länge k $\Rightarrow$ berechne gleitenden Hash für alle Abschnitte der Länge k des Text und vergleiche mit dem Hash des Suchworts

```
def rabin_karp_hash(u):
    h = 0
    d,q = 32, 33554393
    for i in u:
        h = (d * h + i) % q
    return h
```

```
def rabin_karp_update(c1, c2, k):
    h = (d * h + c2) % q          #rechts ein neues Element hinzufuegen
    h = (h - d ** k * c1) % q   #links eins entfernen
    return h
```

10 Rekursion

Definition: eine rekursive Funktion ruft sich selbst auf (evtl. indirekt)

Eigenschaften:

- jeder rekursive Aufruf hat seinen eigenen Speicher für alle lokalen Variablen

```
def f(n):
    r = f(n-1) + 1  # Rekursion
    ...
    return r
```

- jede rekursive Funktion muss mindestens einen nicht-rekursiven Zweig haben ≙ Basisfall bzw. Rekursionsabschluss (sonst: Endlosrekursion)
- jeder rekursive Aufruf muss nach endlich vielen Rekursionsstufen auf den Basisfall zurückgeführt werden
  Anzahl der Stufen bis zum Basisfall ≙ Rekursionstiefe
- jede Rekursion kann so umprogrammiert werden, dass stattdessen eine/mehrere Schleifen und ein Stack verwendet werden
  ⇒ Rekursion und Iteration sind gleich mächtig, welche Algorihmen man damit realisieren kann
  in der Praxis: Entscheidung, was effizienter und/oder lesbarer ist

Arten der Rekursion:

- lineare Rekursion: jeder Ausführungspfad (if: else:) enthält höchstens *einen* rekursiven Aufruf
  ⇒ Rekursionskette
- Baumrekursion: es gibt Ausführungspfade mit 2 oder mehr rekursiven Aufrufen
  ⇒ entsteht verzweigter Rekursionsbaum

Beispiel: Fibonacci-Zahlen 0, 1, 1, 2, 3, 5, 8, 13, ...

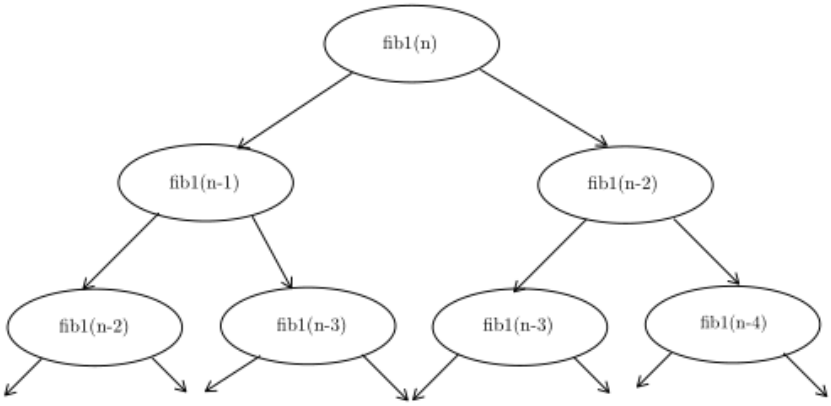
$$f_u = f_{u-1} + f_{u-2}; f_0 = 0, f_1 = 1$$

Aufgabe: Alg. zur Berechnung der n-ten Fibonacci-Zahl

Variante 1: naive Implementation der Definition:

```
def fib1(n):
    if n <= 1:
        return n
    else:
        return fib1(n-1) + fib1(n-2)
```

Nachteil: Baumrekursion



Baumtiefe: $O(n)$ ⇒ Anzahl der Knoten: $O(2^n)\widehat{=}$ Komplexität von fib1(n) ⇒ seeeehr langsam
Variante 2: Umwandlung der Rekursion in Iteration

Satz: Jede Rekursion kann mit Hilfe eines Stacks in eine Iteration umgewandelt werden.

```
def fib2(n):
    stack = [n]     #urspruengliches Problem in den Stack legen
    f = 0           #spaeter das Ergebnis
    while len(stack)>0:
        k = stack.pop() #oberstes Teilproblem
        if k <= 1:      #Rekursionsabschluss => Berechnung
            f += k
        else:           #Rekursion => neue Teilprobleme in den Stack
            stack.append(k-1)
            stack.append(k-2)
    return f
```

Anzahl der Iterationen = Anzahl rekursive Aufrufe in fib1(n)
⇒ Komplexität immer noch $O(2^n)$ ⇒ Umwandlung in Iteration allein verbessert nie die Komplexität

Variante 3: effizienter durch Vermeidung wiederholter Berechnungen der selben Fibonacci-Zahl: *course-of-values-Rekursion*

Aufruf für k läuft rekursiv nur von den Aufrufen für k-1, k-2, ..., k-c ab, für konstante c ⇒ trifft hier zu mit c = 2
⇒ man kann die Baumrekursion vermeiden, indem man die Zwischenergebnisse so lange speichert, bis sie nicht mehr benötigt werden.

Hier: speichere f_k, bis f_{k+2} berechnet ist

```python
def fib3(n):
    f_n_plus_1, f_n = fib3_impl(n)
    return f_n

def fib3_impl(n):
    if n == 0:
        return 1, 0
    else:
        f_n, f_m_minus_1 = fib3_impl(n-1)
        return f_n + f_n_minus_1, f_n
```

entscheidend: Hilfsfunktion ist linear rekursiv $\widehat{=}$ nur 1 rekursiver Aufruf
$\Rightarrow$ statt Rekusionsbaum in fib1(n) $\Rightarrow$ Rekusionskette
$\Rightarrow$ Komplexität $O(N)$, effizienter

Variante 4: Umwandeln in Endrekursion $\widehat{=}$ lineare Regression, wo der rekursive Aufruf der letzte Befehl vor dem *return* ist

Satz: Jede *Course-of-values-Rekursion* kann in Endrekursion umgeschrieben werden.

```python
def fib4(n):
    return fib4_impl(1, 0, n)   #f1, f0, gesuchte Fibonacci-Zahl

def fib4_impl(f_k, f_k_minus_1, counter):
    if counter == 0:            #Rek-Abschluss
        return f_k_minus_1
    else:
        return fib4_impl(f_k + f_k_minus_1, f_k, counter - 1)   #Endrekursion
```

Variante 5: Iterative Version ohne Stack

Satz: Jede endrekursive Funktion kann ohne Stack in Iteration umgewandelt werden.
Idee:

- Jeder rekursiveAufruf hat sein eigenes, privates Set lokaler Variablen
- ist der rekursive Aufruf der letzte Befehl, werden lokale Variablen *danach* nicht mehr benötigt

$\Rightarrow$ wir können die lokalen Variablen für den rekursiven Aufruf recyclen $\widehat{=}$ Überschreiben von Variablen in jeder Schleifeniteration

$\rightarrow$ Manche Programmiersprachen (LISP, SCHEME) machen diese Optimierung automatisch

```python
def fib5(n):
    f_k, f_k_minus_1 = 1, 0
    while n > 0:
```

```
        f_k, f_k_minus_1 = f_k + f_k_minus_1, f_k
        n -= 1
    return f_k_minus_1
```

n Iterationen $\Rightarrow O(n)$

Variante 6: (Hausaufgabe) Komplexität $O(logN)$

10.1 Umwandlung von Rekursion in Iteration

Beispiel für Umwandlung von Rekursion in Iteration mit Stacks

```
def tree_sort(node, a):     #Aufgabe: geg.Suchbaum, Schluessel in aufsteigender
↪  Reihenfolge auslesen und in Array a kopieren
    if node is None: return     #Rekursionsabschluss
    tree_sort(node.left, a)     # kleine Schluessel
    a.append(node.key)          # "mittleren Schluessel" einfuegen
    tree_sort(node.right, a)    #grosse Schluessel
```

in-order-trav...

```
def tree_sort_iterative(node, a):
    stack = []
    traverse_left(node, stack)
    while len(stack) > 0:
        current = stack.pop()
        a.append(current.key)
        traverse_left(current.right, stack)
```

```
def traverse_left(node, stack):
    while node is not None:
        stack.append(node)
        node = node.left
```

Komplexität: $O(2^P) = O(2^{logN}) = O(N)$ falls Baum balanciert

10.2 Komplexitätsberechnung rekursiver Algorithmen

$$T(n) = \underbrace{a_1 * T\left(\frac{N}{b_1}\right)}_{\text{Aufwand für } a_1 \text{ rekursive Teilprobleme der Größe N/}b_1} + \cdots + \underbrace{a_n * T\left(\frac{N}{b_n}\right)}_{a_n \text{ Teilprobleme der Größe } \frac{N}{b_n}} + \underbrace{f(N)}_{\text{Aufwand der aktuellen Fkt}}$$

fib1(n) : 1 Aufruf(n-1), 1 Aufruf(n-2) $\Rightarrow$ 2 Aufrufe $O(N)$

$$n_1 = 2, b_1 = 1, k = 1$$

Ausrechnen durch

- Substitutionsmethode („händisch“) $\Rightarrow$ später

10.2.1 Mastertheorem

- **Mastertheorem** (einsetzen): $T(n) = a * T\left(\frac{N}{b}\right) + f(n)$

  Definiere Rekursionsexponent: $\rho = log_b(a)$

  Fall 1: $f(N)$ sehr effizient: $f(N) \in O(N^{\rho-\epsilon}), \epsilon > 0$
  $\Rightarrow T(N) \in \Theta(N^{\rho})$Rekursion dominiert

  Fall 2: $f(N)$ so effizient wie Rekursion: $f(N) \in \Theta(N^{\rho})$
  $\Rightarrow T(N) \in \Theta(N^{\rho} * logN) \Rightarrow$ Rekursion und f(N) tragen bei

  Fall 3: $f(N)$ nicht so effizient: $f(N) \in \Omega(N^{\rho+\epsilon}), \epsilon > 0$
  $af\left(\frac{N}{b}\right) \leq c * f(N)$ mit $c < 1$
  $\Rightarrow T(N) \in \Theta(f(N)) \Rightarrow f(N)$ dominiert

- **Beispiel** Merge Sort: $T(N) = \underbrace{2T\left(\frac{N}{2}\right)}_{\text{rekursive Aufrufe für linke und rechte Hälfte}} + \underbrace{\Theta(N)}_{\text{Zusammenfügen der sortierten Hälften}}$

  $a = 2, b = 2$

  $$\rho = log_b(a) = log_2(2) = 1 \Rightarrow f(N) \in \Theta(N^2) = \Theta(N)$$

  $\Rightarrow$ Fall 2 $\quad T(N) \in \Theta(N^{\rho} \, log \, N) = \boxed{\Theta(N \, log(N))}$ wie bekannt

10.2.2 Substitutionsmethode

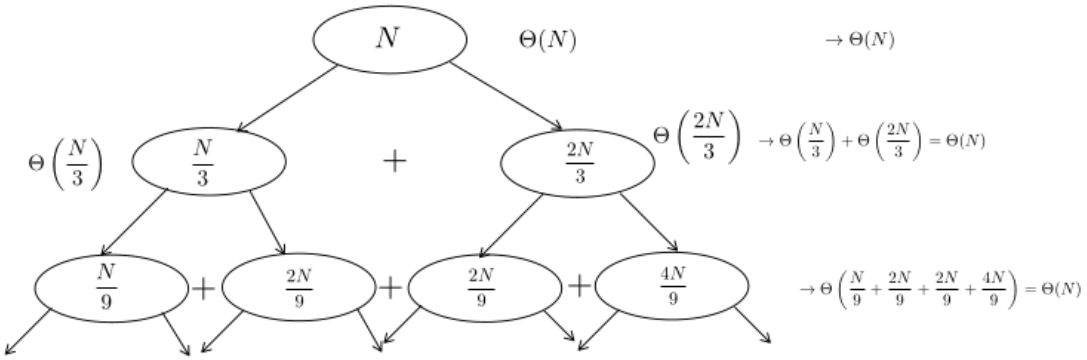
Händisches Berechnen mittels Substitutionsmethode

- **Beispiel**: Merge Sort, wo wir in zwei ungleich große Teilprobleme aufspalten

  $$T(N) = T\left(\frac{N}{3}\right) + T\left(\frac{2N}{3}\right) + \Theta(N)$$

  zeige jetzt: ungleiche Aufteilung ändert nicht die Komplexität

- **Rekursionsbaum**:



Berechnen des Aufwands der eigentlichen Berechnung in jedem Knoten ohne Rekursion
$\Rightarrow$ pro Ebene: $\Theta(N)$ $\Rightarrow$ Gesamtaufwand $\Theta(N * D)$ D ... Tiefe des Baums
Rekursion endet, wenn Knoten nur 1 Element enthält: $\left(\frac{2}{3}\right)^D * N = 1$

$$\Rightarrow\ D\ =\ log_{3/2}(N)\ =\ O(log\ N)$$

- Vermutung: $T(N)\ \in\ O(log_{3/2}(N)\ *\ c * N)\ =\ O\ (log\ N\ *\ N)$
  falls die Vermutung gilt, gilt für großes N:

$$T(N)\ \leq\ d\ *\ N\ *\ \underbrace{log_2(N)}_{ld}$$ Definition der O-Notation mit passender Konstante d

Einsetzen in Rekursionsformel

$$T(N)\ =\ T\left(\frac{N}{3}\right)\ +\ T\left(\frac{2N}{3}\right)\ +\ c * N\ \leq d\frac{N}{3}ld\left(\frac{N}{3}\right) + d\frac{2N}{3}ld\left(\frac{2N}{3}\right)\ +\ c * N$$

$$\leq d * \frac{N}{3}ld(N)\ -\ d\frac{N}{3}ld(3)\ + \frac{2N}{3}ld(N)\ -\ d\frac{2N}{3}\underbrace{ld(2)}_{=1}\ -d\frac{2N}{3}ld(3)\ +\ c * N$$

$$d\ N\ ld(N)\ \underbrace{-\ d\ N\ \left(ld(3)\ -\ \frac{2}{3}\right)\ +\ c * N}_{\text{sollte}\leq 0\text{sein}} - d\ \not{N}\ \left(ld(3)\ -\ \frac{2}{3}\right)\ +\ c*\ \not{N}\ \leq\ 0$$

mit: $d\ \left(\ ld(3)\ -\ \frac{2}{3}\right)\ +\ c*\ \not{N} \leq 0$ und $d\ \left(\ ld(3)\ -\ \frac{2}{3}\right)\ \geq c$ immer erreichbar, weil beliebig groß gewählt werden darf

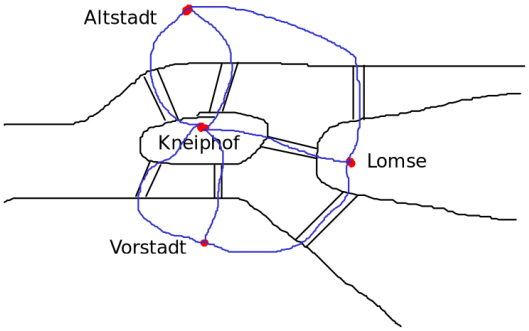
$\leq d\ N\ ld(N) \Rightarrow$ Vermutung bestätigt

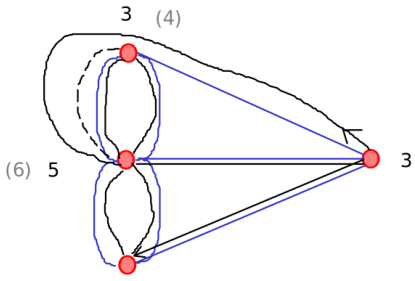
$\Rightarrow T(N)\ \in\ O(N\ log\ N)$ w. z. b. w.

11 Graphen und Graphenalgorithmen

Einführung

- Entstehung des Graphenformalismus: Königsberger Brückenproblem





Graph des Königsberger Brückenproblems

$G = (V, E)$
$E \subset V \times V$

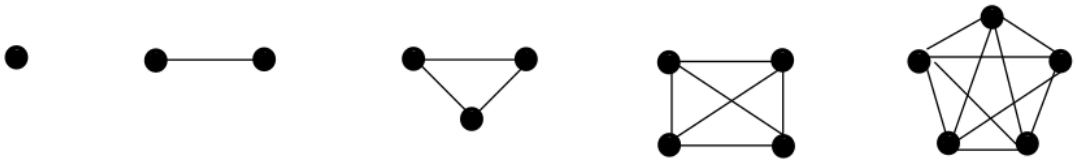
V ... Vertices, Knoten, E ... Edges, Kanten
Multigraph $\widehat{=}$ zwei Knoten können durch mehrere Kanten verbunden sein

- Antwort auf Königsberger Brückenproblem: „Kann man mit einem Spaziergang alle Brücken genau einmal überqueren?“ $\Rightarrow$ Nein
  - Anforderung: jeder Knoten (Stadtteil), der betreten wird, muss auch wieder verlassen werden $\Rightarrow$ zwei Kanten nötig, oder allg.: eine gerade Anzahl an Kanten
  - Ausnahmen: Start- und Endpunt des Spaziergangs, falls verschieden

11.2 Definitionen

- **Arten von Graphen**: ungerichtete und gerichtete
  - ungerichtet: $u, v \in V \qquad (u, v) \in E \Rightarrow (v, u) \in E$ Kanten ohne Richtung
  - gerichtet: $u, v \in V \qquad (u, v) \in E \Rightarrow (v, u)$ nicht notwendigerweise vorhanden, zeichne Kanten als **Pfeile**
- **Grad eines Knotens**:
  - ungerichtet: Anzahl der Kanten, die in einem Knoten anliegen, „degree“
    $deg(u) = |\{ v \mid (u, v) \in E \}| \Leftrightarrow (v, u) \in E$
  - gerichtet:
    - in-degree: Anzahl der Kanten, die im Knoten enden
    - out-degree: Anzahl der Kanten, die im Knoten beginnen
    - $in\_deg(u) = |\{ v \mid (v, u) \in E \}|$, $out\_deg(u) = |\{ u \mid (u, v) \in E \}|$

- vollständiger Graph: alle möglichen Kanten existieren auch, jeder Knoten ist mit jedem Knoten *direkt* verbunden



ungerichteter vollständiger Graph: $|E| = \frac{|V|\ (|V|\ -\ 1)}{2}$
Rätsel: jeder stößt auf Party mit jedem an, es macht 78 mal „pling“
Wie viele Gäste waren da? $|V| = 13$

- Subgraphen: $G' = (V', E')$ ist Subgraph von $G = (V, E)$, wenn :
  $V' \subseteq V$, $E' \subseteq E$ und für alle $(u, v) \in E'$ muss $(u, v) \in V'$

  Spezialfälle:
  - $V' = V$: aufspannender Teilgraph
  - lösche zuerst Knoten $V' \subset V$, lösche dann alle Kanten (u, v) wo u oder $v \notin V' \Rightarrow$ *induzierter Teilgraph*

- Wege in Graphen: rekursive Definition
  - ein einzelner Knoten $v \in V$ ist ein trivialer Weg der Länge 0
  - ist eine Folge $(v_1, \cdots, v_{k-1})$ ein Weg und die Kante $(v_{k-1}, v_k) \in E$ existiert, so ist $\underbrace{(v_1, \cdots v_{k-1}, v_k)}$ auch ein Weg

    Länge (k-1)$\widehat{=}$Zähle die Kanten im Weg

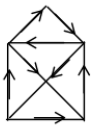
- Pfad: Weg, wo jeder Knoten *höchstens ein mal* vorkommt ($\widehat{=}$ keine Kreuzungen)

- Zyklus: Weg, wo $v_1 = v_k$ Anfangs- gleich Endknoten

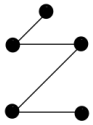
- Kreis: Zyklus ohne Kreuzung $(v_1, \ldots, v_{k-1}, v_k)$ ist Zyklus
  d.h. $v_1 = v_k$ und $(v_1, \ldots, v_{k-1})$ ist Pfad

- Erreichbarkeit $u \rightsquigarrow v$ „v ist erreichbar von u“$\Leftrightarrow$ es gibt einen Weg $(u, \ldots, v)$
  ungerichtet: $u \rightsquigarrow v \ \Rightarrow\ v \rightsquigarrow u$
  gerichtet: gilt das nicht unbedingt
  } Konsequenzen für Zusammenhangskomponenten (später)

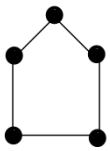
- **Eulerweg**: enthält jede Kante genau ein Mal
  (Brückenproblem: „Kante“ $\widehat{=}$ „Brücke“)
  Beispiel: Haus vom Nikolaus



- **Hamiltonweg**: Weg, der alle *Knoten* genau einmal enthält
  Bsp.: Haus vom Nikolaus:

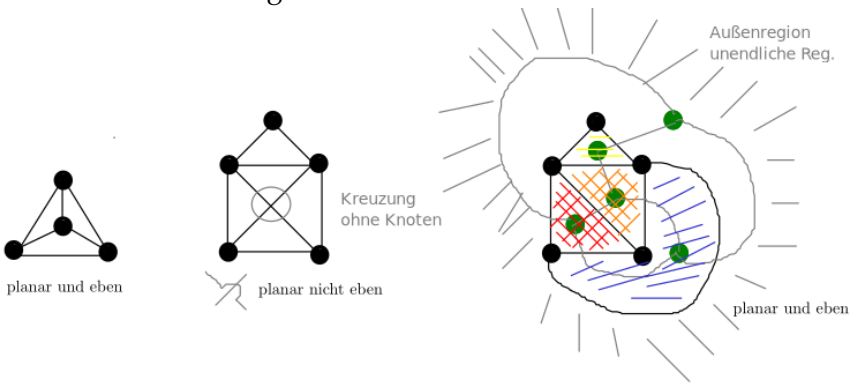


- **Hamiltonkreis**: Kreis, der alle Knoten genau einmal enthält (außer Anfang/Ende)
  Bsp.: Haus vom Nikolaus
  Beispiel: Problem des Handlungsreisenden: Suche den kürzesten Hamiltonkreis, der gegebene Städte verbindet ⇒ Prototypisches Problem für *NP-vollständig*

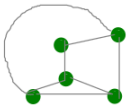


11.3 Planare Graphen, ebene Graphen

- planar: Graph kann ohne Überkreuzungen in der Ebene gezeichnet werden
- eben: wenn tatsächlich so gezeichnet



  - ebener Graph definiert eindeutig Flächen / Regionen in der Ebene



  - dualer Graph: jede Region ist ein dualer Knoten
    duale Knoten werden durch Kanten verbunden, wenn die Regionen eine gemeinsame Grenze haben (*nicht nur* gemeinsamen Knoten)

  Eulersche Gleichung: $|V| - |E| + |F| = 2$

- **Baum**: *zusammenhängender Graph*, der keine Zyklen enthält
  $\forall u, v : u \rightsquigarrow v \Rightarrow$ bei $|V|$ Knoten, $|E| = |V| - 1$ Kanten

- **Spannbaum**: Baum als Subgraph eines Graphen G, der alle Knoten enthält (zusammenhängend $\Leftrightarrow$ G auch zusammenhängend)

  - Problem: minimaler Spannbaum $\widehat{=}$ kürzeste Kanten ⇒ später

– wenn G nicht *zusammenhängend*: Wald (hat keine Zyklen) $\widehat{=}$ Menge von Spannbäumen (einer pro Zusammenhangskomponente)

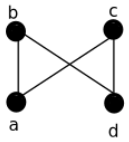
11.4 Repräsentation von Graphen

Adjazenzmatrix

$$A\,(\,|v|\,\times\,|v|\,)\,:\,a_{i,j}=\begin{cases}1 & \text{wenn}(u_i,u_j)\in E\\ 0 & \text{sonst}\end{cases}$$

- ungerichteter Graph: A immer symmetrische Matrix: $A = A^T$
  $a_{ij} = a_{ji}$ weil $(u_i, u_j) \in E \Rightarrow (u_j, u_i) \in E$
- gerichteter Graph: A im allgemeinen nicht symmetrisch
- Vorteil:
  - man kann in Graphalgorithmen Matrix-Funktionen benutzen
  - Speicher-effizient für dicht besetzte Graphen $|E| \in O(|V|^2)$
- Nachteil:
  - Speicherverschwendung für dünn besetzte Graphen
    viele $a_{i,j} = 0$ $|E| \in O(|V|)$ (z.B. planare Graphen)

Bsp.:



$$A=\underbrace{\begin{pmatrix}0&1&0&1\\1&0&1&0\\0&1&0&1\\1&0&1&0\end{pmatrix}}_{a,b,c,d} \qquad al = [[b,d],[a,c],[b,d],[a,c]]$$

Adjazenzlisten

Array von Arrays: ein Array pro Knoten enthält die Indizes der Nachbarn

- Vorteil:
  - speichereffizient für dünn besetzte Graphen ($a_{ij} = 0$ nicht explizit gespeichert), „dünn besetzte Matrixdarstellung“
  - elegante Schleife über alle Nachbarn und alle Kanten

```
for node in graph:      #graph = Adjazenzliste
    for neighbor in node:
        #verarbeiten Kante (node, neighbor)
```

- Nachteil:
  - manche Operationen umständlicher
  - man muss bei ungerichteten Graphen auf die Symmetrie achten

Transponierter Graph

- für alle Kanten: Richtung invertiert G^T
- bei ungerichteten Graphen: wieder der gleiche Graph
- Adjazenzmatrix: $A_{GT} = A_G^T$
- Adjazenzliste:

```python
def transpose_graph(graph):
    gt = [[] for u in graph ]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            gt[neighbor].append(node)
    return gt
```

11.5 Iterieren durch Graphen

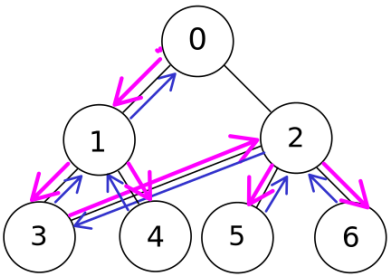
Traversieren von Graphen

- alle Knoten in einer sinnvollen Reihenfolge (genau einmal) besuchen
- Arten:
  - Tiefensuche (depth first search, DFS) ⇒ erst in die Tiefe, dann zu den anderen Nachbarn
  - Breitensuche (breadth first search, BFS) ⇒ erst zu allen Nachbarn, dann in die Tiefe

Tiefensuche – DFS

```python
def dfs(graph, startnode):          # graph: zusammenhaengenge Adjazenzliste
    visited = [False] * len(graph)  # Flags, welche Knoten schon besucht
    def visit(node):
        if not visited[node]:
            visited[node] = True
            print(node)                 # in der Praxis: anwendungsrelevante Rechnung
            for neighbor in graph[node]: # Adjazenzliste regelt Reihenfolge
                visit(neighbor)
            # print(node)
    visit(startnode)
```

```
dfs(graph, 0)
0   #5
1   #6
3   #2
2   #3
5   #4
6   #1
4   #0
=> disovery order / pre-order: verarbeite jeden Knoten VOR seinen Nachbarn -> Hinweg der Rekursion
# => finishing order / post-order: auf dem Rueckweg der Rekursion nach den Nachbarn
```



Anwendungen

pre-order traversal

- Graphen kopieren: kopiere erst den Knoten, dann seine Nachbarn und Kanten
- Zusammenhangskomponenten in Graphen finden
  (Variante von DFS ⇒ später)
- wenn Graph schon Baum ist: Abstand jedes Knotens von Wurzel
- Taschenrechner (Graph ist Parse tree):
  Drucken des Ausdrucks in Funktionsschreibweise add(3, mul(4,2)) [Operationsschreibweise: in-order Traversal 3 + (4 * 2)]

post-order traversal

- Graphen löschen: erst Nachbarn löschen, dann den Knoten
- Bestimme topologische Sortierung eines gerichteten Graphen (⇒ später)
- wenn Graph schon Baum ist: Abstand jedes Knotens von den Blättern
- Taschenrechner (Graph ist Parse tree):
  Auswertung / Berechnung des Ausdrucks

- Beides: Weg aus einem Labyrinth ⇒ Hausaufgabe
  pre-order: Vorwärtsweg ⇔ post-order: Backtracking aus Sackgassen

Für viele Anwendungen brauchen wir zusätzliche Informationen über den Graphen, die DFS sammeln kann: z.Zt. Flags *visited*
sinnvoll z.B.

- Reihenfolge der Pre- oder Postorder
- Elternknoten bei der DFS / von wo wurde node erreicht

⇒ property maps: Arrays, die für Knoten i und prop[i] die Info speichern

Tiefensuche mit property maps

```python
def dfs(graph, startnode):
    visited = [False] * len(graph)
    parents = [None] * len(graph)   #zusaetzliche property maps
    disovery_order = []             #
    finishing_order = []            #

    def visit(node, parent):
        if not visited[node]:
            visited[node] = True
            parents[node] = parent
            discovery_order_append(node)
            for neighbor in graph[node]:
                visit(neighbor, node)
            finishing_order.append(node)
    visit(startnode, None)
    return parents, discovery_order, finishing_order
    # Benutze in anderen Algorithmen
```

Am Beispiel des letzten Graphen: parents = [None, 0, 3, 1, 1, 2, 2]
0, 1, 2, 3, 4, 5, 6

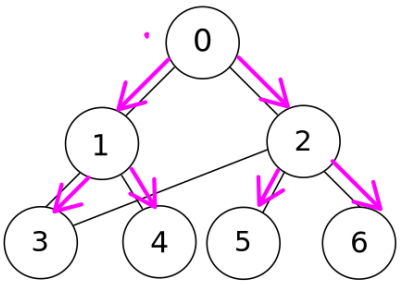
Konvention: Parent des Startnode ist der Startnode selbst ⇒ visited Array kann eingespart werden

Breitensuche

```python
    from collections import deque
    def bfs(graph, startnode):
        parents = [None] * len(graph)
        parents[startnode] = startnode
        q = deque()
        q.append(startnode)

        while len(q) > 0:
            node = q.popleft()                          #q.popright() => Tiefensuche
            print(node)
            for neighbor in graph[node]:
                if parents[neighbor] is None:   #neighbor node nicht besucht
                    parents[neighbor] = node
                    q.append(neighbor)
```

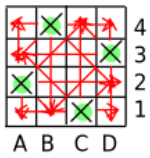
```
bfs(graph, 0)
0
1
2
3
4
5
6
=> level order nach Abstand vom startnode und von links
 ↪  nach rechts
```



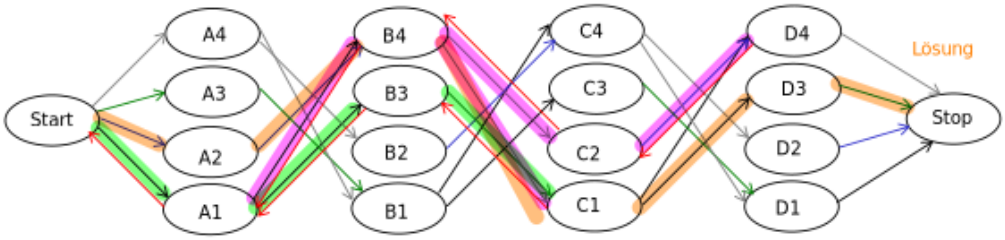
Anwendungen von Tiefensuche – Damenproblem (beim Schach)

Aufgabe: Platziere N Damen so auf einem NxN Schachbrett, dass sie sich nicht gegenseitig schlagen

Bsp.: N = 4 (echte Schachbretter: N = 8)



Graph repräsentiert Damenproblem

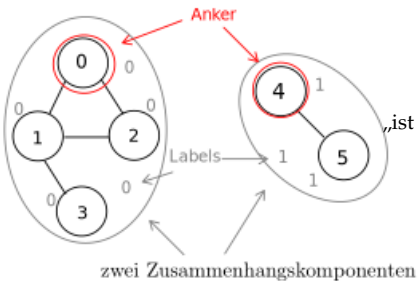


DFS beginnend bei Start

- checke in discovery_order, ob Damen sich bis jetzt nicht schlagen können
- wenn Test fehlschlägt ⇒ backtracking ≙ gehe zurück und probiere den nächsten Nachbarn
- beim Damenproblem: überprüfe das, indem man den Pfad durch das Parent-Array zurückgeht und den Test mit allen Knoten in dem Pfad ausführt
- das erfordert eine adaptive Version von DFS:
  - mit parent property map
  - Kompatibilitätstest der Damen statt print

Anwendungen der Tiefensuche – Zusammenhangskomponenten

- Bestimmen von Zusammenhangskomponenten: in unzusammenhängendem ungerichtetem Graph:



Definition:

1. $u \underbrace{\sim} v \Leftrightarrow u \underbrace{\rightsquigarrow} v$

„ist äquivalent" „ist in der gleichen Komponente" es gibt Pfad von u nach v

2. jede Komponente enthält alle von dort erreichbaren Knoten, ist also *maximal* zusammenhängender Subgraph, d.h. wenn man einen weiteren Knoten zum Subgraphen hinzufügt, wäre er nicht mehr zusammenhängend

- Idee des Algorithmus:
  1. – definiere für jede Komponente einen *Anker*: ausgezeichneter Knoten, der die Komponente repräsentiert
     – Konvention: der Knoten mit dem kleinsten Index
  2. starte Tiefensuche von jedem Anker ⇒ alle so erreichten Knoten gehören zur selben Komponente
  3. markiere jeden Knoten mit dem Label (laufende Nummer) der jeweiligen Komponente

```python
def connected_componends(graph):            #graph als Adjazenzliste
    anchors = [None] * len(graph)
    labels = [None] * len(graph)

    def visit(node, anchor):
        if anchors[node] is None:               #node noch nicht besucht
            anchors[node] = anchor                  #anker merken = node als visited
            ↪ markiert
            labels[node] = labels[anchor]
            for neighbor in graph[node]:
                visit(neighbor, anchor)
    current_label = 0                               #label der ersten Komponente
    for node in range(len(graph)):
        if anchors[node] is None:             #neuer Anker gefunden
            labels[node] = current_label
            visit(node, node)                       #Rekursiv alle Knoten der ZK von
            ↪ node besuchen
            current_label += 1
    return anchors, labels
```

Dieser Algorithmus verwendet das *Anlagerungsprinzip*

- starte von einem Knoten (anchor) und verbinde suksessive alle Knoten der Komponente $\widehat{=}$ wie ein Virus sich ausbreitet

Gegenteil: *Verschmelzungsprinzip* (⇒ später)

Test, ob ein zusammenhängender Graph ein Baum ist ($\widehat{=}$ ohne Zyklen) oder Zyklen hat

- Definition:

- Baum $\widehat{=}$ es gibt *genau einen* Weg von $u \rightsquigarrow v$ für jedes Paar $u, v \in V$
- alternative Wege sind nur bei Zyklen möglich

- Idee:
  - Tiefensuche: sobald ein Knoten zum zweiten Mal gefunden wird, gab es zwei alternative Wege ⇒ Zyklus
  - Ausnahme: trivialer Zyklus parent → node → parent darf nicht gewertet werden ⇒ Modifikation der Tiefensuche

```python
def undirected_cycle_test(graph):       #graph = zusammenhaengende Adjazenzliste
    visited = [False] * len(graph)
    def visit(node, parent):
        if not visited[node]:
            visited[node] = True
            for neighbor in graph[node]:
                if neighbor == parent:
                    continue                    #trivialen Zyklus ueberspringen
                if visit(neighbor, node):
                    #returns True wenn rekursiv Zyklus gefunden wurde
                    return True
                return False                    #bei "node" kein Zyklus gefunden
        else:
            return True                         #node zum zweiten Mal besucht => Zyklus
    startnode = 0
    return visit(startnode, startnode)
```

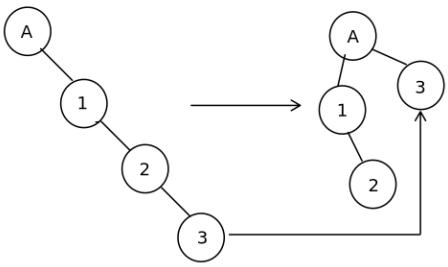
Alternativer Algorithmus für Zusammenhangskomponenten: Union-Find

- wie zuvor: der kleinste Index jeder ZK ist der Anker
- aber: Verschmelzungsprinzip:
  - anfangs ist jeder Knoten eine Komponente
  - Komponenten schließen sich sukzessive mit ihren Nachbarn zusammen
  - wenn kein Zusammenschließen mehr möglich ist ⇒ Komponenten sind maximal, also ZK des Graphen
- Hilfsfunktion für find-Schritt (Variante 1):
  gegeben: node und property map anchors ⇒ finde den Anker von node

```python
def find_anchor1(anchors, node):
    while node != anchors[node]:    # Konvention: Anker zeigt in anchors auf sich selbst
        node = anchors[node]        # Kette bis zum Anker verfolgen
    return node                     # jetzt der Anker
```

- Verbesserte Variante 2: Pfadkompression ≙ Verbinde jeden Knoten direkt mit dem Anker ≙ mache aus der Kette einen Stern

```python
def find_anchor(anchor, node):
    start = node
    while node != anchors[node]:
        node = anchors[node]
    anchors[start] = node    #Direktverbindung
    ↪  start->Anker
    return node
```



Idee:

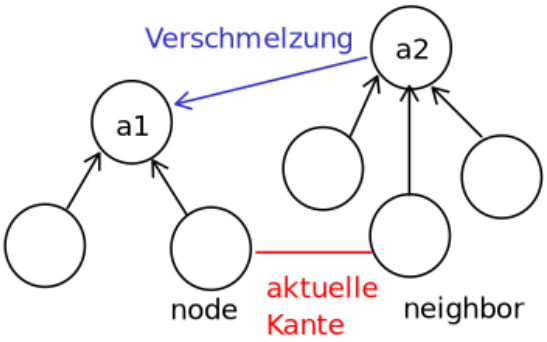
- Anfangs ist jeder Knoten ein Anker
- Iteriere über alle Kanten und verschmelze die Endpunkte (falls sie noch in unterschiedlichen Komponenten sind)

⇒ wenn alle Kanten abgearbeitet sind ⇒ ZK gefunden

- Konvention: beim Verschmelzen von zwei Komponenten wird der Anker mit kleinerem Index zum gemeinsamen Anker
- betrachte Kanten nur in der Richtung kleiner Index → großer Index

```python
def union_find(graph):
    anchors = list(range(len(graph))    #Anfangszustand: jeder Knoten ist Anker
    ↪  anchors[node] == node
    #Komponenten finden
    for node in range(len(graph)):
        for neighbor in graph[node]:
            if neighbor < node: continue    #falsche Kantenrichtung ueberspringen
            a1 = find_anchor(anchors, node)
            a2 = find_anchor(anchors, neighbor)
            if a1 < a2: anchors[a2] = a1
            else: anchors[a1] = a2

    #labels zuweisen
    labels = [None] * len(graph)
    current_label = 0
    for node in range(len(graph)):
        a = find_anchor(anchors, node)
        if a == node:                                    #node ist anchor
            label[a] = current_label
            current_label += 1
        else:
            labels[node] = labels[a]
    return anchors, labels
```



Anwendung von Breitensuche – kürzeste Wege

- ungewichtete Graphen: Länge des Wegen $\widehat{=}$ Anzahl der Kanten
  (Gegensatz: gewichtete Graphen: jede Kante hat individuelle Länge)
- Idee: property map parents: zeigt für jeden Knoten an, woher man kommt
  $\Rightarrow$ Rückverfolgung der Kette parents[target] $\rightarrow$ startnode $\widehat{=}$ kürzester Pfad

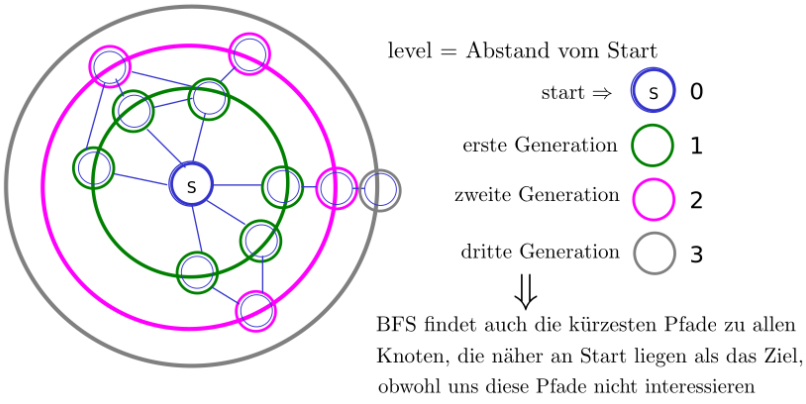
```
1  from collections import deque
2
3  def shortest_path(graph, start, target):
4      parents = [None] * len(graph)
5      parents[start] = start
6      q = deque()
7      q.append(start)
8      while len(q) > 0:
9          node = q.popleft()
10         if node == target:
11             break               # target gefunden => Schleife beenden
12         for neighbor in graph[node]:
13             if parents[neighbor] is None:
14                 parents[neighbor] = node
15                 q.append(neighbor)
16     if parents[target] is None:
17         return None     #es gibt keinen Pfad
18     #pfad zurueckverfolgen
19     pfad = [target]
20     while pfad[-1] != start:
21         pfad.append(parents[pfad[-1]])
22     # Pfad target -> start in start->target umwandeln
23     pfad.reverse()
24     return pfad
```

Warum findet Breitensuche den kürzesten Weg?

- BFS besucht Knoten in level-order, als? nach Abstand vom Start

⇒ Wellenförmige Ausbreitung vom Start

⇒ sobald die Welle das Ziel erreicht, haben wir den kürzesten Pfad gefunden



11.6 Gewichtete Graphen

- Knotengewichteter Graph: jedem Knoten ist eine Zahl (reell oder ganz) zugeordnet
- Kantengewichteter Graph: jeder Kante ist eine Zahl (reell oder ganz) zugeordnet (gerichtete Graphen: hin- und Rückkante haben im Allgemeinen verschiedene Gewichte)
- oder beides gleichzeitig

Beispiele für kantengewichtete Graphen

- Straßennetzwerke: Gewichte sind Entfernungen oder Fahrzeiten, im Allgemeinen gerichtete Graphen: Einbahnstraßen, Berge
- Wechselkurse: (Knoten sind Währungen)
- elektrische Netzwerke

Repräsentation der Gewichte

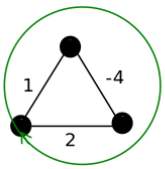
- Adjazenzmatrix: $a_{i,j} \in \{0, 1\} \Rightarrow a_{i,j} = w_{i,j} \qquad (w_{i,j} = 0 \Leftrightarrow (u_i, v_i) \notin E)$
- Property Maps: weights[(i, j)] = $w_{i,j}$

Definition: Kürzester Weg in gerichteten Graphen

- Länge des Weges: [start = $u_0, u_1, \ldots, u_{k-1}, u_k$ = target] = P
  länge(p) = $\sum_{i=1}^{k} w_{i-1,i}$
- kürzester Weg: P(start, target) $\widehat{=}$ Menge aller Wege mit u_0 = start, u_u = target, k beliebig
  p^* = arg $\min_{p \in P(\text{start, target})}$ länge(p)

Das Problem unterscheidet sich, wenn es nur positive Gewichte oder positive und negative Gewichte gibt.

Schwierig ist der Fall, wenn es Zyklen negativer Länge gibt
⇒ Bellmann-Ford Algorithmus: bricht Suche ab, wenn negativer Zyklus gefunden, sonst findet er den kürzesten Pfad

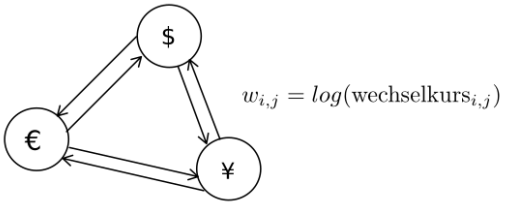


länge(p) = -1

$\Rightarrow$ der kürzeste Pfad durchläuft Zyklus unendlich oft

$\Rightarrow$ Gesamtlänge ist $-\infty$

- Vorteil: negative Gewichte erlaubt
- Nachteil: langsamer
- Beispiel: Arbitrage-Geschäfte



- wenn alle $w_{i,j} > 0$: *Algorithmus von Dijkstra* , Komplexität $O(|E| * log|V|)$
  Idee: Verwende statt einer Queue (in BFS) eine priority queue
  $\Rightarrow$ expandiere kurze Wege zuerst

```python
import heapq
# Konvention: MinHeap ist ein Python-Array (list),
# Elemente sind Tupel(priority, data1, data2, ...) (Anwendungsdaten)

def dijkstra(graph, weights, start, target):
    parents = [None] * len(graph)
    q = []
    #Heap statt Deque
    heapq.heappush[q, (0.0, start, start))        # Prioritaet, aktueller Knoten,
    ↪  parent
    while len(q) > 0:
        length, node, parent = heapq.heappop(q)
        # Knoten nicht zweimal besuchen
        if parents[node] is not None:
            continue  # wir kennen bereits kuerzeren Weg
        parents[node] = parent
        if node == target: break                  # Ziel gefunden
        for neighbor in graph[node]:
            # kuerzester Weg zu neighbor noch nicht bekannt
            if parents[neighbor] is None:
            # Prioritaet / Laenge berechnen
            new_length = length + weight[(node, neighbor)]
            heapq.heappush(q, (new_length, neighbor, node))
    if parents[target] is None: return None, None
    path = [target]
    while path[-1] != start:
    path.append(parents[path[-1]])
    path.reverse()
    return path, length
```

Komplexität vom Dijkstra-Algorithmus

- while len(q) > 0:
  ... #1 #heappop(q): $O(log(len(q)))$
  if parents[node] is not None: continue ⇐ jeder Knoten wird höchstens einmal expandiert
  parents[node] = parent
  ... #2 #für jeden Knoten höchstens 1-mal

- jede Kante kann höchstens zweimal gefunden werden: (u,v) und (v, u), weil jeder Anfangsknoten nur einmal expandiert
  tatsächlich wird *jede Kante nur einmal gefunden*, weil wir Kanten überspringen, deren Endknoten schon expandiert wurde

⇒ im Heap liegen Kanten, d.h. len(q) ∈ $O(|E|)$
Komplexität des Heap-Zugriffs: $O(log|E|)$

- in gewöhnlichen Graphen (zw. zwei Knoten höchstens eine Kante): $|E| \in O(|V|^2)$

- max $|E|$ Durchläufe durch die while-Schleife

⇒ insgesamt: Komplexität $O(|E| * log|E|) = O(|E|log|V|^2) = O(|E|log|V|)$

Korrektheit: Findet Dijkstra wirklich den kürzesten Pfad?

Beobachtung: length wird in der nächsten Iteration der while-Schleife nie kürzer

$$length_{i-1} \geq length_i$$

Beweis:(indirekt) Angenommen, $l_{i+1} < l_i$ und l_{i+1} ist Top-Element in Iteration i

- Fall 1: Der Weg der Länge l_{i+1} war schon in Iteration i bekannt ⇒ l_i war nicht Top in Iteration i⇒ w!

- Fall2: Der Weg der Länge l_{i+1} wurde in Iteration i entdeckt. $l_{i+1} = l_i + w_{uv} > l_i \Rightarrow w$!

Korrektheitsbeweis(indirekt):

- Annahme: Dijkstra: node → parent ⇝ start l
  wirklicher kürzester Weg: node → x ⇝ start $l' < l$
  In Iteration i war node → parent das Top-Element des Heaps, aber

- Fall 1: x ⇝ start ist auch schon im Heap
  wenn x ⇝ start kürzer ist als node ⇝ start, hätte er schon früher Top-Element sein müssen → w!

- Fall 2: x ⇝ start war noch nicht im Heap ⇒ länge (x ⇝ start) ist wegen der Monotonie von length nicht kürzer als l
  also ist l'= length (node → x ⇝ start) = $\underbrace{\text{length}(x \rightsquigarrow \text{start})}_{>l \Rightarrow w!} + \underbrace{w_{x,node}}_{>0}$

Induktiver Beweis für alle Iterationen:

Induktionsanfang: Weg start $\rightarrow$ start $\widehat{=}$ Länge 0 $\Rightarrow$ kürzester Pfad, Fall target = start

Induktionsschritt: wir kennen den kürzesten Weg zu allen Knoten mit Länge $\leq l$ (= Länge (node $\rightarrow$ parent $\rightsquigarrow$ start)
dann ist das nächste Top-Element (node $\rightarrow$ parent $\rightsquigarrow$ start) der kürzeste Weg für node $\rightsquigarrow$ start (s.o.)

Induktionsende:Sobald das Top-Element (target $\rightarrow$ parent $\rightsquigarrow$ start) ist haben wir den kürzesten Weg target $\rightsquigarrow$ start gefunden

Konsequenz: Dijkstra findet auch alle kürzesten Wege, die kürzer als target $\rightsquigarrow$ start sind (wie Breitensuche), kann ineffizient sein

Bsp.: Weg von Frankfurt(Main) $\rightsquigarrow$ Dresden (460 km)

findet auch die kürzesten Wege Frankfurt $\left.\begin{array}{l}\rightsquigarrow \text{Stuttgart(210 km)}\\ \rightsquigarrow \text{Dortmund(220 km)}\end{array}\right\}$ ignorieren

aber: kürzesten Wege Frankfurt $\left.\begin{array}{l}\rightsquigarrow \text{Erfurt (260km)}\\ \rightsquigarrow \text{Suhl (210km)}\end{array}\right\}$ nicht ignorieren
könnten Teil des Weges Fr $\rightsquigarrow$ Dresden sein

Wie entscheiden wir, welche Wege ignoriert werden dürfen?

A* – Algorithmus

- benötigt Schätzfunktion für die Restentfernung guess(Zwischenziel, target)
- Trick: ändere Prioritaet von length nach length + guess
- Voraussetzung: length(Zwischenziel, target) $\geq$ guess(Zwischenziel, target)
  Dann ist garantiert, dass immernoch der Korrekte kürzeste Pfad gefunden wird:
  - nur erfüllbar, wenn 0 = length(target, target) = guess(target, target)
    $\Rightarrow$ Target wird mit der gleichen Länge zum Top-Element wie bei Dijkstra
  - für alle Zwischenziele auf dem wahren kürzesten Weg gilt:
    length(zwischen, start) + guess(zwischen, target) $\leq$ length(target, start)
    $\Rightarrow$ alle Zwischenziele waren Top-Elemente vor target und wurden bereits expandiert $\widehat{=}$ man ignoriert nie die korrekten Zwischenziele
  - aber: Zwischenziele mit length(zwischen, start) + guess(zwischen, target) > length(target, start) werden ignoriert, $\Rightarrow$ A^* effizienter als Dijkstra

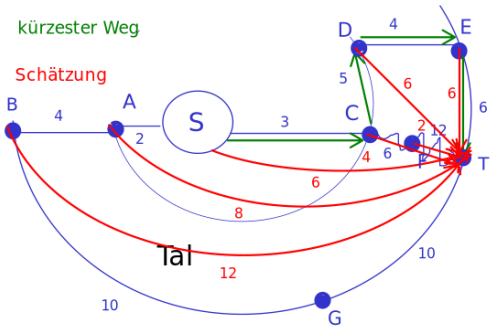
Beispiel für kürzester Weg mit Dijkstra und A*

Minimaler Spannbaum

(minimum spanning tree, MST)

Dijkstra

Prio	Pfad
0	S
2	SA
3	SC
6	SAB
8	SCB
9	SCF
12	SCDE
16	SABC
18	SCD
21	SCFT
26	SABGT



A*

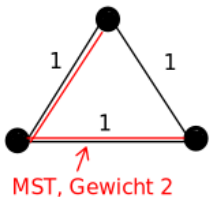
Prio	Pfad
6 = 0 + 6	S
7 = 3 + 4	SC
10 = 2 + 8	SA
11 = 9 + 2	SCF
14 = 8 + 6	SCD
18 = 12 + 6	SCDE
18 = 6 + 12	SAB
21 = 21 + 0	SCFT
18 = 18 + 0	SCDET

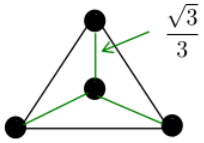
Definition: gewichteter ungerichteter Graph $G = (V, E, w)$ zusammenhängend

- gesucht: $G' = (V, E' \subset E, w' \subset w)$, so dass $\sum_{l=E'} we' \rightarrow$ *minimal* und G' zusammenhängend
- „Spann“: V' = V
- „Baum“: G' ist immer ein Baum. Hätte G' ein Zyklus, könnten wir eine Kante im Zyklus löschen, ohne den Zusammenhang zu stören, und dabei die Summe $\sum_{l=E'} we'$ verringern

Variante: ist G nicht zusammenhängend
⇒ minimaler Spannbaum für jede Komponente ⇒ Wald von minimalen Bäumen

Beispiel:

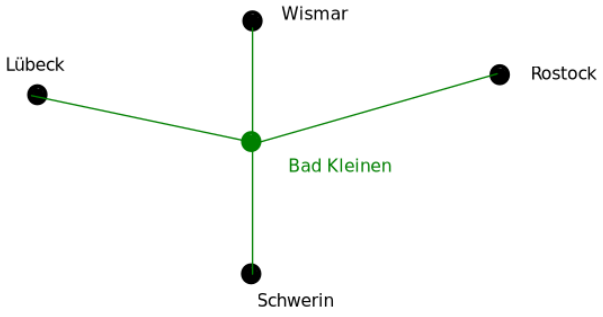




MST mit zusätzlichem Punkt: Gewicht $\sqrt{3} \approx 1.7 < 2$

Beobachtung: Wenn man neue Knoten hinzufügen darf, kann der neue Spannbaum kürzer sein als der alte („Steiner Punkte“)

Anwendung: Eisenbahnknoten



Im MST-Problem ist das Hinzufügen neuer Punkte nicht erlaubt.

Algorithmen: Prim(Anlagerungsprinzip), Kruskal (Verschmelzungsprinzip)

Prim:

- starte mit einem beliebigen Knoten und füge den Knoten mit der kürzesten Kante hinzu, solange dadurch kein Zyklus entsteht, andernfalls ignoriere die Kante
- Algorithmus entspricht Tiefensuche und Breitensuche
  Datentruktur: Stack(letzten expandieren) und Queue(ältesten expandieren), Prim: Heap(nächsten Knoten expandieren)

```python
import heapq

def prim(graph, weights):                    #Adjazenzliste, Property map
    parents = [None] * len(graph)
    q = []
    heapq.heappush(q, (0.0, 0, 0))      #Prioritaet, start, parent
    sum = 0.0
    while len(q) > 0:
        w, node, parent = heapq.heappop(q)
        # Knoten zweimal besuchen = Zyklus => ueberspringen
        if parents[node] is not None:
            continue
        parents[node] = parent
        sum += w
        for neighbor in graph[node]:
            if parents[neighbor] is None:
                heapq.heappush(q, (weights[(node, neighbor)], neighbor,
                ↪  node))#prio,node,parent
    return parents, sum
```

Kruskal:

- Anfangs ist jeder Knoten ein Baum für sich, in jeder Iteration werden die Teilbäume mit der kürzesten Kante verschmolzen, beachte:
  dabei nur Kanten benutzen, deren Enden in verschiedenen Bäumen liegen, die anderen werden übersprungen, weil Zyklus entstehen würde
- zweckmäßig: Kanten anfangs nach Priorität sortieren

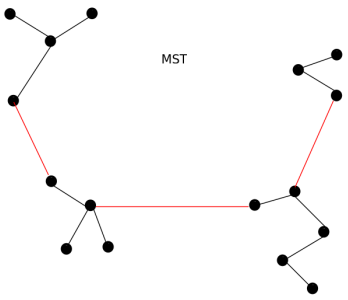
```python
def kruskal(graph, weights):
    anchors = list(range(len(graph)))   # wie bei union-find
    results = []                        # enthaelt spaeter die Kanten des Baums
    q = []
    sum = 0.0
    for edge, w in weights.items():     # edge:(u,v)
        heap.heappush(q, (w, edge))
    while len(q) > 0:
        w, edge = heapq.heappop(q)
        a1 = find_anchor(anchors, edge[0])
        a2 = find_anchor(anchors, edge[1])
        #if w > w_max: break    => Clusterung
```

```
13            if a1 != a2:                    # u und v in verschiedenen Teilbaeumen
14                anchors[a2] = a1            # Teilbaeume verschmelzen
15                result.append(edge)
16                sum += w
17        return results, sum
```

Komplexität: Heap enthält maximal $|E|$ Elemente $\Rightarrow$ Zugriff $O(log|E|)$
insgesamt: $O(|E| log|E|) = O(|E| log|V|)$ weil $|E| \in O(|V|)$

Anwendung von Kruskal-Alg. zur Bestimmung von Datenclustern

vollständiger Graph mit $w_{u,v}$ = dist(u, v)



im MST gibt es zwei Arten von Kanten:

- kurze $\widehat{=}$ innerhalb der Cluster
- lange $\widehat{=}$ zwischen Clustern

$\Rightarrow$ lange Kanten löschen

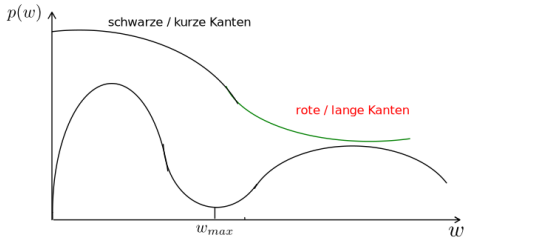
$\Rightarrow$ unzusammenhängender Graph $\Rightarrow$ Zusammenhangskomponenten sind Cluster $\widehat{=}$ Gruppen von Knoten, die sich nahe sind, während die Clusterzentren größeren Abstand haben

mit Kruskel: neuen Funktionsparameter w_{max} übergeben, so dass

$w \leq w_{max} \widehat{=}$ „kurz“, $w > w_{max} \widehat{=}$ „lang“

Schleife über sortiere Kanten abbrechen, sobald $w > w_{max}$

- „Single linkage Clustering“
- schwierig: Bestimmung des richtigen w_{max}



Was ist w_{max} hier?

11.7 Algorithmen für gerichtete Graphen

Anwendungen gerichteter Graphen:

- Straßenbahnnetzwerke mit Einbahnstraßen und/oder unterschiedlichen Entfernungen / Fahrzeiten für hin vs. zurück
- Abhängigkeitsgraphen:

  - welche Aktion muss man vor einer anderen Aktion ausführen
  - zeitliche Beziehung past → present → future
  - kausale Abhängigkeiten Ursache → Wirkung
  - Softwareabhängigkeiten: Python-Modul import json importiert zuerst *decimal*, das wiederum *copy* und *collections*, danach *json.encoder* und *json.decoder*, die wiederum *re* und *sys*
- Internet: Hyperlinks sind gerichtet

Anwendung: sequence alignment

- verschiedene Sequenzen des gleichen Phänomens, die nicht exakt gleich ablaufen, so gegeneinander verschieben / skalieren (linear alignment), dass korrespondierende Ereignisse übereinander liegen
- oder nicht-linear verzerren
- Bsp.: EKG

Anwendung von sequence alignment : *edit distance*

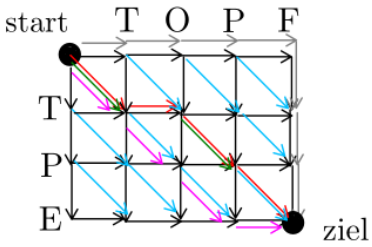
- Schreibprogramm mit Rechtschreibprüfung, getippt wurde „TPE“
- Welches Wort könnte gemeint sein (zur Anzeige im Kontextmenü)?
- Welches Wort ist ähnlich? ⇒ Wie viele Editieroperationen sind nötig, um „TPE“ in ein sinnvolles Wort zu überführen? *edit distance*
- zwei erlaubte Operationen:
  - ein Jokerzeichen einfügen (Kosten a)
  - einen Buchstaben tauschen (Kosten b)
- sinnvolles Wort: „TOPF“, mögliche Alignments:

T	O	P	F	T	O	P	F	?	?	?	T	O	P	F
T	?	P	E	?	?	?	?	T	P	E	T	P	E	?
Kosten a + b				7 * a							a + 2*b			

usw.edit dist $\widehat{=}$ minimum der Kosten

Aufgabe: finde sequence alignment, das die Kosten minimiert.

Lösung: Dijkstra-Algorithmus auf gerichtete und gewichtete Graphen



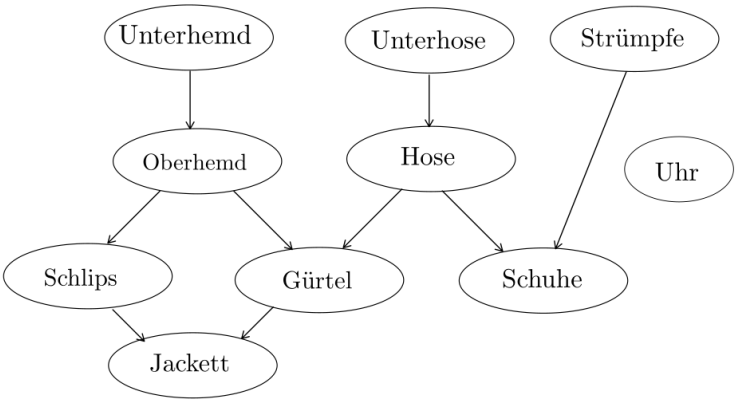
Bedeutung der Kanten:

- → Joker im linken Wort einfügen
- ↓ Joker im rechten Wort einfügen (Kosten a)
- ↘ zwei Buchstaben matchen und passt (Kosten 0)
- ↘ zwei Buchstaben matchen und tauschen (Kosten b)

Optimale Lösung: kürzester Weg Start → Ziel

Anwendung: Abhängigkeitsgraph

Welche Aktion muss man vor einer anderen Aktion ausführen?



Azyklischer gerichteter Graph (Zyklus $\widehat{=}$ Anziehen wäre unmöglich)
DAG: directed acyclic graph

Satz: Jeder DAG definiert eine Halbordnung:

$x \leq y$ (Reflexivität)

$x \leq y \wedge y \leq x \Rightarrow x = y$ (Antisymmetrie)

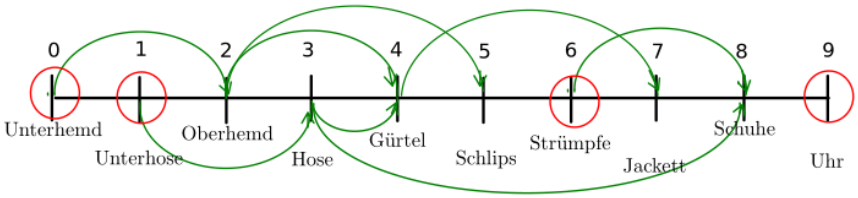
$x \leq y \wedge y \leq z \Rightarrow x \leq z$ (Transitivität)

bei Totalordnung zusätzlich: $x \leq y$ oder $y \leq z$ ist wahr, es gibt Kante (x,y) oder (y, z)

bei Halbordnung: $x \leq y \Rightarrow$ „unknown“, falls x und y nicht vergleichbar $\widehat{=}$ keine Kante (x, y) oder (y, z)

Aufgabe: *Topologische Sortierung*, d.h. Totalordnung, die die Halbordnung enthält

Idee der topologischen Sortierung ein DAG: Weise jedem Knoten eine Zahl $0, 1, \ldots, N-1$ zu, sodass die Ordnung dieser Zahlen die Halbordnung enthält, d.h. wenn wir den Graphen auf eine Gerade zeichnen, gehen alle Kanten nach rechts (falls (x, y) als $x \leq y$ true interpretiert wird)



Beobachtungen:

- es gibt viele erlaubte Totalordnungen
- wichtig sind die Knoten *ohne* eingehende Pfeile (0, 1, 6, 9)

Lösung 1:

1. Suche Knoten mit Eingangsgrad 0 und setze ihne an die nächste freie Position
2. entferne diesen Knoten aus dem Graphen inkl. seiner ausgehenden Kanten
   ⇒ dadurch verringert sich der Eingangsgrad seiner Kinder
3. gehe zu 1.

   wenn es keine Knoten mit Eingangsgrad 0 mehr gibt, aber noch nicht alle Knoten platziert wurden, ist der Graph zyklisch ⇒ keine topologische Sortierung möglich

```python
def topological_sort(graph):
    in_degree = [0] * len(graph)
    for node in range(len(graph)):
        for neighbor in graph[node]:
            in_degree[neighbor] += 1
    result = [] #result[node] -> Position von node auf der Geraden
    for node in range(len(graph)):
        if in_degree [node] == 0:
            result.append(node)
    k = 0
    while k < len(result):
        node = result[k]
        k += 1
        for neighbor in graph[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                result.append(neighbor)
    if len(result) == len(graph): # alle Knoten eingefuegt
        return result
    else:                         # Zyklus
        return None
```

Lösung 2: die reverse post-order (finishing order von Tiefensuche rückwärts) eines DAGs ist eine topologische Sortierung

```python
def reverse_post_order(graph):
    result = []
    visited = [False] * len(graph)

    def visit(node):
        if not visited[node]:
            visited[node] = True
            for neighbor in graph[node]:
                visit(neighbor)
            result.append(node)
    for node in range(len(graph)):
        visit(node)
    result.reverse()
    return result
```

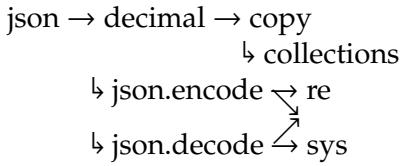
Algorithmus gibt die richtige Lösung, wenn graph ein DAG, sonst eine bestimmte Reihenfolge, die keine topologische Sortierung ist
⇒ Erweiterung des Algorithmus nötig, wenn Zyklen möglich sind (siehe Skript) (z.B. return None)

- Pre-order ist keine topologische Sortierung!

Zusammenhangskomponenten von gerichteten Graphen

Zwei Arten:

1. $v \in$ weak_comp(u), falls es einen Pfad $u \rightsquigarrow v$ gibt, aber nicht notwendigerweise auch $v \rightsquigarrow u$:
   transitive Hülle von u $\widehat{=}$ alle Knoten, die von u erreichbar sind
   Alg.: Tiefensuche / Breitensuche von u aus
   Anwendung: z.B. Abhängigkeit von Python-Modulen

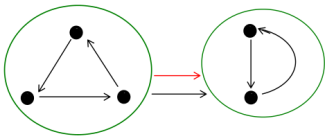
   Transitive Hülle von json $\widehat{=}$ alle Module, die pip oder conda auch installieren müssen, wenn „pip install json“

2. $v \in$ strong_comp(u) falls Pfade $u \rightsquigarrow v$ und $v \rightsquigarrow u$ existieren

Anmerkung: in ungerichteten Graphen gibt es diese Unterscheidung nicht, weil der Pfad $u \rightsquigarrow v$ immer rückwärts als Pfad $v \rightsquigarrow u$ existiert

Anwendung:

- starke Zusammenhangskomponenten gibt es nur, wenn der Graph zyklisch ist (sonst ist jeder Knoten eine starke Komponente für sich)
- definiere „meta graphen“, wo jede starke Komponente ein Knoten ist ⇒ DAG



Algorithmus von Kosaraju

1. Bestimme die reverse post-order (Alg. siehe oben)
2. Bilde den transponierten Graphen G^T (transpose_graph() $\Rightarrow$ erste VL über Graphen)
3. Bestimme die Zusammenhangskomponenten von G^T mittels Tiefensuche ($\widehat{=}$ transitive Hülle der Anker), aber *nicht* in der Reihenfolge der Knotenindizes, sondern in reverse post-order [for node in range(len(graph)): $\Rightarrow$ for node in rev_post_order:]