Pseudocode

```
Algorithmus nellosAlter(paramter...)
//Was macht die Methode //Eingabe
//Ausgabe
if parameter[0] ≠ 42
    alter <- 0
    alterCheck <- 0
    for i <- 1 to parameter[0] do alter <- alter + 1
    while alterCheck < parameter[0] do
                                alterCheck <- + 1
                            if alterCheck = alter
     return "Nello ist " + alter
else
    Return "Nello ist 42"
```

ADT

Name	
Schnittstelle	Interface: Bsp.: new() isEmpty()
Invariante	Stets geltende Eigenschaft (Alle Elemente sind immer sortiert/stabil)

Conditions der Funktionen:
New():
Precondition: -
Postcondition: $Q_{post} = \emptyset$

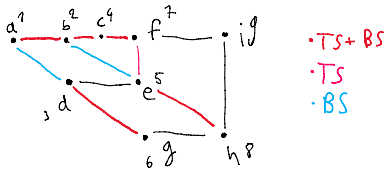
isEmpty(Q):
Precondition: -
Postcondition: -
Resultat: Q = ∅ ?

Grapheneigenschaften:

ungerichtet	Kanten sind Linien
gerichtet	Kanten sind Pfeile
Schleife	Kante auf sich selbst
Zyklus	Pfad von einer Ecke zu sich selbst (keine Kante doppelt)
Kreis	Zyklus und kein Knoten doppelt
Mehrfachkanten	Nicht erlaubt. Erweiterung -> Multigraph
Kantendichte	vollständig: Alle erlaubten Kanten da dicht: Es fehlen nur wenige Kanten licht: Es gibt nur wenige Kanten
gewichtet	Zahlenwerte zu Kanten
Zusammenhangs-komponenten	Maximalgroße zsmhängende Teilgraphen vom Graph
[stark] zsmhängend	Alle Eckpaare verbunden
[schwach zsmhängend]	Ungerichteter Graph dazu ist verbunden

[] = nur gerichteter Graph

Tiefen- und Breitensuche:



Tiefensuche (auf einer Seite einfügen & löschen)	Breitensuche (auf einer Seite einfügen und anderer löschen)
[] → [a] → [b, a] → [c, b, a] → [f, c, b, a] → [e, f, c, b, a] → [d, e, f, c, b, a] → [g, d, e, f, c, b, a] → [h, g, d, e, f, c, b, a] → [i, h, g, d, e, f, c, b, a] → [h, g, d, e, f, c, b, a] → [g, d, e, f, c, b, a] → [d, e, f, c, b, a] → [e, f, c, b, a] → [f, c, b, a] → [c, b, a] → [b, a] → [a] → []	[] → [a] → [b, a] → [d,b,a] → [d,b] → [c,d,b] → [e,c,d,b] → [e,c,d] → [g,e,c,d] → [g,e,c] → [f,g,e,c] → [f,g,e] → [h,f,g,e] → [h,f,g] → [h,f] → [i,h,f] → [i,h] → [i]→[]
Prüfung auf Zsmhang, Zsmhangskomponenten, Zyklenfreiheit	Finde kürzesten Pfade von einem Knoten zum anderen + Tiefensuche

Topologisches Sortieren:

(in welche Reihenfolge kann ich einen Graphen abarbeiten) Benötigt einen DAG (Directed Acyclic Graph)
- Zyklenfreier gerichteter Graph

1.Schritt: Führe Tiefensuche
2.Schritt: Vermerke die Reihenfolge, in der Knoten vom Stack gelöscht werden
3.Schritt: umgekehrte Reigenfolge ist eine Lösung

Alternativ:
1.Schritt: Ermittle alle Einstiegspunkte
2.Schritt: Entferne die Einstiegspunkte + Seine Kanten
3.Schritt: Vermerke die Einstiegspunkte, die gelöscht wurden
4.Schritt: Kehre zu Schritt 1 zurück, wenn es noch Einstiegspunkte gibt

Effizenzanalyse Beispiel:

```
ALGORITHM: UniqueElements(A[0..n-1])
for i ← 0 to n - 2 do
    for j ← i + 1 to n - 1 do
        if A[i] = A[j] return false
return true
```

I. Problemgröße:	Array n
II. Basisopp.:	Vergleich im if
III. Best case: Worst case:	A[0] = A[1] Alle Elemente einzigartig oder A[n-2] = A[n-1]
IV. Laufzeitfunktion:	$T_{best}(n) = 1$ $T_{worst}(n) =$ $\sum_{i=0}^{n-2}\left(\sum_{j=i+1}^{n-1} 1\right) = \sum_{i=0}^{n-2}(n-1-(i+1)+1)$ $= \sum_{i=0}^{n-2}(n-i-1) = (n-1)+(n-2)+\cdots+1 = \sum_{i=1}^{n-1} i$ $= \frac{(n-1)*(n-1+1)}{2}$ $= \frac{n(n-1)}{2}$ $= \frac{1}{2}n^2 - \frac{1}{2}n$
V. Komplexitätsklasse:	$T(n) \in \Omega(1), T(n) \in O(n^2)$

Summenformeln:

$$\sum_{i=l}^{u} 1 = u - l + 1$$

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

$$\sum_{i=1}^{n} i^2 = \frac{n*(n+1)*(2n+1)}{6} \approx n^3$$

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

$$\sum_{i=0}^{n} a + b = \sum_{i=0}^{n} a + \sum_{i=0}^{n} b$$

$$\sum_{i=0}^{n} ca = c * \sum_{i=0}^{n} a$$

$$\sum_{i=0}^{n} a = \sum_{i=0}^{m} a + \sum_{i=m+1}^{n} a$$

Master Theorem:

$T(n) = a*T(n/b) + f(n)$ | mit $f(n) \in \theta(n^d)$

Falls $a < b^d$, dann $T(n) \in \theta(n^d)$
Falls $a = b^d$, dann $T(n) \in \theta(n^d \log n)$ Falls $a > b^d$,
dann $T(n) \in \theta(n^{\log_b a})$

L'Hostpital
Vorgabe: $\lim_{n\to\infty} f(n) = \infty$ und $\lim_{n\to\infty} g(n) = \infty$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Stirlingformel (Abschätzung)
$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$

Wachstumsfunktionen:

O(1)	konstant
O(log n)	logarithmisch
O(n)	linear
O(n * log n)	überlogerithmisch
$O(n^2)$	quadratisch
$O(n^3)$	kubisch
$O(k^n)$	exponentiell
O(n!)	faktoriell

ALGORITHM: fak(n)
if n= 0 return 1
else return fak**(n-1) * n**

I. Problemgröße:	Wert von n
II. Basisopp.:	Multiplikation
III. Best case: Worst case:	gleich
IV. Laufzeitfunktion:	T(n)= 0 falls n = 0 T(n)= 1 + T(n-1) sonst Rückwärtiges Einsetzen: 1.Schritt T(n) = 1+T(n-1) | solange n > 0 2.Schritt T(n) = 1+(1+T(n-2)) | da T(n-1) = 1 + T((n-1)-1) = 2+T(n-2) ... K.Schritt T(n) = K + T(n-K) = ... Abbruch bei k = n, da T(n-k) dann T(0) T(n) = n + T(n-n) = n + T(0) = n
V. Komplexitätsklasse:	$T(n) \in \theta(n)$

Dynamische Programmierung

- Problem: Große Probleme, werden in viele kleine überlappende Probleme gesplittet -
Ziel: Diese kleinen Probleme jeweils nur 1 Mal lösen

Coin Row:

index	0	1	2	3	4	5	6
coins	-	5	1	2	9	6	2
F()	**0**	**5**	**5**	**7**	**14**	**14**	**16**

F(n) = max{F(links), coin(n) + F(2 links)} ausfüllen
Dann:
1. Linkeste höchste F(n) - coin(n)
2. Beim Ergebnis das linkeste F(n) nehmen und wdh.

Zeit & Speicher Effizienz: Linear

Münzwechsel: Optimum aus
Münzen: 1, 3, 4

index	0	1	2	3	4	5	6
F()	**0**	**1**	**2**	**1**	**1**	**2**	**2**
Münze	-	**1**	**1**	**3**	**4**	**1**	**3**

index = Wert der Münzen
F() = Wv Münzen braucht man Münze = verwendete Münze für jeweiligen index (wenns 2 mehrere sind nimmt man einfach eine) Endergebnis = Letzte Münze - index (wdh. bis man durch ist)

Zeit Effizienz: n*m; Speicher: Linear zu n

Rucksackproblem:

Gegenstand	Gewicht	Wert
1	2 kg	12 €
2	1 kg	10 €
3	3 kg	20 €
4	2 kg	15 €

Kapazität ->	0	1	2	3	4	5
-	0	0	0	0	0	0
1 (1)	0	0	12	12	12	12
1-2 (2)	0	10	12	22	22	22
1-3 (3)	0	10	12	22	30	32
1-4 (4)	0	10	15	25	30	37

Einpacken: 4, 2

Ergebnis einer Zelle: Zelle darüber **oder** Wert aktuellen Gegenstands + Wert des übrigen Platzes aus Zeile darüber.
Ergebnis Einpacken:
1. Beginn bei höchstem Wert
2. Änderung zur Zeile darüber?
   a. JA -> Eingepackt -> 2. für Kapazität - Gewicht des Gegenstands b.
   NEIN -> 2. für gleiche Kapazität

Komplexitätsklasse:

O(g(n)): Alle die nicht schneller als g(n) wachsen
Θ(g(n)): Alle die genauso schnell wie g(n) wachsen
Ω(g(n)): Alle die mindestens so schnell wie g(n) wachsen

Horspool

Shift-Tabelle:
Wort: ANANAS

A	B	E	N	R	S	T	V	Y
1	6	6	2	6	6	6	6	6

-> Wert = Abstand vom letzten Buchstaben

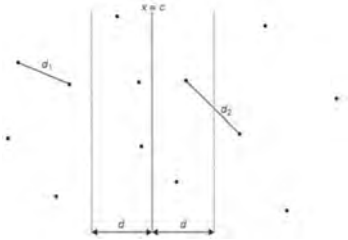
6 Vergleiche (Zeilen), 14 Verschiebungen (Striche)
Bruteforce:
Verschiebungen = Satzlänge - Wortlänge
Vergleiche = Verschiebungen + Matches

Closest Pair:

Suchen des kürzesten Abstands zwischen zwei Punkten

1. Schritt: Teile die Menge deiner Punkte in zwei Mengen (m1, m2).
2. Schritt: Berechne den kürzesten Abstand in den Untermengen (z.B. durch erneutes Teilen)

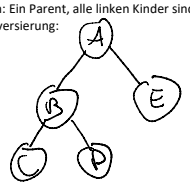
3. Schritt: Berechne das Minimum d aus dem Abstand m1 und m2
   Problem: Der Abstand von einem Punktepaar aus m1 und m2 kann sein als d, sofern der Abstand zur Mitte zwischen m1 und m2 < d ist



4. Schritt: Berechne den Abstand aller Punktpaare von m1 und m2, dabei liegt m1 in der linken Menge und hat von der Mitte einen kleineren x Abstand als d und m2 liegt in der rechten Menge und hat zur Mitte einen kleineren x Abstand als d und zu m1 einen kleineren y Abstand als d. Den kleinsten Abstand nennst du dc
5. Nun Berechne das Minimum von d und dc. Das ist der kürzeste Abstand.

Bäume:

Binärbaum: Ein Parent, alle linken Kinder sind kleiner, alle rechten Größer
Baumtranversierung:



Preorder: A,B,C,D,E
Postorder: C, D, B, E, A
Inorder: C,B,D, A, E

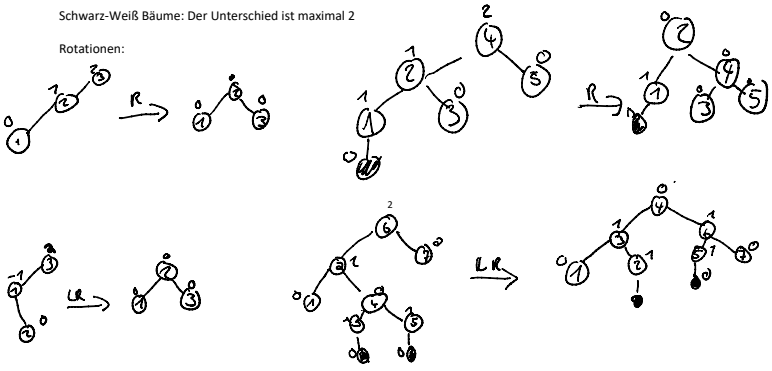
Balacierte Bäume:

Jeder Knoten erhält ein Wert, der durch den schnellsten linke Pfad und dem schnellstem rechtem Pfad ermittelt wird.

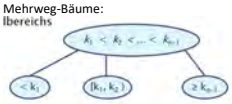
AVL-Bäume: Der Unterschied vom linken und rechtem Baum betragen höchstens 1.

Schwarz-Weiß Bäume: Der Unterschied ist maximal 2

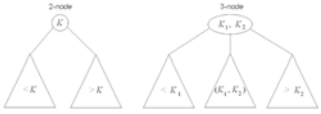
Rotationen:



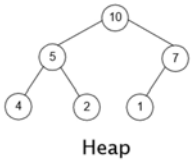
Mehrweg-Bäume:
Ibereichs



2-3 Baum: Ist ein Baum mit 2 oder 3 Knoten:
$\log_3(n+1)-1 <= h <= \log_2(n+1) - 1$ | h = höhe n = knoten



Heapbaum:
Ist ein Binärbaum, in dem alle Ebenen vollständig sind. Nur in der letzten dürfen rechts welche fehlen.
Im Gegensatz zu einem Binärbaum sind alle Kinder kleiner



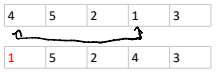
Heap

Linkes Kind von j:Position 2j
Kind von j: Position 2j+1
Elternknoten von j: Position j/2 (abgerundet)
Elternknoten belegen die ersten n/2 (abgerundet) Positionen

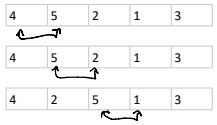
Beim Einfügen in einen Heap, wird das Element an letzter Stelle Eingefügt und nach oben getauscht.
Beim Löschen wird das letzte Element als neue Wurzel genommen und falls nötig nach unten getauscht.

Sortierverfahren:

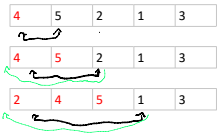
Selctionsort:
Sucht das kleinste Element in einem Array und tauscht das nach vorne.



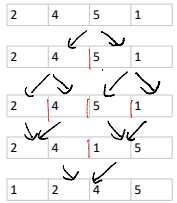
Bubblesort:
Die großen Blasen steigen nach oben. Tauscht das größte Element nach rechts.



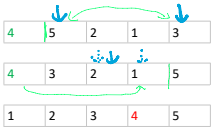
Insertionsort:
Elemente werden durchgegangen und an der richtigen Stelle eingetragen (nicht geswapt)



Mergesort:



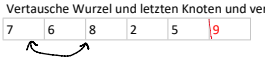
Quicksort:
Bei Quicksort wird immer wieder ein Pivot gewählt und alle Elemente, die größer als der Pivot sind links von diesem einsortiert, alle anderen rechts. Dieses funktioniert mit 2 Zeigern, die die Menge von links und rechts durchgehen und immer stoppen, wenn das Element größer/kleiner als der pivot ist



Heapsort:
Auf Heapbaum.

Konstruiere ein Heap

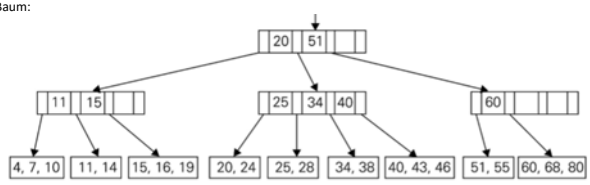
Vertausche Wurzel und letzten Knoten und vermindere Größe um 1:



Stelle Heapeigenschaft wieder her:

Name	Laufzeit			stabil	inPlace
	Best	AVG	Worst		
Selectionsort	Θ(n2)Θ(n2)	Θ(n2)Θ(n2)	Θ(n2)Θ(n2)	✗	✓
Bubblesort	Θ(n)Θ(n)	O(n2)O(n2)	Θ(n2)Θ(n2)	✓	✓
Insertionsort	Θ(n)Θ(n)	Θ(n2)Θ(n2)	Θ(n2)Θ(n2)	✓	✓
Quicksort	Θ(n·log(n))Θ(n·log (n))	Θ(n·log(n))Θ(n·log (n))	Θ(n2)Θ(n2)	✗	✓
Heapsort	O(n·log(n))O(n·log (n))	O(n·log(n))O(n·log (n))	O(n·log(n))O(n·log (n))	✗	✓
Mergesort	Θ(n·log(n))Θ(n·log (n))	Θ(n·log(n))Θ(n·log (n))	Θ(n·log(n))Θ(n·log (n))	✓	✗

B-Baum:



- Ein B-Baum ist vollständig balanciert, d.h. alle Blätter auf der selben Ebene (mit Ordnung m >= 3)
  - Jede Wurzel hat 0 oder 2-m Kinder
  - Alle anderen Knoten haben zwischen (aufgerundet) m/2 und m Kinder zwischen aufgerundet (m/2)-1 und m-1 Schlüssel
  - Der Baum ist vollständig balanciert

Hashing (open and Closed)

Anforderung:
- Leicht zu berechnen
- Schlüssel sollten gleichmäßig über Hashtabelle verteilt sein - wenig Kollisionen

Beispiel: A: 0, B: 1, C: 4, D: 5, E: 5, F: 6
Open Hashing:

Beispiel Closed Hashing:

$\alpha = \frac{Werte}{Platz}$