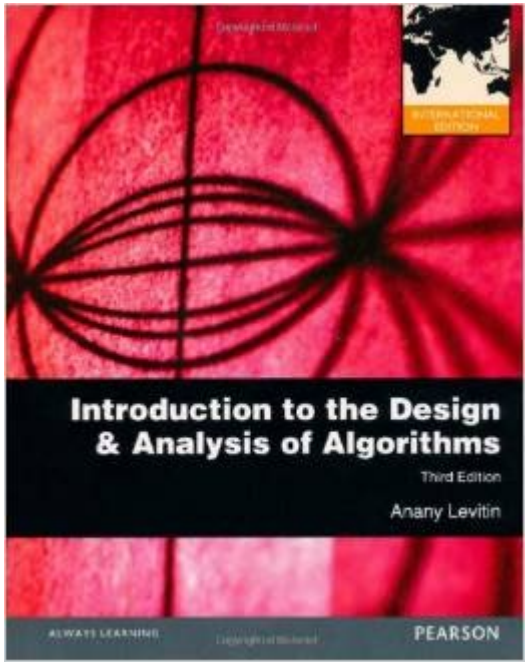
Algorithmen & Datenstrukturen

Decrease-and-Conquer-Algorithmen

Literaturangaben

Diese Lerneinheit basiert größtenteils auf dem Buch „The Design and Analysis of Algorithms“ von Anany Levitin.



In dieser Einheit behandelte Kapitel:
4 Decrease-and-Conquer
4.1 Insertion Sort
4.2 Topological Sorting
4.3 Algorithms for Generating Combinatorial Objects (Auszug)
4.4 Decrease-by-a-Constant-Factor Algorithms (Auszüge)
4.5 Variable-Size-Decrease Algorithms (Auszüge)

Decrease-and-Conquer Strategie

- Reduziere das Problem auf ein gleichartiges Problem **geringerer Größe**
- Löse das kleinere Problem (**top-down** oder **bottom-up**)
- **Erweitere die Lösung** des kleineren Problems zur Lösung des größeren Problems

Alternative Namen:

- **Induktiver** Ansatz
- **Inkrementeller** Ansatz

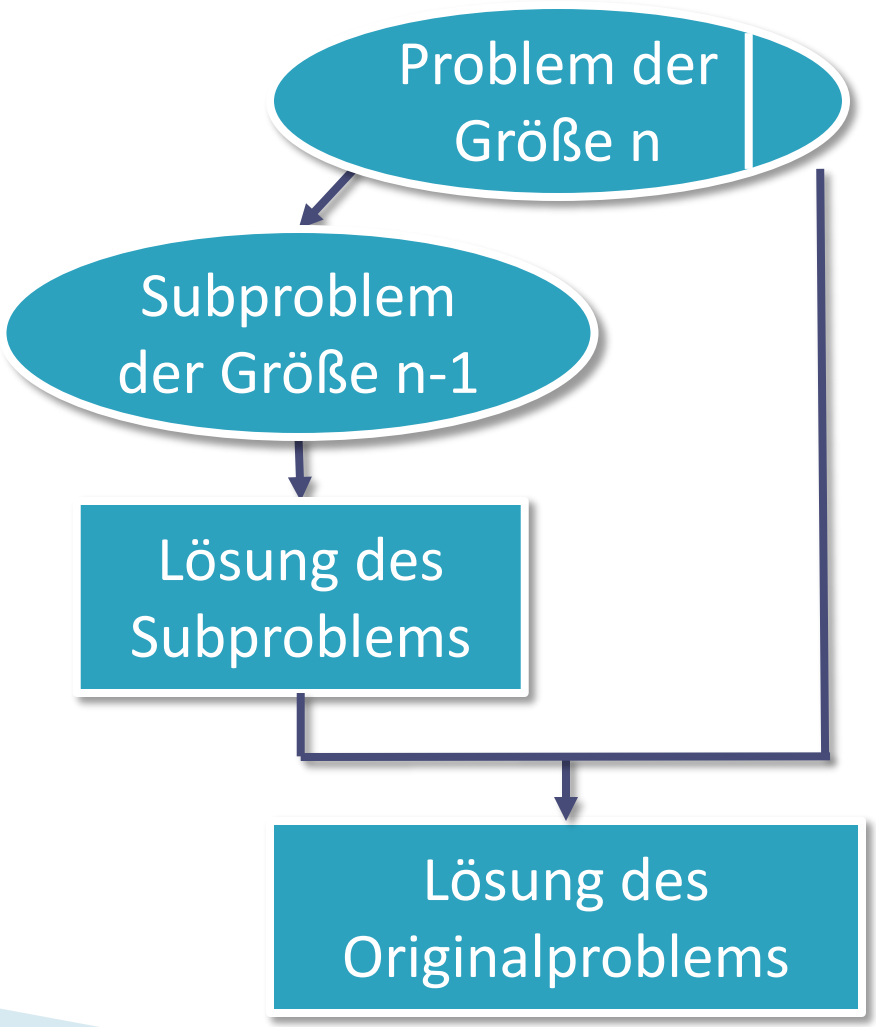
Drei Typen von Decrease-and-Conquer

- **Reduktion um Konstante** (meist 1)
  - Insertion Sort
  - Topologische Sortierung
  - Generierung von Permutationen, Teilmengen

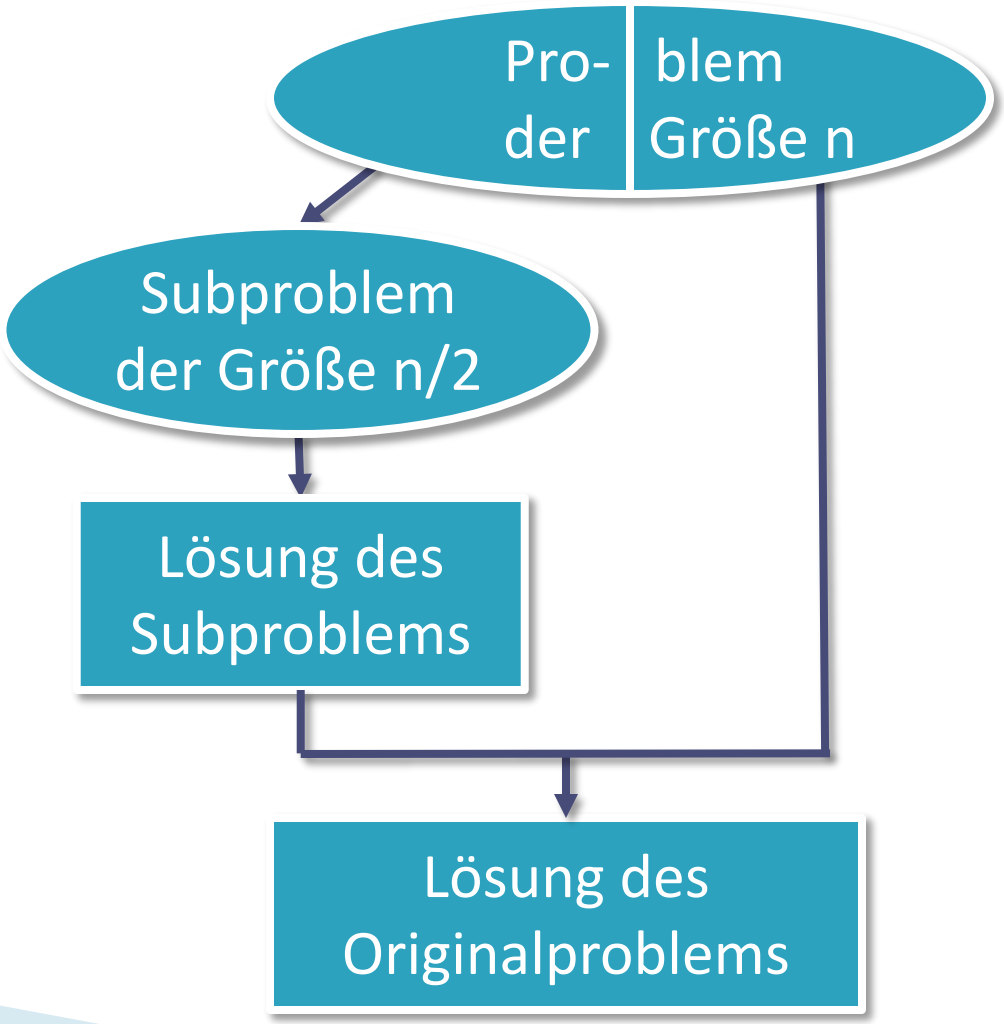
- **Reduktion um konstanten Faktor** (meist 2)
  - Binäre Suche
  - Potenzieren durch Quadrieren
  - Multiplikation à la Russe

- **Reduktion um variable Größe**
  - Euklidischer Algorithmus
  - Selektionsproblem
  - Algorithmen auf binären Suchbäumen
  - Interpolationssuche

Reduktion um Konstante



Reduktion um konstanten Faktor



Decrease-and-Conquer

- Reduktion um Konstante

Insertion-Sort

- **Grundidee**
  Um das Array *A[0..n-1]* zu sortieren, sortiere zunächst das Array *A[0..n-2]* rekursiv und füge dann *A[n-1]* an der richtigen Stelle von *A[0..n-2]* ein
- **Implementierung**
  Üblicherweise **bottom-up** (**nicht-rekursiv**)
- **Beispiel:**

  Sortiere 7, 2, 8, 4, 5

  7 | 2, 8, 4, 5
  2, 7 | 8, 4, 5
  2, 7, 8 | 4, 5
  2, 4, 7, 8 | 5
  2, 4, 5, 7, 8

Pseudocode Insertion-Sort

ALGORITHM *InsertionSort*$(A[0..n-1])$

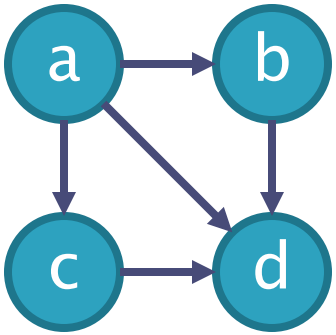
```
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

Analyse Insertion-Sort

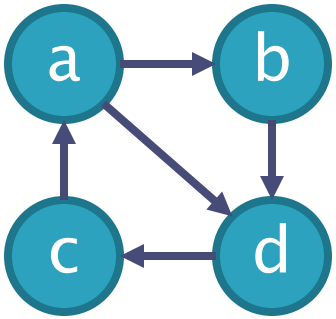
- **Zeiteffizienz**:
  - $T_{worst}(n) = n(n-1)/2 \rightarrow T(n) \in O(n^2)$
  - $T_{best}(n) = n-1 \rightarrow T(n) \in \Omega(n)$
  - $T_{avg}(n) \approx n^2/4$
- **Speicherplatzeffizienz**: in-place
- Insgesamt **bestes elementares Sortierverfahren**
- Mögliche **Optimierung**: Binäres Insertion Sort

DAGs und topologisches Sortieren

- DAG: Directed Acyclic Graph
  - Gerichteter Graph ohne (gerichtete) Zyklen



DAG

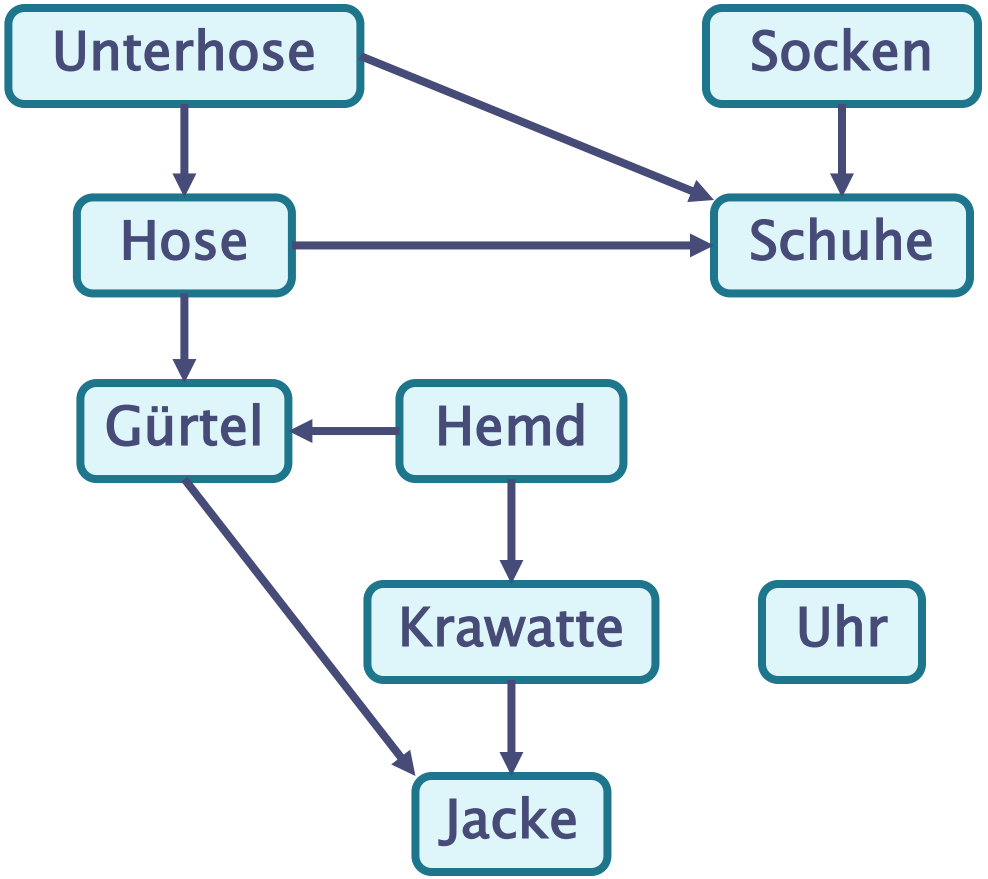


kein DAG

  - Anwendungen: Modellierung von Problemen mit Abhängigkeiten/Vorbedingungen (z. B. Konstruktionsprojekte, Versionskontrollsysteme)
- Topologisches Sortieren:
  - Lineare Anordnung der Knoten, so dass die Startknoten aller Kanten vor ihren Endknoten liegen
  - Graph ist DAG ↔ topologische Sortierung möglich

Topologisches Sortieren: Beispiel

Bestimmen Sie eine korrekte Ankleidefolge



Topologische Sortierung mit Tiefensuche

1) Führe auf dem Graphen eine **Tiefensuche** aus
2) Vermerke die Reihenfolge, in der die Knoten vom Stack **gelöscht** werden
3) Die **umgekehrte Reihenfolge** ist eine Lösung des Problems

- **Effizienz**
  - Tiefensuche $\Theta(|V|+|E|)$ bzw. $\Theta(|V|^2)$
  - Ausgabe in umgekehrter Reihenfolge $\Theta(|V|)$

Topologische Sortierung durch Entfernung von Einstiegspunkten

- **Einstiegspunkt**:
  - Knoten, zu dem keine Kante führt
  - Ein DAG besitzt **mindestens einen** Einstiegspunkt
- **Algorithmus**
  1) Ermittle alle Einstiegspunkte
  2) Entferne alle Einstiegspunkte und die von ihnen ausgehenden Kanten
  3) Wiederhole Schritt 1 und 2 bis es keine Knoten mehr gibt
  4) Die Reihenfolge, in der die Einstiegspunkte gefunden werden, ist eine Lösung des Problems
- **Effizienz**: Wie beim Algorithmus mit Tiefensuche

Permutationen erzeugen

- **Permutationen mit minimalen Veränderungen**
  - Wenn $n = 1$ gebe 1 zurück, sonst
  - generiere rekursiv die Liste aller Permutationen von $12\ldots n-1$ und
  - füge n an jede Stelle dieser Permutationen ein.
  - Arbeite dabei zunächst von rechts nach links und ändere bei jeder neuen Permutation die Arbeitsrichtung.
- **Beispiel: $n=3$**

Start:	1		
2 in 1 von rechts nach links einfügen:	12	21	
3 in 12 von rechts nach links einfügen:	123	132	312
3 in 21 von links nach rechts einfügen:	321	231	213

Decrease-and-Conquer

- Reduktion um konstanten Faktor

Beispiel: Binäre Suche

Algorithm *BinarySearch*(*A[0..n-1], K*)

// Implements nonrecursive binary search

// Input: An array A[0..n-1] sorted in ascending order and search key K

// Output: An index of the array's element that is equal to K or -1 if there
// is no such element

$l \leftarrow 0, r \leftarrow n - 1$

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

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

  if $K = A[m]$ **return** m

  else if $K < A[m]$ $r \leftarrow m - 1$

  else $l \leftarrow m + 1$

return -1

Binäre Suche: Analyse

- **Zeiteffizienz**:

  $T_{worst}(n) = T_{worst}(\lfloor n/2 \rfloor) + 1,\ T_{worst}(1) = 1$

  Lösung: $T_{worst}(n) = \lceil log_2(n+1) \rceil$
- $T(n) \in O(\log n)$ (**sehr** schneller Algorithmus)
- **Optimal** für sortierte Arrays
- **Einschränkung**: Arbeitet nur mit sortierten Arrays (nicht mit verketteten Listen)

Multiplikation à la Russe

- **Problem**
  Berechne das Produkt zweier positiver Ganzzahlen m und n
- **Lösungsansatz**
  m halbieren und n verdoppeln bis $m = 1$
- **Es gilt**

$$m * n = \frac{m}{2} * 2n \qquad falls\ m\ gerade$$

$$m * n = \frac{(m-1)}{2} * 2n + n \qquad falls\ m\ ungerade\ und\ m > 1$$

$$m * n = n \qquad falls\ m\ = 1$$

Multiplikation à la Russe: Beispiel

- Berechne 22 * 27

m	n	+
22	27	
11	54	54
5	108	108
2	216	
1	432	432
		594

- Reduktion auf die Addition aller n-Werte, bei denen m ungerade ist

Decrease-and-Conquer

- Reduktion um variable Größe

Beispiel: Euklidscher Algorithmus

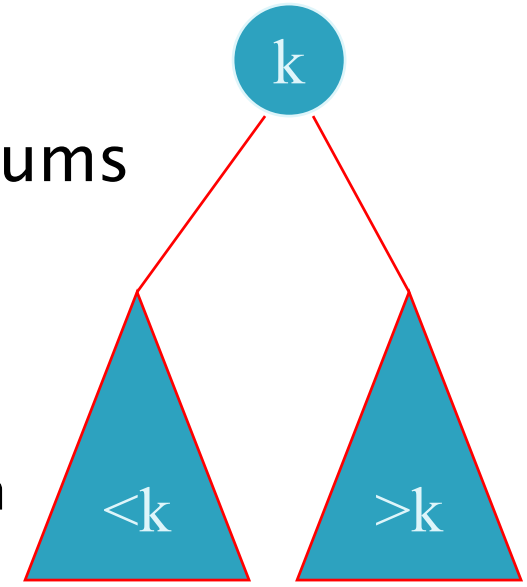
- **Problem**: Finde den **größten gemeinsamen Teiler (ggT)** zweier positiver Ganzzahlen m und n
- **Idee**:
  - Sei $m > n$ (sonst vertausche m und n)
  - $ggT(m,n) = ggT(n, m \bmod n)$
- **Beispiel**:
  - $ggT(91,52) = ggT(52,39) = ggT(39,13) = ggT(13,0) = 13$
- **Effizienz**:
  - Nach zwei Schritten hat sich n **mindestens halbiert**
  - $T_{euklid}(n) \in O(\log n)$

Selection Problem (Auswahlproblem)

- **Ziel**
  Finde das k-t kleinste (z. B. viert kleinste) Element in einer Liste von n Elementen
- **Sonderfälle**
  k = 1: Minimum
  k = n: Maximum
  k = n/2: Median (Zentralwert)
- **Lösungsansätze**
  - Auswahl nach Sortierung
  - Wiederholte Minimum- oder Maximumsuche
  - Wiederholte Partitionierung, z. B. Quickselect (Decrease-and-Conquer-Algorithmus mit Reduktion um variable Größe)

Algorithmen auf binären Suchbäumen

- Eine Reihe von Algorithmen auf binären Suchbäumen erfordern nur die Verarbeitung **eines** Unterbaums
- **Beispiele**
  - Schlüsselsuche
  - Schlüssel einfügen
  - größten/kleinsten Schlüssel bestimmen
- Die entsprechenden Algorithmen sind typischerweise Decrease-and-Conquer-Algorithmen mit Reduktion um eine variable Größe



Suche in binären Suchbäumen

ALGORITHM *BinaryTreeSearch(x, v)*

// Sucht im binären Suchbaum mit der Wurzel x
// nach einem Knoten dessen Schlüsselwert
// gleich v ist

if x = NIL **return** -1

else if v = k(x) return x

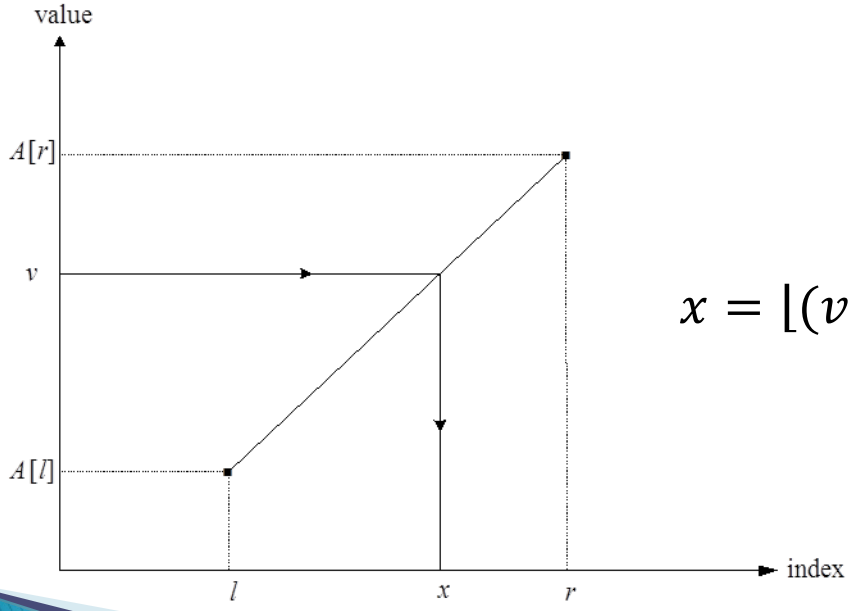
else if v < k(x) return *BinaryTreeSearch(left(x), v)*

else return *BinaryTreeSearch(right(x), v)*

- **Effizienz**
  - Worst case: $T(n) \in O(n)$
  - Average case: $T(n) \approx 2 \ln n \approx 1.39\, log_2 n$

Interpolationssuche

- **Ähnlich binärer Suche**: durchsucht sortierte Arrays
- **Schätzt/interpoliert** die Position des Suchschlüssels innerhalb der Grenzen A[l] und A[r]
- Hilfreich bei in etwa **linear wachsenden Werten**



$$x = \lfloor (v - A[l])/(A[r] - A[l]) \cdot (r - l) \rfloor + l$$

Analyse der Interpolationssuche

- **Effizienz**
  - Durchschnittlicher Fall: $T(n) < \log_2(\log_2(n + 1))$
  - Schlechtester Fall: $T(n) = n$
- **Binärer Suche vorzuziehen**
  - bei **SEHR** großen Arrays
  - bei **teuren** Vergleichsoperationen