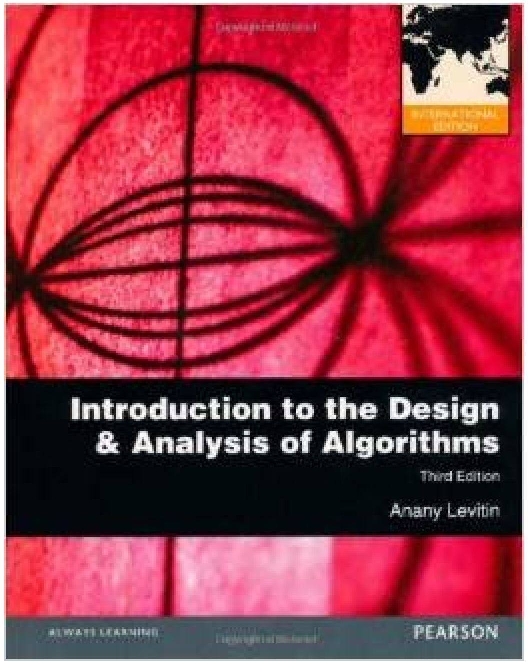
Algorithmen & Datenstrukturen

Divide-and-Conquer-Algorithmen

Uwe Neuhaus

Literaturangaben

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

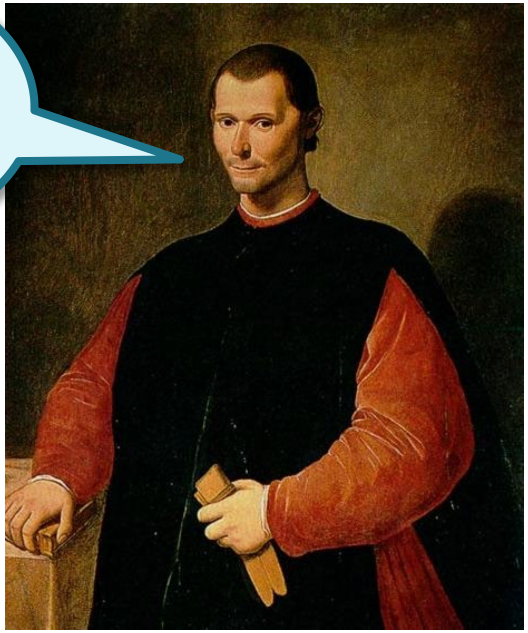


In dieser Einheit behandelte Kapitel:
5 Divide-and-Conquer
5.1 Mergesort
5.2 Quicksort
5.3 Binary Tree Traversals and Related Problems (Auszüge)

Divide-and-Conquer Strategie

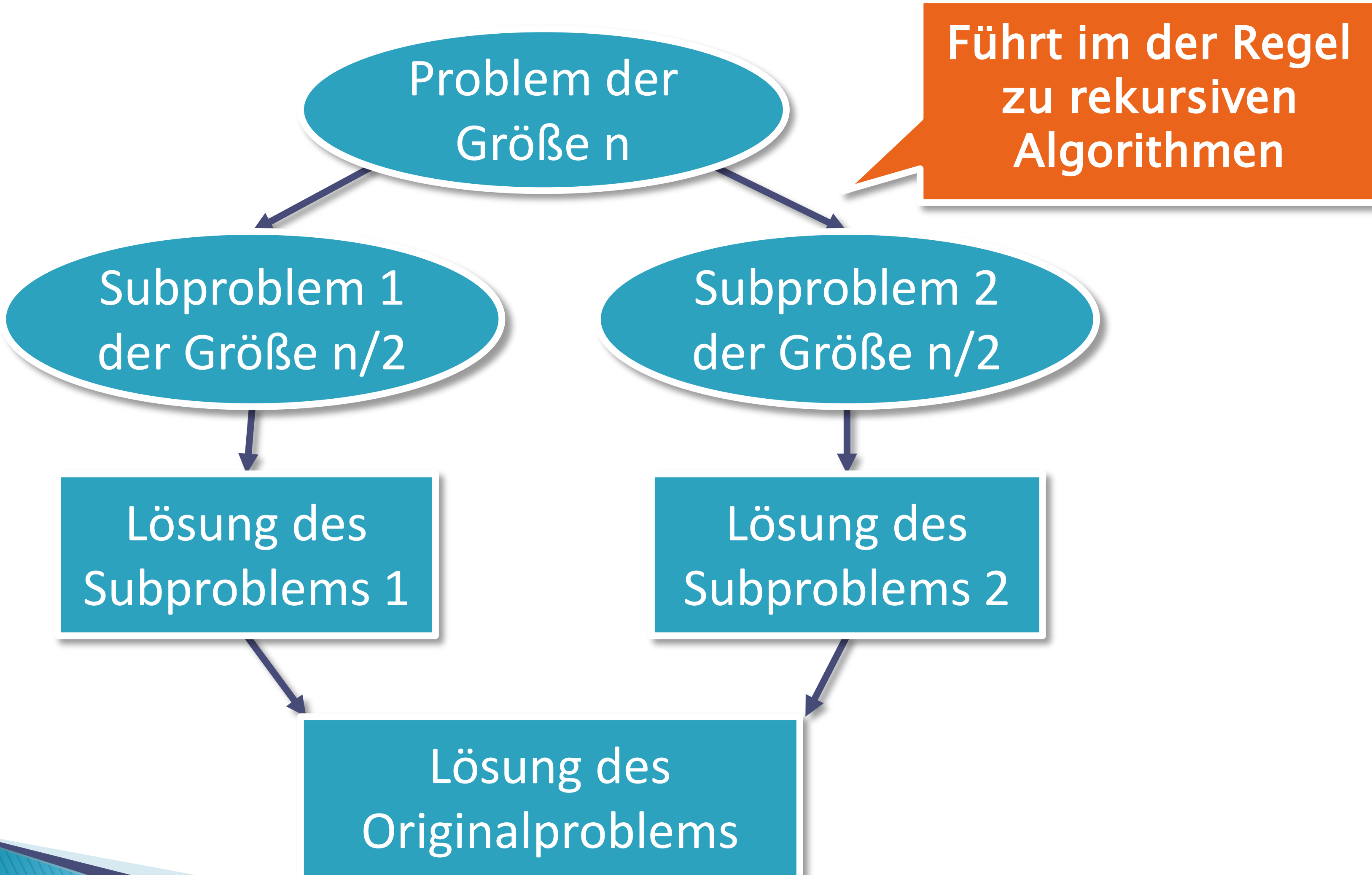
- **Altbewerte Strategie der (Macht-) Politik**
- **Deutscher Name**
  - Teile und herrsche
- **Bekannteste Designstrategie für Algorithmen**
  1. Teile das gegebene Problem in zwei oder mehr kleinere, gleichartige Problem-Exemplare
  2. Löse die kleineren Problem-Exemplare rekursiv
  3. Kombiniere die Lösungen der kleineren Problem-Exemplare zur Lösung des größeren Exemplars

divide et impera



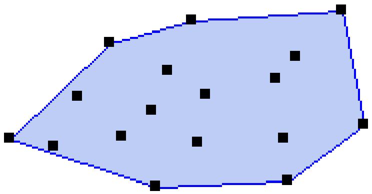
Niccolò Machiavelli

Divide-and-Conquer Strategie



Beispiele für Divide-and-Conquer-Algorithmen

- Sortieren: Mergesort und Quicksort
- Traversierung von Binärbäumen
- Multiplikation großer Ganzzahlen
- Matrizenmultiplikation: Strassen-Algorithmus
- Closest-Pair-Algorithmus
- Konvexe-Hüllen-Algorithmus



Mergesort: Grundlegende Idee

- Teile Array A[0..n−1] in zwei Hälften und kopiere Sie in die Arrays B und C
- Sortiere die Arrays B und C rekursiv
- Füge die Arrays B und C sortiert zusammen:
  - Wiederhole bis ein Array leer ist:
    - Vergleiche die vordersten, noch nicht verarbeiteten Elemente der Arrays
    - Kopiere das kleinere der beiden Elemente an die nächste freie Stelle von A, dieses Element ist nun verarbeitet
  - Sobald alle Elemente eines Arrays verarbeitet wurden, kopiere die restlichen Elemente des zweiten Arrays in A

Mergesort: Pseudocode (I)

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

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    $Mergesort(B[0..\lfloor n/2 \rfloor - 1])$
    $Mergesort(C[0..\lceil n/2 \rceil - 1])$
    $Merge(B, C, A)$

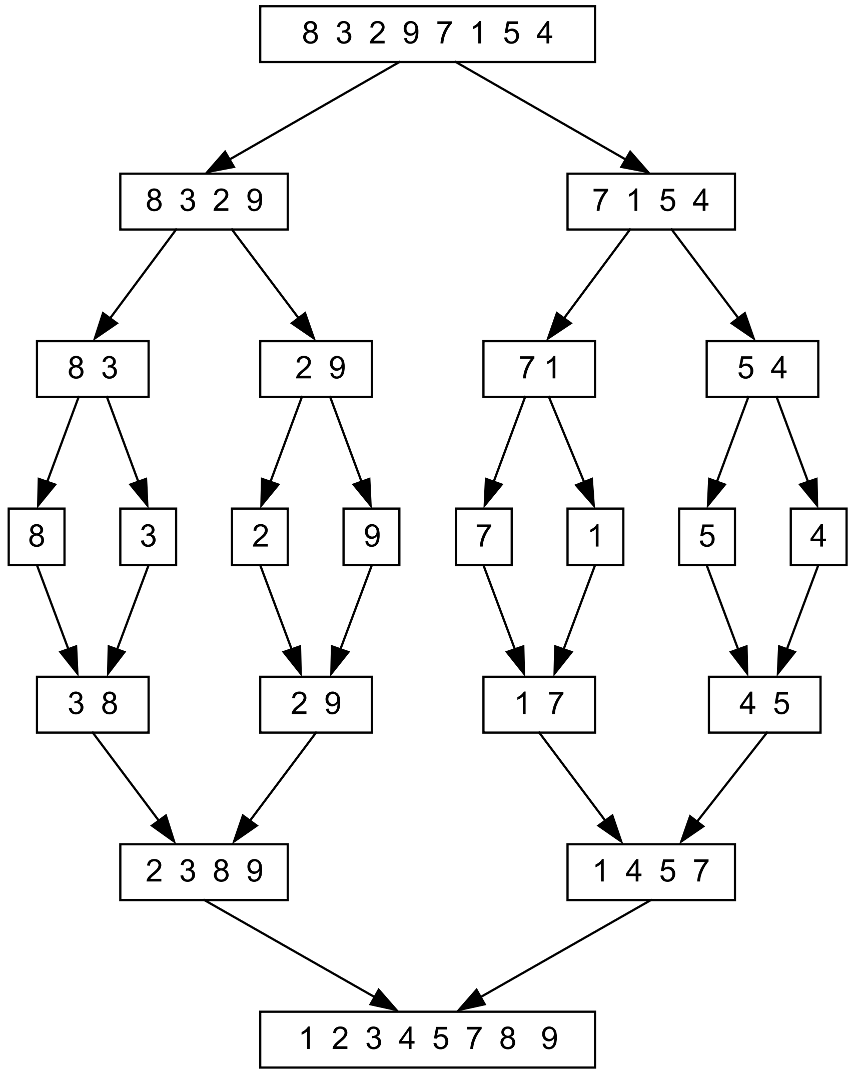
Mergesort: Pseudocode (II)

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C

$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
    else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$
    $k \leftarrow k + 1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Mergesort: Beispiel



Mergesort: Analyse

- **Laufzeit**
  - in allen Fällen ≈ n log n, somit $T(n) \in \Theta(n \log n)$
  - Anzahl Vergleiche im schlechtesten Fall nahe am theoretischen Minimum vergleichsbasierten Sortierens:

$$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$$

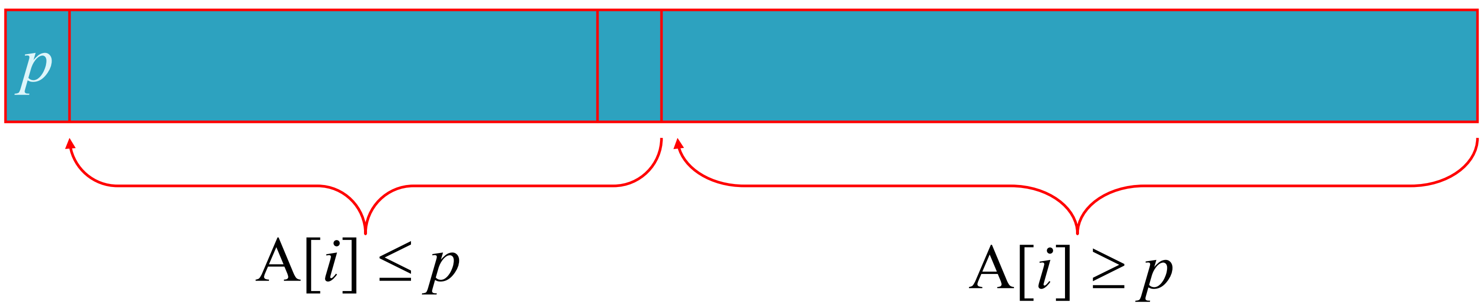
- **Speicherplatz**
  - theoretisch $\Theta(1)$ erreichbar, Umsetzung aber komplex
  - in der Praxis $\Theta(n)$ und somit nicht in-place

Mergesort: Praktischer Einsatz

- Implementierung ohne Rekursion (**bottom-up**) möglich
- **Stabiles** Sortierverfahren
- Arrays werden **sequentiell** verarbeitet
  - kein wahlfreier Zugriff notwendig
  - gut geeignet zur **Sortierung verketteter Listen**
  - gut geeignet zur **Sortierung von Dateien auf externen Speichermedien** (**externes Sortierverfahren**)

Quicksort: Grundidee

- Wähle einen **Pivot p** (**Partitionierungselement**) – hier z. B. einfach das erste Element
- Verschiebe alle restlichen Elemente so, dass die Elemente **kleiner-gleich** dem Pivot links von den Elementen **größer-gleich** dem Pivot stehen



- Tausche den Pivot mit dem letzten Element des ersten Sub-Arrays – Der Pivot steht nun auf seiner **endgültigen Position**
- Sortiere die beiden Unter-Arrays **rekursiv**

Quicksort

Algorithm *Quicksort(A[l...r])*

// Sort a subarray by quicksort

// Input: Subarray of A[1..n-1], defined by its left and

// right indices l and r

// Output: Subarray A[l..r] sorted in non-decreasing order

if *l < r*

$s \leftarrow$ *Partition*(A[l..r]) // s is split position

Quicksort(A[l..s-1])

Quicksort(A[s+1..r])

Partitioning nach Hoare

Algorithm $Partition(A[l..r])$
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
//   indices l and r $(l < r)$
//Output: A partition of $A[l..r]$, with the split position returned as
//   this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
    repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
    repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
    swap($A[i], A[j]$)
until $i \geq j$
swap($A[i], A[j]$)   //undo last swap when $i \geq j$
swap($A[l], A[j]$)
return j

Quicksort: Beispiel

5 3 1 9 8 2 4 7

Quicksort: Analyse

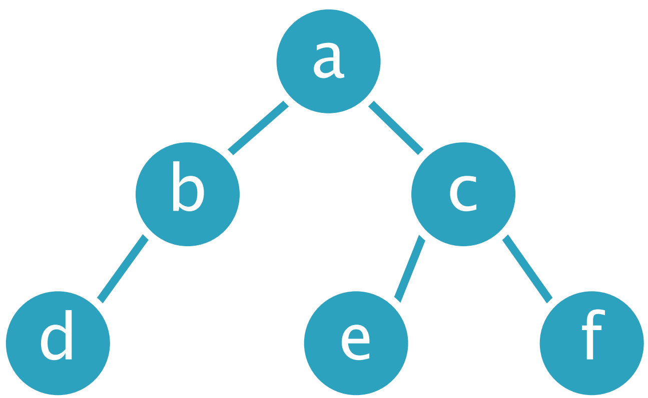
- **Laufzeit**
  - **Bester Fall**: Teilung in der Mitte: $T(n) \in \Omega(n \log n)$
  - **Schlechtester Fall**: bereits sortiertes Array, dann wird nur der Pivot abgespalten: $T(n) \in O(n^2)$
  - **Mittler Fall**: Zufällig angeordnete Arrays: $T(n) \approx n \log n$
- **Speicherplatz**
  - **Stack für Rekursion** benötigt (Übergabe der Array-Grenzen)
  - Minimal **$O(\log n)$** Speicherplätze notwendig
  - Quicksort daher **nicht in-place**

Quicksort: Praktischer Einsatz

- Quicksort **nicht stabil**
- **Verbesserungen**:
  - Bessere Pivot-Auswahl:
    - z. B. mittleres Element von drei zufällig gewählten
    - viele weitere Strategien möglich
  - Wechsel auf Insertion-Sort bei kleinen Sub-Arrays
  - Dreiteilige Partition ($<$, $>$, $=$ Pivot)
  - Gesamtverbesserungspotential: ca. 20 – 25%
- Bestes Verfahren für **interne Sortierung großer Datensätze** ($n \geq 10000$)

Binärbaum-Algorithmen (I)

- Binärbäume bieten sich aufgrund ihrer Struktur unmittelbar für Divide-and-Conquer-Algorithmen an!
- **Beispiel 1**: Baumtraversierung (inorder, preorder, postorder)



Algorithm *Inorder*(T)
if $T \neq \emptyset$
  Inorder(T_{left})
  print(root of T)
  Inorder(T_{right})

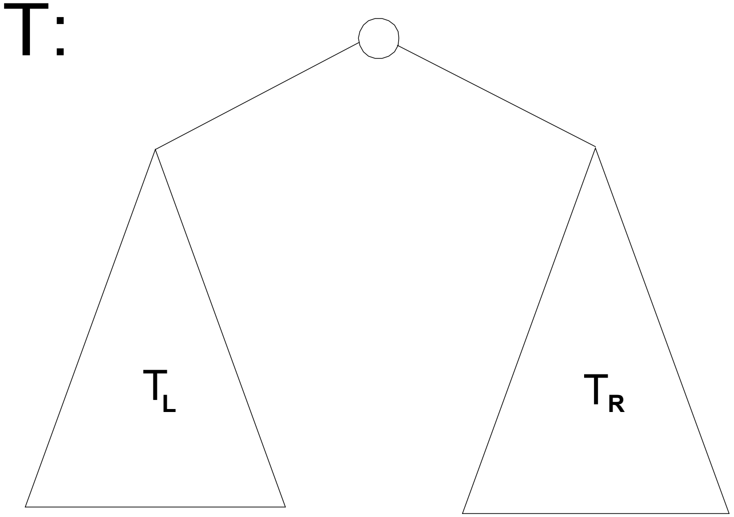
Pre-order

Post-order

- Laufzeiteffizienz: $\Theta(n)$

Binärbaum-Algorithmen (II)

- **Beispiel 2:** Berechne die Höhe h des Binärbaums T:



$h(T) = 1 + \max\{h(T_L), h(T_R)\}$ falls $T \neq \varnothing$

$h(\varnothing) = -1$

- Laufzeiteffizienz: $\Theta(n)$

Divide-and-Conquer Algorithmen: Allgemeine Rekursionsgleichung

$T(n) = a \cdot T(n/b) + f(n)$ mit $f(n) \in \Theta(n^d)$, $d \geq 0$

Master-Theorem: 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 $T(n) \in \Theta(n^{\log_b a})$

Hinweis: Das Gleiche gilt für die O-Notation

Beispiele:
$T(n) = 4T(n/2) + n \Rightarrow T(n) \in \Theta(n^2)$
$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in \Theta(n^2 \log n)$
$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in \Theta(n^3)$

Macht es einen Unterschied?

- Betrachte: **Potenzberechnung $f(n) = a^n$**
  - Allgemein: $f(0) = 1$, $f(1) = a$
- **Brute-Force**:
  - $f(n) = \underbrace{a \cdot a \cdot a \cdot \ldots \cdot a}_{n}$
- **Divide-and-Conquer**
  - $f(n) = f(\lfloor n/2 \rfloor) \cdot f(\lceil n/2 \rceil)$ für $n > 1$
- **Reduzierung um Konstante** (1)
  - $f(n) = f(n-1) \cdot a$ für $n > 1$
- **Reduzierung um konstanten Faktor** (2)
  - $f(n) = f(n/2)^2$ für gerade n
  - $f(n) = f((n-1)/2)^2 \cdot a$ für ungerade n

Anzahl der Multiplikationen?