

---

# Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

**Universität zu Lübeck**

**Institut für Informationssysteme**

Tanya Braun (Übungen)

sowie viele Tutoren



# Algorithmen

---

- Verfahren zur Berechnung eines Problems
- Für ein gegebenes Problem gibt es mehrere Algorithmen
- Unsere Aufgabe:
  - **Verstehen** von Algorithmen
  - **Entwickeln** von Algorithmen
- Verstehen: „Idee“ (**Entwurfsmuster**) eines Algorithmus erläutern können

# Entwurfsmuster / Entwurfsverfahren

---

- Schrittweise Berechnung
  - Beispiel: Bestimmung der Summe eines Feldes durch Aufsummierung von Feldelementen
- Ein-Schritt-Berechnung
  - Beispiel: Bestimmung der Summe eines Feldes ohne die Feldelemente selbst zu betrachten (geht nur unter Annahmen)
- Verkleinerungsprinzip
  - Beispiel: Sortierung eines Feldes
    - Unsortierter Teil wird immer kleiner, letztlich leer
    - Umgekehrt: Sortierter Teil wird immer größer, umfasst am Ende alles → Sortierung erreicht

# Notation für Ideen hinter Algorithmen

---

- Skizzen von beispielhaften Verarbeitungssequenzen
- Hinter den meisten Ideen stecken sogenannte **Invarianten**
- Wenn Invarianten (und die Idee dahinter) verstanden sind, kann man einen **Algorithmus notieren**
- Spezielle Notationen für Algorithmen sind **Programme** (nützlich für Computer)!
- Menschen erzielen durch Ansehen von Programmen meist nur mit großem Aufwand ein Verständnis der Idee eines Algorithmus (und der Invarianten dahinter)

# Laufzeitanalyse

---

- Laufzeit als **Funktion der Eingabegröße**
- Verbrauch an Ressourcen: **Zeit, Speicher**, Bandbreite, Prozessoranzahl, ...
- Laufzeit bezogen auf serielle Maschinen mit wahlfreiem Speicherzugriff
  - **von Neumann-Architektur ...**
  - ... und Speicherzugriffszeit als konstant angenommen
- Laufzeit kann von der Art der Eingabe abhängen (**besten Fall, typischer Fall, schlechtesten Fall**)
- Meistens: schlechtesten Fall betrachtet

# Aufwand für Zuweisung, Berechnung, Vergleich?

1: <b>procedure</b> INSERTION-SORT( $A$ )	Zeit	Wie oft?
2: <b>for</b> $j \leftarrow 2$ to $length(A)$ <b>do</b>	$c_1$	$n$
3: $key \leftarrow A[j]$	$c_2$	$n - 1$
4:         ▷ Insert $A[j]$ into $A[1..j - 1]$		
5: $i \leftarrow j - 1$	$c_4$	$n - 1$
6: <b>while</b> $i > 0$ and $A[i] > key$ <b>do</b>	$c_5$	$\sum_{j=2}^n t_j$
7: $A[i + 1] \leftarrow A[i]$	$c_6$	$\sum_{j=2}^n (t_j - 1)$
8: $i \leftarrow i - 1$	$c_7$	$\sum_{j=2}^n (t_j - 1)$
9: $A[i + 1] \leftarrow key$	$c_8$	$n - 1$

$t_j =$  Anzahl der Durchläufe der *while*-Schleife im  $j$ -ten Durchgang.

$c_1 - c_8 =$  unspezifizierte Konstanten.

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$$

# Bester Fall: Feld ist aufsteigend sortiert

---

Ist das Array bereits aufsteigend sortiert, so wird die *while*-Schleife jeweils nur einmal durchlaufen:  $t_j = 1$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Also ist  $T(n)$  eine **lineare Funktion** der Eingabegröße  $n$ .

# Schlechtester Fall: Feld ist absteigend sortiert

Ist das Array bereits absteigend sortiert, so wird die *while*-Schleife maximal oft durchlaufen:  $t_j = j$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$$

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1 \qquad \sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \left( \frac{n(n+1)}{2} - 1 \right) + (c_6 + c_7) \frac{n(n-1)}{2}$$

Also ist  $T(n)$  eine **quadratische Funktion** der Eingabegröße  $n$ .



# Schlimmster vs. typischer Fall

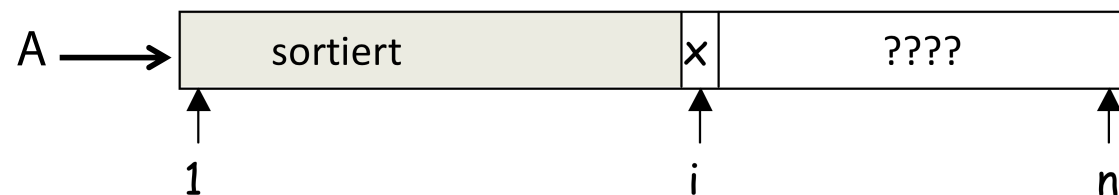
---

Meistens geht man bei der Analyse von Algorithmen vom (*worst case*) aus.

- *worst case* Analyse liefert **obere Schranken**
- In vielen Fällen ist der *worst case* die Regel
- Der aussagekräftigere (gewichtete) Mittelwert der Laufzeit über alle Eingaben einer festen Länge (*average case*) ist oft bis auf eine multiplikative Konstante nicht besser als der *worst case*.
- Belastbare Annahmen über die mittlere Verteilung von Eingaben sind oft nicht verfügbar.

# Eine andere "Idee" für Sortieren

- Gegeben:  $a = [4, 7, 3, 5, 9, 1]$
- Gesucht: In-situ-Sortierverfahren



```
1: procedure SELECTION-SORT( $A$ )
2:   for  $i \leftarrow 1$  to  $length(A)$  do
3:      $min \leftarrow i$ 
4:     for  $j \leftarrow i + 1$  to  $length(A)$  do
5:        $\triangleright$  find the minimum in the unsorted part
6:       if  $A[j] < A[min]$  then
7:          $min \leftarrow j$ 
8:      $x \leftarrow A[i]; A[i] \leftarrow A[min]; A[min] \leftarrow x$ 
9:      $\triangleright$  swap the found minimum into the sorted part
```

# Sortieren durch Auswählen (Selection-Sort)

---

- Gleiches **Entwurfsmuster: Verkleinerungsprinzip**
- Aufwand im **schlechtesten Fall?**
  - $T(n) = c_1 n^2$
- Aufwand im **besten Fall?**
  - $T(n) = c_2 n^2$
- Sortieren durch Auswählen scheint also noch schlechter zu sein als Sortieren durch Einfügen !
- Kann sich jemand eine Situation vorstellen, in der man trotzdem zu Sortieren durch Auswählen greift?
  - Was passiert, wenn die Elemente sehr groß sind?
  - Verschiebungen sind aufwendig

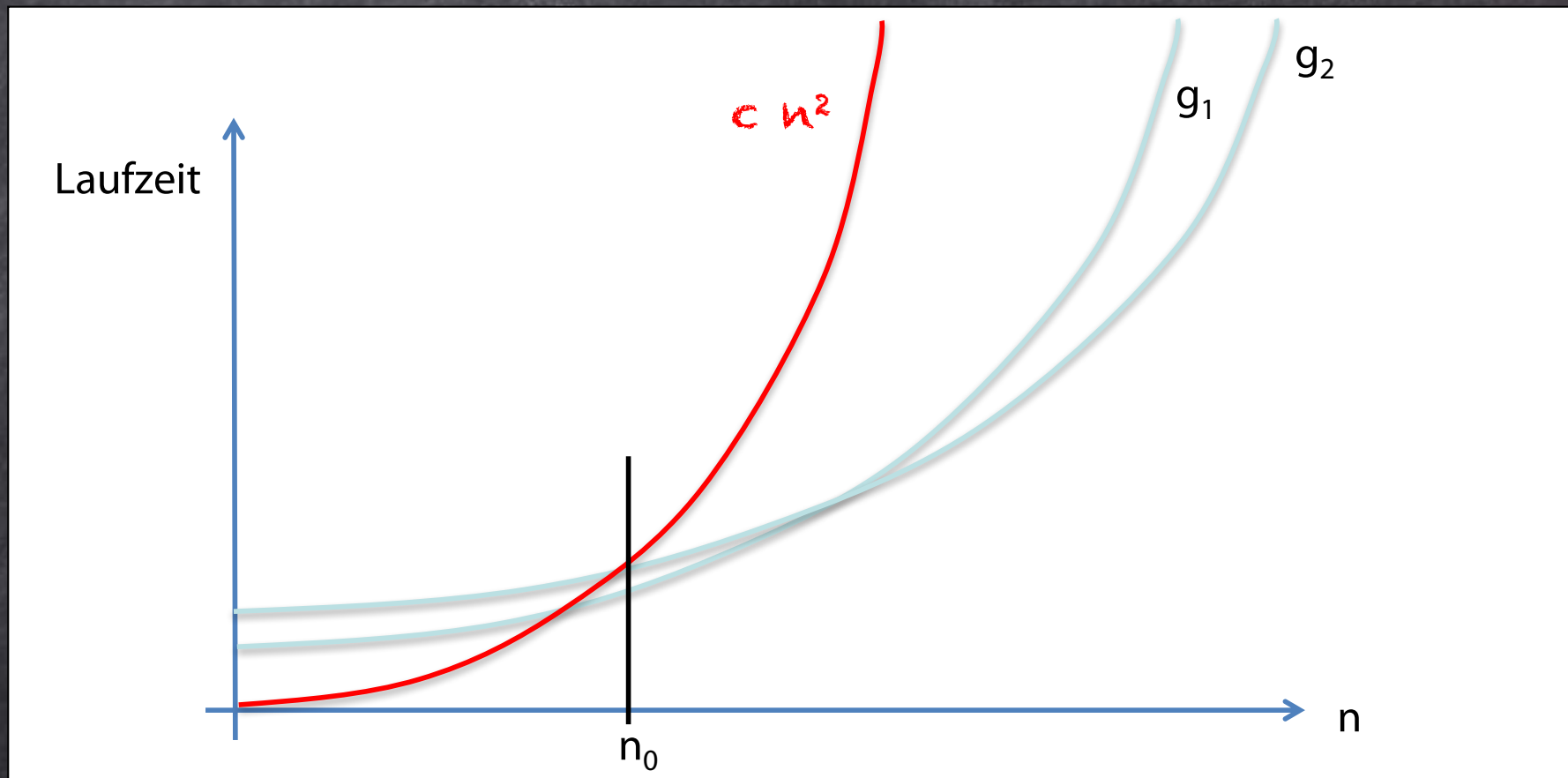
# Vergleich der beiden Algorithmen

---

- Anzahl Vergleiche?
- Anzahl Zuweisungen?
- Berechnungen?
  
- Was passiert, wenn die Eingabe immer weiter wächst?
  - $n \mapsto \infty$
- Asymptotische Komplexität eines Algorithmus
- Die Konstanten  $c_i$  sind nicht dominierend
- Lohnt es sich, die Konstanten zu bestimmen?
- Sind die beiden Algorithmen substantiell verschieden?

# Quadratischer Aufwand

- Algorithmus 1:  $g_1(n) = b_1 + c_1 * n^2$
- Algorithmus 2:  $g_2(n) = b_2 + c_2 * n^2$



# Oberer „Deckel“: O-Notation

---

- Charakterisierung von Algorithmen durch Klasse von Funktionen

$$O(f(n)) = \{ g: \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0 > 0 \wedge \exists c > 0, \text{ so dass } \forall n \geq n_0: g(n) \leq c \cdot f(n) \}$$

- Sei  $f(n) = n^2$
- $g_1 \in O(n^2)$
- $g_2 \in O(n^2)$
  
- $g_1$  und  $g_2$  sind „von der gleichen Art“

Erstmals vom deutschen Zahlentheoretiker Paul Bachmann in der **1894** erschienenen zweiten Auflage seines Buchs *Analytische Zahlentheorie* verwendet. Verwendet auch vom deutschen Zahlentheoretiker Edmund Landauer, daher auch Landau-Notation genannt (**1909**) [Wikipedia 2015]

In der Informatik populär gemacht durch Donald Knuth, In: *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley, **1968**

# O-Notation

---

$$O(f(n)) = \{ g: \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0 > 0 \wedge \exists c > 0, \text{ so dass } \forall n \geq n_0: g(n) \leq c \cdot f(n) \}$$

- Linearer Aufwand:  $O(n)$
- Konstanter Aufwand:  $O(1)$

# Quadratischer Aufwand

- $g(n) = n^2 + n + 42$

- $g \in O(n^2)?$



# Quadratischer Aufwand

- $g(n) = 17n^2 + n + 42$

- $g \in O(n^2)?$

# Anmerkungen

---

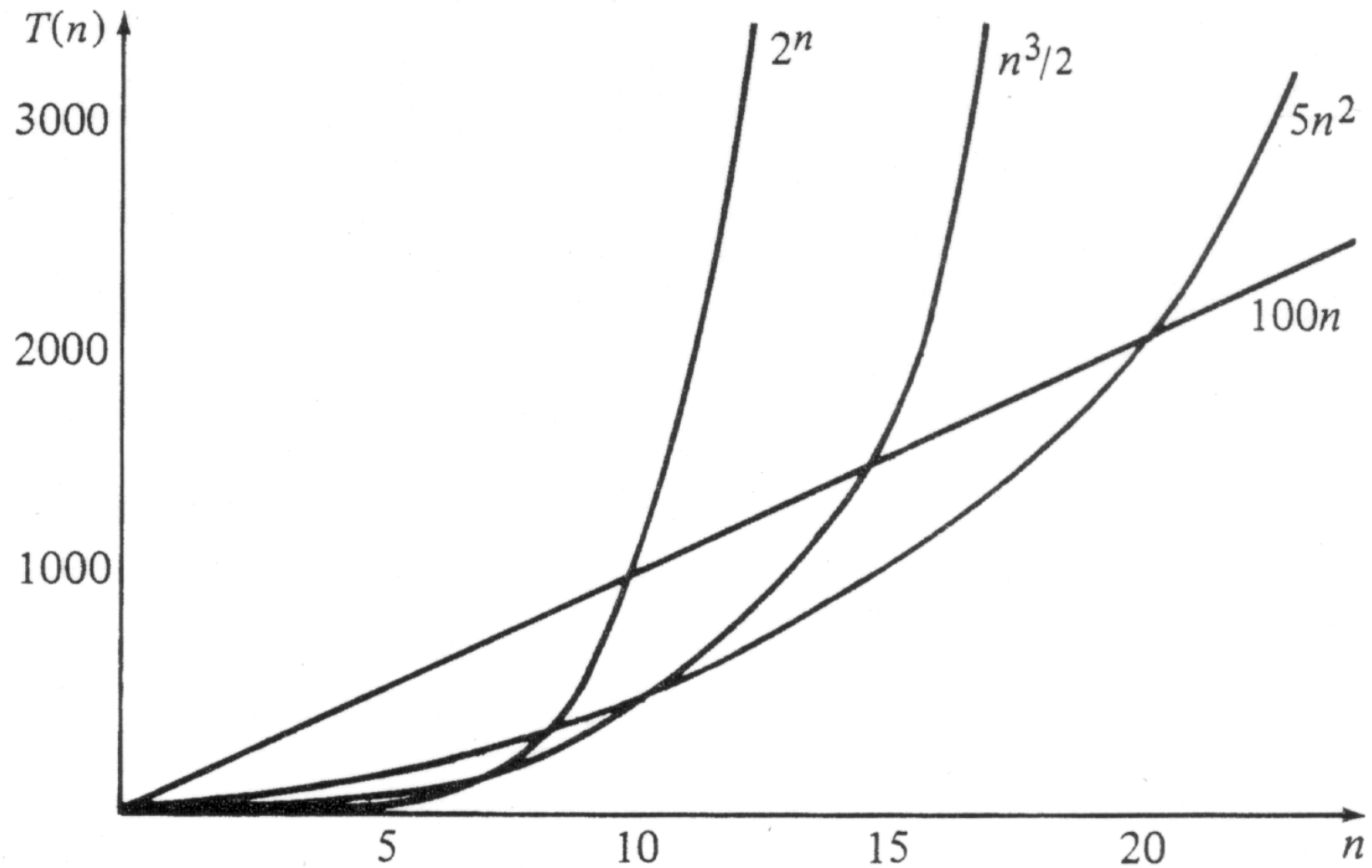
- Die O-Notation gibt eine "obere Schranke" an, das heißt aber nicht, dass der Algorithmus auch jemals soviel Zeit benötigt, es heißt nur, dass garantiert NIE mehr Zeit benötigt wird. Vielleicht tritt der untersuchte ungünstigste Fall in der Praxis nie auf. Deshalb sollte man immer auch noch eine Betrachtung des durchschnittlichen und besten Falls in Erwägung ziehen.
- Die systemabhängige Konstante  $c$  ist im Allgemeinen unbekannt, muss aber nicht unbedingt klein sein. Dies birgt ein erhebliches Risiko. Z.B. würde man natürlich einen Algorithmus, welcher  $n^2$  Nanosekunden benötigt, einem solchen vorziehen, der  $n$  Jahrhunderte läuft, anhand der O-Notation könnte man diese Entscheidung allerdings nicht treffen, da eine lineare Komplexität besser zu sein scheint als eine quadratische. Hier ist also höchste Vorsicht geboten!
- Die Konstante  $n_0$ , ab der für alle  $n > n_0$  gilt:  $g(n) \leq c \cdot f(n)$ , ist ebenfalls unbekannt und muss genau wie  $c$  nicht unbedingt klein sein. So könnte  $n_0$  1 Million betragen, in der Praxis könnten allerdings meist nur weniger Eingabedaten vorkommen. D.h. dass die Laufzeit des Algorithmus' über der mittels O-Notation angegebenen Schranke liegt und somit ein falsches Ergebnis vorliegt.
- Schließlich ist die O-Notation eine Abschätzung der Laufzeit bei einer unendlichen Eingabemenge. Da jedoch keine Eingabe unendlich ist, sollte man bei der Wahl von Algorithmen die realistische Eingabelänge mit einbeziehen.

# Verfeinerung: Lernziele

---

- Entwickeln einer **Idee** zur Lösung eines Problems
- **Notieren** der Idee
  - Zur Kommunikation mit Menschen (Algorithmus)
  - Zur Ausführung auf einem Rechner (Programm)
- Analyse eines Algorithmus in Hinblick auf den **Aufwand**
  - Kann auf Ebene eines Programms erfolgen
  - Kann auf Ebene der Idee erfolgen (O-Notation)
- Entwickeln eines Verständnisses, ob der Algorithmus **optimal** ist
  - **Asymptotische Komplexität** des Algorithmus ( $O(T(n))$ ) ist dann gleich der **Komplexität des Problems**

# Laufzeiten



# Das In-situ-Sortierproblem

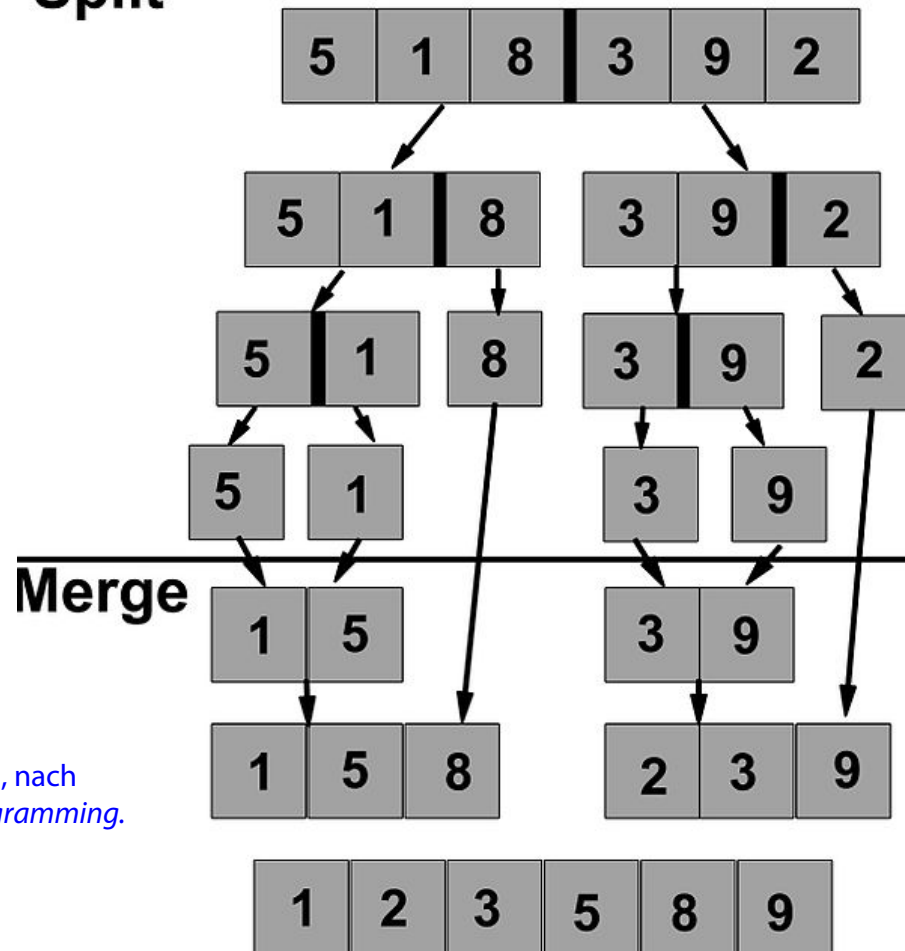
---

- Betrachtete Algorithmen sind quadratisch, d.h. in  $O(n^2)$
- In-situ-Sortierproblem ist nicht schwieriger als quadratisch
  - $n^2$  ist ein Polynom, daher sagen wir,
  - das Problem ist polynomial lösbar (vielleicht aber einfacher)
- Kann man das In-situ-Sortierproblem im typischen Fall schneller lösen?
- In-situ-Sortierprobleme brauchen im allgemeinen Fall mindestens  $n$  Schritte (jedes Element falsch positioniert)
  - Ein vorgeschlagener Algorithmus, der eine konstante Anzahl von Schritten als asymptotische Komplexität hat, kann nicht korrekt sein
  - Was ist mit logarithmisch vielen Schritten?

# In-situ-Sortieren: Geht es schneller?

- Zentrale Idee: Teile und Herrsche

## Split



Entwickelt von John von Neumann, 1945, nach  
Donald E. Knuth: *The Art of Computer Programming*.  
*Volume 3: Sorting and Searching*.  
2 Auflage. Addison-Wesley, 1998, S. 158.



# Entwurfsmuster/-verfahren: Teile und Herrsche

---

Das Entwurfsverfahren *divide-and-conquer* (Teile und Herrsche, *divide et impera*) dient dem Entwurf **rekursiver Algorithmen**.

Die Idee ist es, ein Problem der Größe  $n$  in mehrere gleichartige aber kleinere **Teilprobleme** zu zerlegen (*divide*).

Aus rekursiv gewonnenen Lösungen der Teilprobleme gilt es dann, eine Gesamtlösung für das ursprüngliche Problem zusammensetzen (*conquer*).

**Beispiel:** Sortieren durch Mischen (*merge sort*):

Teile  $n$ -elementige Folge in zwei Teilfolgen der Größe  $n/2 \pm 1$ .

Sortiere die beiden Teilfolgen rekursiv.


Füge die nunmehr sortierten Teilfolgen zusammen durch Reißverschlussverfahren.

# Sortieren durch Mischen


1: **procedure** MERGE-SORT( $A, p, r$ )


2:     ▷ Sortiere  $A[p..r]$

3:     **if**  $p < r$  **then** **Split**


4:          $q \leftarrow \lfloor (p + r) / 2 \rfloor$  

5:         MERGE-SORT( $A, p, q$ )


6:         MERGE-SORT( $A, q + 1, r$ ) 

7:         MERGE( $A, p, q, r$ ) 

1: **procedure** MERGE( $A, p, q, r$ )

2:     ▷ Sortiere  $A[p..r]$ , **Merge** ~~nehme an, dass  $A[p..q]$  und  $A[q + 1..r]$  sortiert~~ 

3:      $i \leftarrow p; j \leftarrow q + 1$

4:     **for**  $k \leftarrow 1$  **to**  $r - p + 1$  **do** 

5:         **if**  $j > r$  **or** ( $i \leq q$  **and**  $A[i] \leq A[j]$ ) **then**

6:              $B[k] \leftarrow A[i]; i \leftarrow i + 1$  **else**  $B[k] \leftarrow A[j]; j \leftarrow j + 1$

7:     **for**  $k \leftarrow 1$  **to**  $r - p + 1$  **do**  $A[k + p - 1] \leftarrow B[k]$

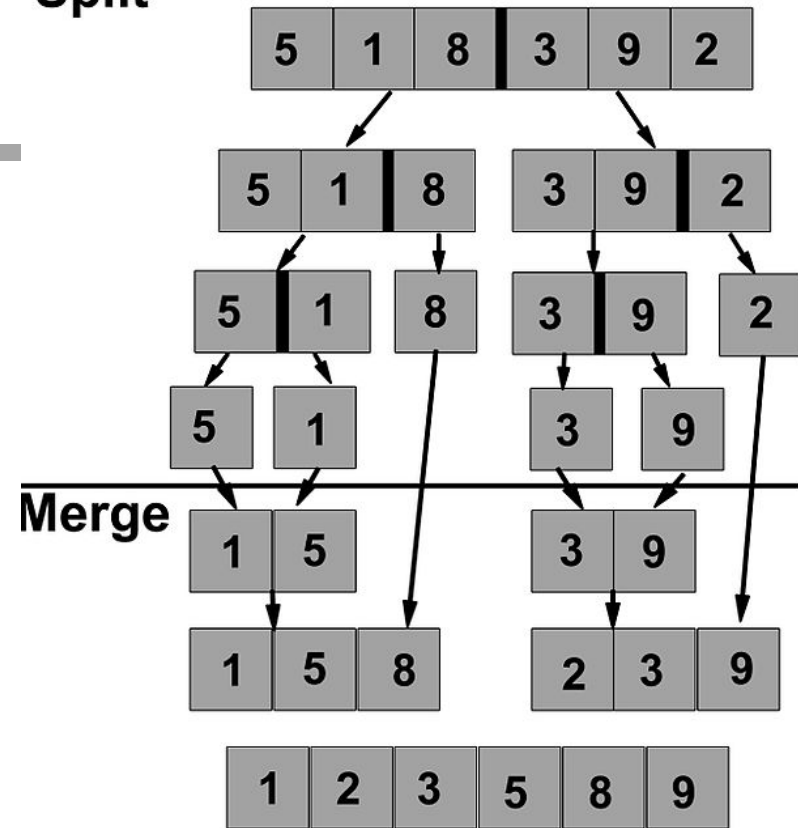


# Sortieren durch Mischen

- 1: **procedure** MERGE-SORT( $A, p, r$ )
- 2:     ▷ Sortiere  $A[p..r]$
- 3:     **if**  $p < r$  **then**
- 4:          $q \leftarrow \lfloor (p + r)/2 \rfloor$
- 5:         MERGE-SORT( $A, p, q$ )
- 6:         MERGE-SORT( $A, q + 1, r$ )
- 7:         MERGE( $A, p, q, r$ )

- 1: **procedure** MERGE( $A, p, q, r$ )
- 2:     ▷ Sortiere  $A[p..r]$ , nehme an, dass  $A[p..q]$  und  $A[q + 1..r]$  sortiert
- 3:      $i \leftarrow p; j \leftarrow q + 1$
- 4:     **for**  $k \leftarrow 1$  **to**  $r - p + 1$  **do**
- 5:         **if**  $j > r$  **or**  $(i \leq q$  **and**  $A[i] \leq A[j])$  **then**
- 6:              $B[k] \leftarrow A[i]; i \leftarrow i + 1$  **else**  $B[k] \leftarrow A[j]; j \leftarrow j + 1$
- 7:     **for**  $k \leftarrow 1$  **to**  $r - p + 1$  **do**  $A[k + p - 1] \leftarrow B[k]$

Split



# Analyse der Merge-Sort-Idee

---

- Was haben wir aufgegeben?
- Speicherverbrauch nicht konstant, sondern von der Anzahl der Elemente von A abhängig:
  - Hilfsfeld B (zwar temporär aber gleiche Länge wie A!)
  - Logarithmisch viele Hilfsvariablen
  - Wir können vereinbaren, dass Letzteres für In-situ-Sortieren noch OK ist
  - Es ist aber kaum OK, eine „Kopie“ B von A anzulegen
- Merge-Sort löst also nicht (ganz) das gleiche Problem wie Insertion-Sort (oder Selection-Sort)
- Problem mit dem Mischspeicher B lässt sich lösen

# Analyse von Merge-Sort

---

Sei  $T(n)$  die Laufzeit von MERGE-SORT.

Das **Aufteilen** braucht  $O(1)$  Schritte.

Die **rekursiven Aufrufe** brauchen  $2T(n/2)$  Schritte.

Das **Mischen** braucht  $O(n)$  Schritte.

Also:

$T(n) = c + 2T(n/2) + c'n$ , wobei die Konstanten für die Ordnung  $O$  irrelevant sind

$$T(n) \approx 2T(n/2) + n$$

NB  $T(1)$  ist irgendein fester Wert.

Die Lösung dieser Rekurrenz ist  $T(n) \in O(n \log(n))$ .

Für große  $n$  ist das besser als  $O(n^2)$  trotz des Aufwandes für die Verwaltung der Rekursion.

NB:  $\log(n)$  bezeichnet den Zweierlogarithmus ( $\log(64) = 6$ ). Bei  $O$ -Notation spielt die Basis des Logarithmus keine Rolle, da alle Logarithmen proportional sind. Z.B.:

$$\log(n) = \ln(n) / \ln(2).$$

# Iterative Expansion

---

Annahme:  $n = 2^k$  (also  $k = \log n$ ).

$$\begin{aligned}T(n) &= 2T(n/2) + n \\&= 2(2T(n/2^2) + n/2) + n \\&= 2^2T(n/2^2) + 2n \\&= 2^2(2T(n/2^3) + n/2^2) + 2n \\&= 2^3T(n/2^3) + 3n \\&= \dots \\&= 2^kT(n/2^k) + kn \\&= nT(1) + n \log n\end{aligned}$$

$$T(n/2) = 2T(n/2^2) + n/2$$

$$T(n/2^2) = 2T(n/2^3) + n/2^2$$

# Analyse von Merge-Sort

---

**Intuitiv:** Rekursionstiefe:  $\log(n)$ , auf dem Tiefenniveau  $k$  hat man  $2^k$  Teilprobleme der Größe jeweils  $g_k := n/2^k$ , jedes verlangt einen Mischaufwand von  $O(g_k) = O(n/2^k)$ . Macht also  $O(n)$  auf jedem Niveau:  $O(n \log(n))$  insgesamt.

**Durch Formalismus:** Sei  $T(n)$  für Zweierpotenzen  $n$  definiert durch

$$T(n) = \begin{cases} 0, & \text{wenn } n = 1 \\ 2T(n/2) + n, & \text{wenn } n = 2^k, k \geq 1 \end{cases}$$

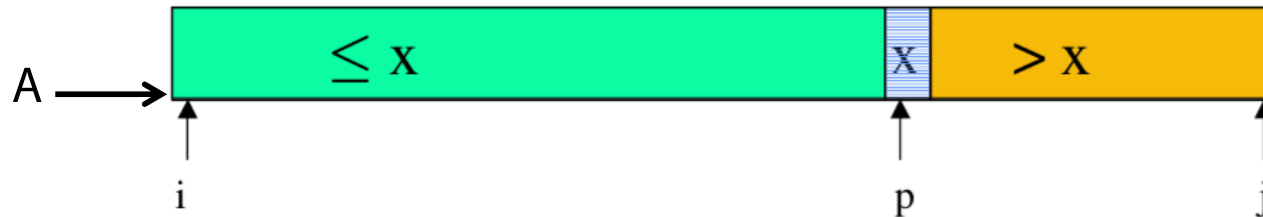
Dann gilt  $T(2^k) = k2^k$ , also  $T(n) = n \log(n)$  für  $n = 2^k$ .

Beweis durch Induktion über  $k$  (oder  $n$ ).

Dass es dann für Nicht-Zweierpotenzen auch gilt, kann man beweisen.

# Quicksort: Vermeidung des Mischspeichers

Idee: wähle „Pivotelement“  $x$  in Feld und stelle Feld so um:

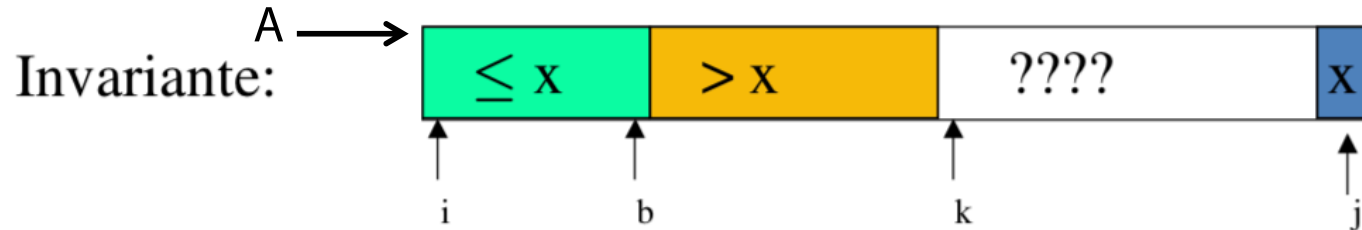


sortiere **Teilfeld der „kleinen“ Elemente ( $\leq x$ )** rekursiv

sortiere **Teilfeld der „großen“ Elemente ( $> x$ )** rekursiv

- 1: procedure QS( $A$ )
- 2:      $n \leftarrow \text{length}(A)$
- 3:     QUICKSORT( $A, 1, n$ )
- 4: procedure QUICKSORT( $A, i, j$ )
- 5:     **if**  $i < j$  **then**
- 6:          $p \leftarrow \text{PARTITION}(A, i, j)$
- 7:         QUICKSORT( $A, i, p - 1$ )
- 8:         QUICKSORT( $A, p + 1, j$ )

# Partitionierung



- 1: **function** PARTITION( $A, i, j$ )
- 2:      $x \leftarrow A[j]$      ▷ Es muss das letzte sein, damit der Code funktioniert.
- 3:      $b \leftarrow i - 1$
- 4:     **for**  $k \leftarrow i$  **to**  $j$  **do**
- 5:         ▷ swap  $A[k]$  and  $A[b + 1]$
- 6:          $temp \leftarrow A[k]$
- 7:          $A[k] \leftarrow A[b + 1]$
- 8:          $A[b + 1] \leftarrow temp$
- 9:         **if**  $A[b + 1] \leq x$  **then**
- 10:              $b \leftarrow b + 1$
- 11:     **return**  $b$

# Analyse von Quicksort

---

- Wenn man „Glück“ hat, liegt der zufällig gewählte Pivotwert nach der Partitionierung immer genau in der Mitte
  - Laufzeitanalyse: Wie bei Merge-Sort
  - Platzanalyse: Logarithmisch viel Hilfsspeicher
- Wenn man „Pech“ hat, liegt der Wert immer am rechten (oder linken) Rand des (Teil-)Intervall
  - Laufzeitanalyse:  $T(n) = n^2$
  - Platzanalyse: Linearer Speicherbedarf
- In der Praxis liegt die Wahrheit irgendwo dazwischen



# Lampsort: Es geht auch nicht-rekursiv

- Führe eine Agenda von Indexbereichen eines Feldes (am Anfang  $[1, n]$ ), auf denen Partition arbeiten muss
- Solange noch Einträge auf der Agenda:
  - Nimm Indexbereich von der Agenda, wenn ein Element im Indexbereich partitioniere und setze zwei entsprechende Einträge auf die Agenda

```
1: procedure LAMP-SORT( $A$ )
2:    $ag \leftarrow newAgenda()$ 
3:   ADD( $ag, [1, length(A)]$ )
4:   while not EMPTY( $ag$ ) do
5:      $[i, j] \leftarrow GET(ag)$ 
6:     if  $i < j$  then
7:        $p \leftarrow PARTITION(A, i, j)$ 
8:       ADD( $ag, [i, p - 1]$ )
9:       ADD( $ag, [p + 1, j]$ )
```

# Analyse von Lampsort

---

- Kontrollfluß und Idee von Teile und Herrsche entkoppelt
  - Benötigt logarithmisch viel Hilfsspeicher (aber kein Mischfeld von der Länge der Eingabe nötig)
  - Asymptotische Zeitkomplexität:  $n \cdot \log n$
  - Leichter parallelisierbar
- 
- Aber: Verstehen wir schon die Komplexität des Problems In-situ-Sortieren?

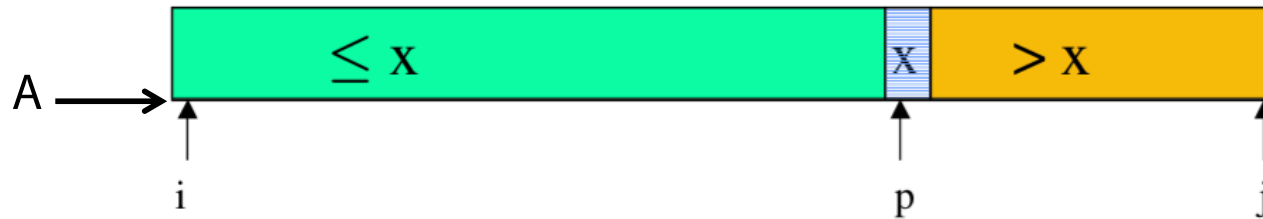
# Unser Referenzproblem: Sortieren eines Feldes

---

- **Gegeben:  $A[1..n] : N$** 
  - Feld (Array)  $A$  von  $n$  Zahlen aus  $N$  (natürliche Zahlen)
- **Gesucht:**
  - Transformation  $S$  von  $A$ , so dass gilt:  $\forall 1 \leq i < j \leq n: A[i] \leq A[j]$
  - Nebenbedingung: Es soll nur logarithmisch viel Hilfsspeicher verwendet werden
- **Erweiterungen:**
  - In den Feldern sind komplexe Objekte enthalten, mit Zahlen als Werte eines sog. **Sortierschlüssels**, sowie weiteren Attributen
  - Bei gleichem Sortierschlüssel soll die Reihenfolge der Objekte bestehen bleiben (**Stabilität**)

# Wiederholung: Quicksort (Teile-und-Herrsche)

Idee: wähle „Pivotelement“  $x$  in Feld und stelle Feld so um:

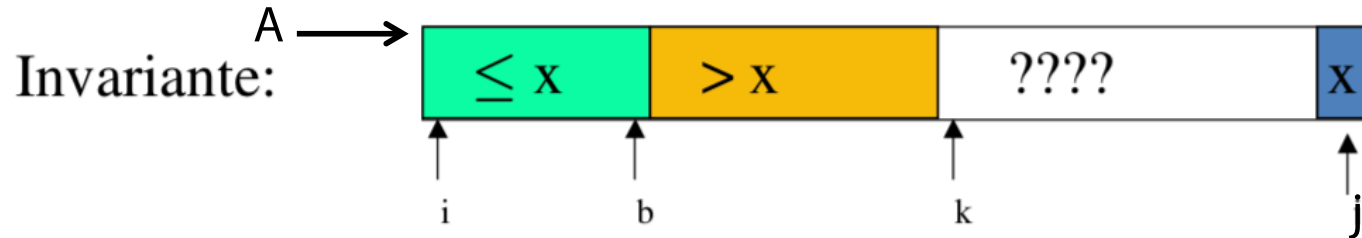


sortiere **Teilfeld der „kleinen“ Elemente ( $\leq x$ )** rekursiv

sortiere **Teilfeld der „großen“ Elemente ( $> x$ )** rekursiv

- 1: procedure QS( $A$ )
- 2:      $n \leftarrow \text{length}(A)$
- 3:     QUICKSORT( $A, 1, n$ )
- 4: procedure QUICKSORT( $A, i, j$ )
- 5:     **if**  $i < j$  **then**
- 6:          $p \leftarrow \text{PARTITION}(A, i, j)$
- 7:         QUICKSORT( $A, i, p - 1$ )
- 8:         QUICKSORT( $A, p + 1, j$ )

# Partitionierung – Stabilität???



- 1: **function** PARTITION( $A, i, j$ )
- 2:      $x \leftarrow A[j]$      ▷ Es muss das letzte sein, damit der Code funktioniert.
- 3:      $b \leftarrow i - 1$
- 4:     **for**  $k \leftarrow i$  **to**  $j$  **do**
- 5:         ▷ swap  $A[k]$  and  $A[b + 1]$
- 6:          $temp \leftarrow A[k]$
- 7:          $A[k] \leftarrow A[b + 1]$
- 8:          $A[b + 1] \leftarrow temp$
- 9:         **if**  $A[b + 1] \leq x$  **then**
- 10:              $b \leftarrow b + 1$
- 11:     **return**  $b$

# Asymptotische Komplexität

---

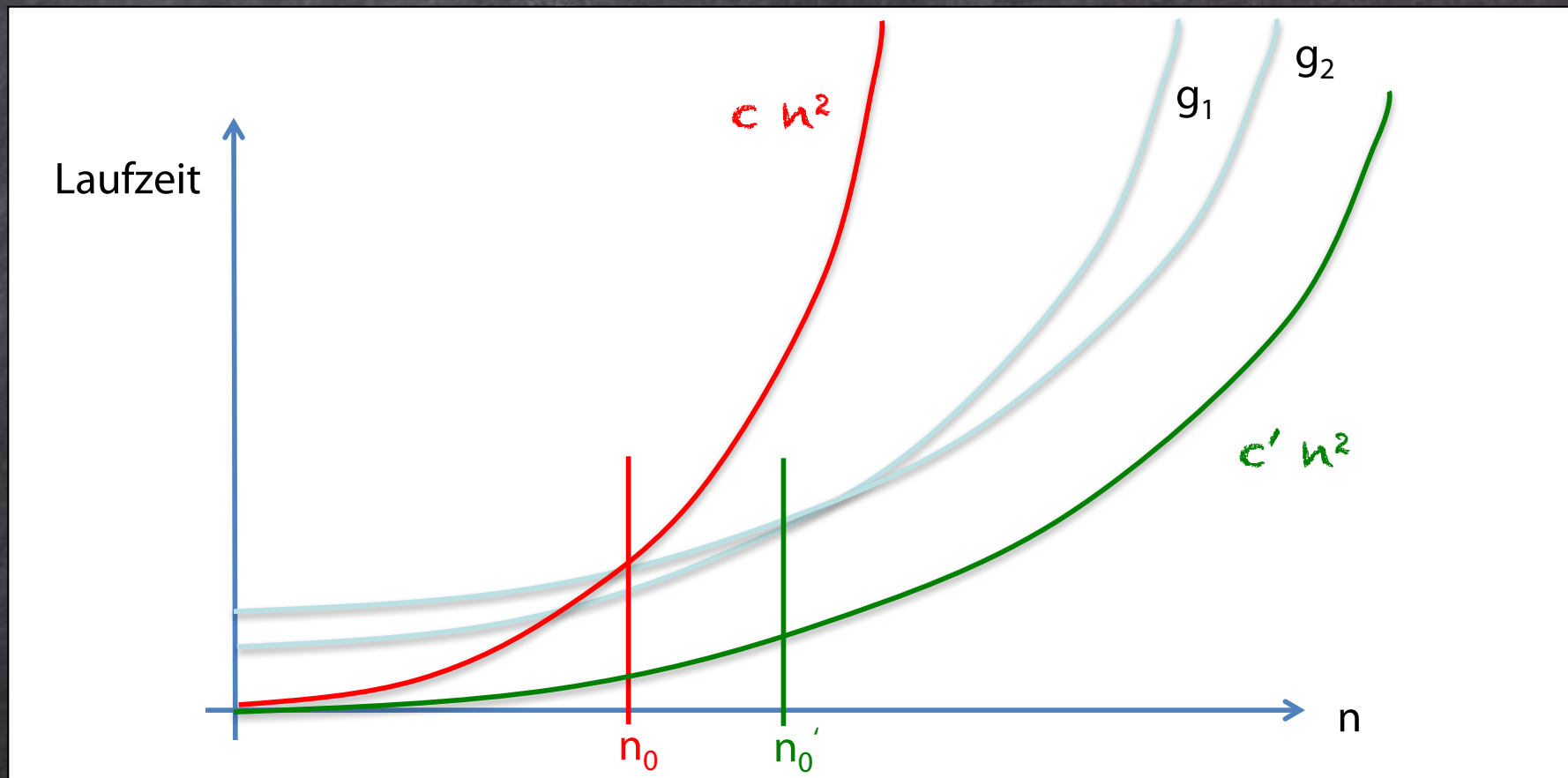
- O-Notation (oberer Deckel)
  - Relativ einfach zu bestimmen für Algorithmen basierend auf dem Verkleinerungsprinzip
  - Nicht ganz einfach für Algorithmen, die nach dem Teile-und-Herrsche-Prinzip arbeiten:

$$T(n) = aT(n/b) + f(n)$$

- Rekursionsbaummethode  
(Ausrollen der Rekursion, Schema erkennen, ggf. Induktion)
- Substitutionsmethode (Lösung raten, einsetzen)
- Master-Methode (kommt später)

# Wiederholung: Quadratischer Aufwand

- Algorithmus 1:  $g_1(n) = b_1 + c_1 * n^2$
- Algorithmus 2:  $g_2(n) = b_2 + c_2 * n^2$



# Asymptotische Komplexität: Notation

- $O(f) = \{g : N \rightarrow N \mid \exists n_0 > 0 : \exists c > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
- $\Omega(f) = \{g : N \rightarrow N \mid \exists n_0 > 0 : \exists c > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$
- $\Theta(f) = O(f) \cap \Omega(f)$

Statt  $g \in O(f)$  mit  $f(n) = n^2$  schreibt man einfach  $g \in O(n^2)$

Einige Autoren schreiben  $g(n) \in O(f(n))$  oder  $g(n) \in O(n^2)$

Man findet sogar  $g = O(n^2)$  oder  $g(n) = O(n^2)$



# Aufgaben

- Ist Selection-Sort in  $\Omega(n^2)$ ?
- Ist Insertion-Sort in  $\Omega(n^2)$ ?
- Ist Insertion-Sort in  $\Theta(n^2)$ ?
- Ist Quicksort in  $\Theta(n \log n)$ ?

# Quicksort

---

- Nur, wenn man „Glück hat“ in  $O(n \log n)$

# Danksagung

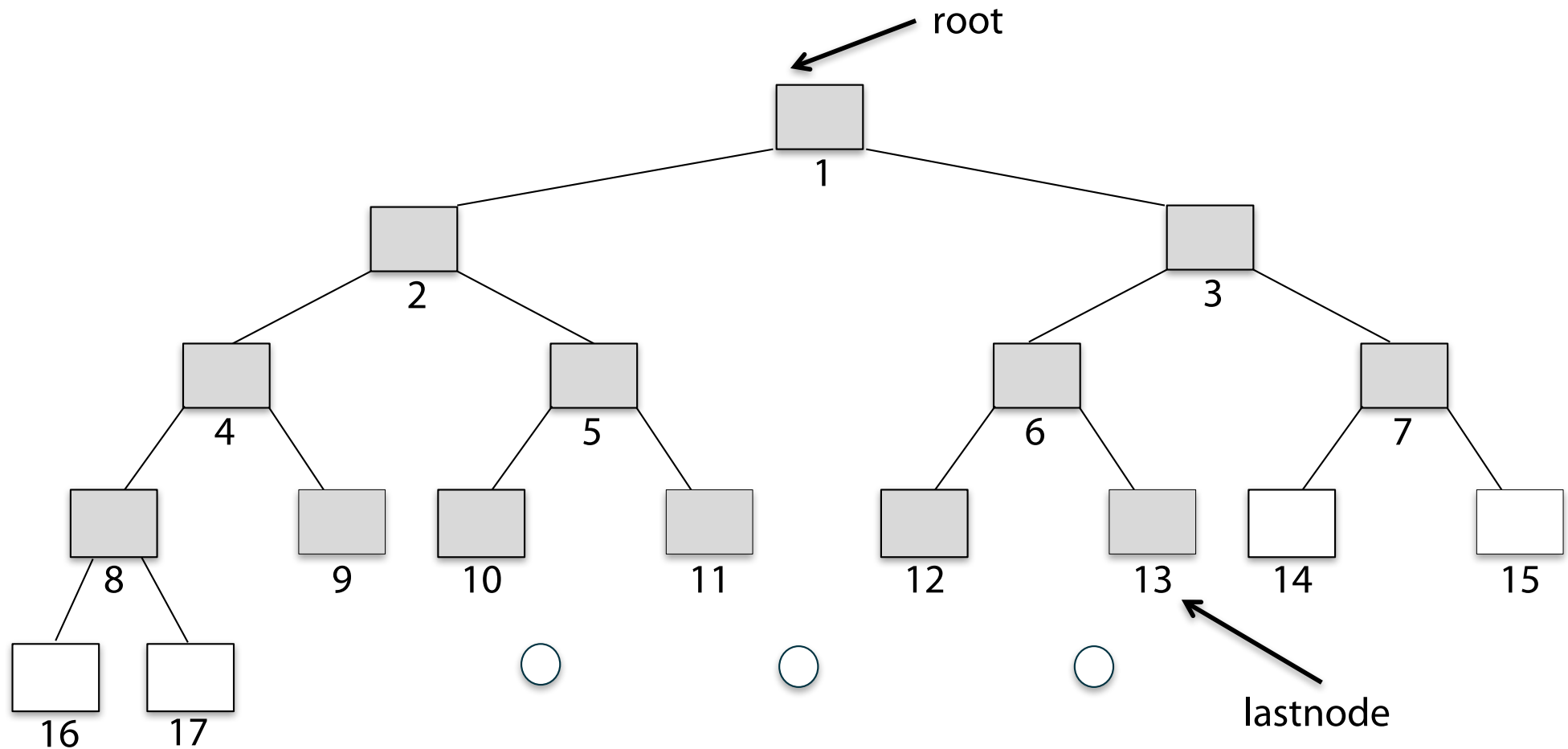
---

Nachfolgende Präsentationen für Heap-Sort und die Komplexität des Problems In-situ-Sortieren sind von Raimund Seidels Präsentationen aus der Vorlesung „Grundzüge von Algorithmen und Datenstrukturen“ aus dem WS 2014/2015 inspiriert.

Siehe <http://www-tcs.cs.uni-sb.de/course/60/>

# Ein Baum ...

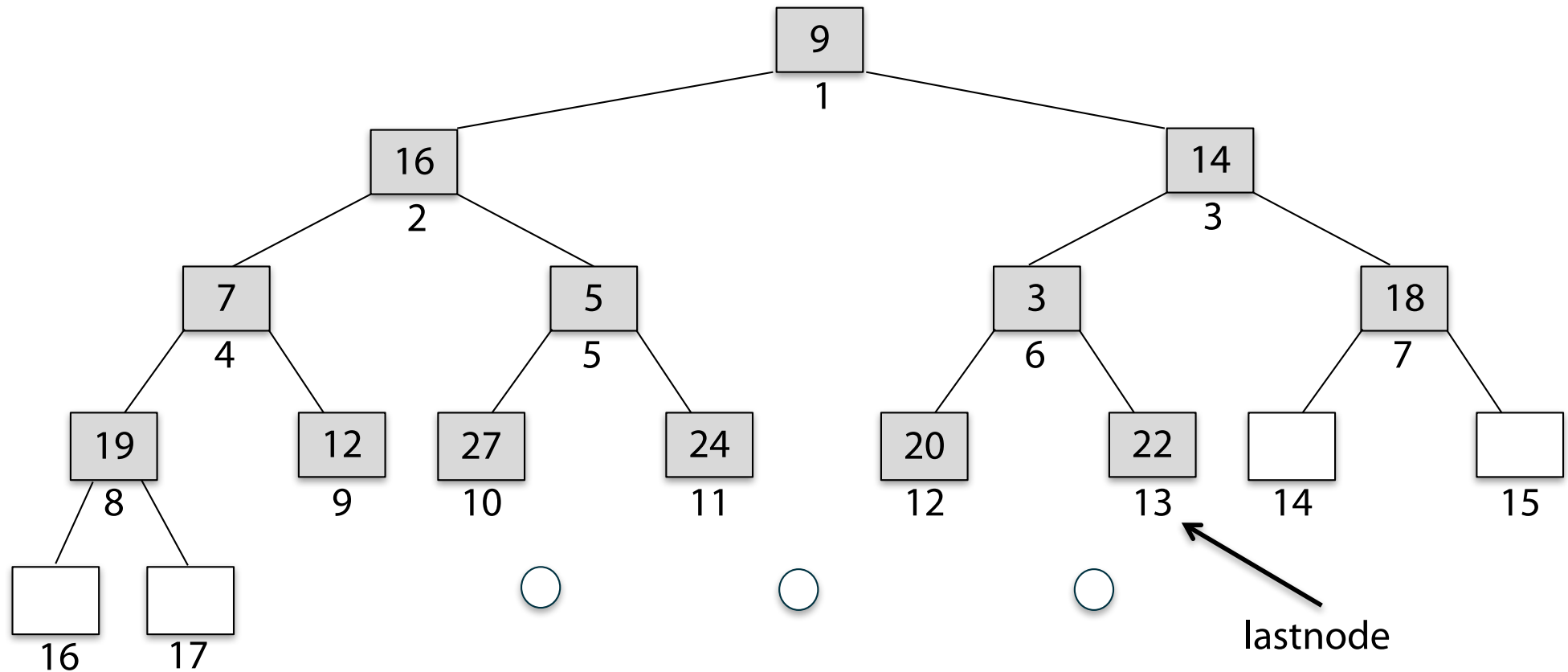
**Beispiel:**  $A = [9, 16, 14, 7, 5, 3, 18, 19, 12, 27, 24, 20, 22]$  mit  $n = 13$



Die „ersten 13“ Knoten (in Niveau-Ordnung) in einem größeren binären Baum

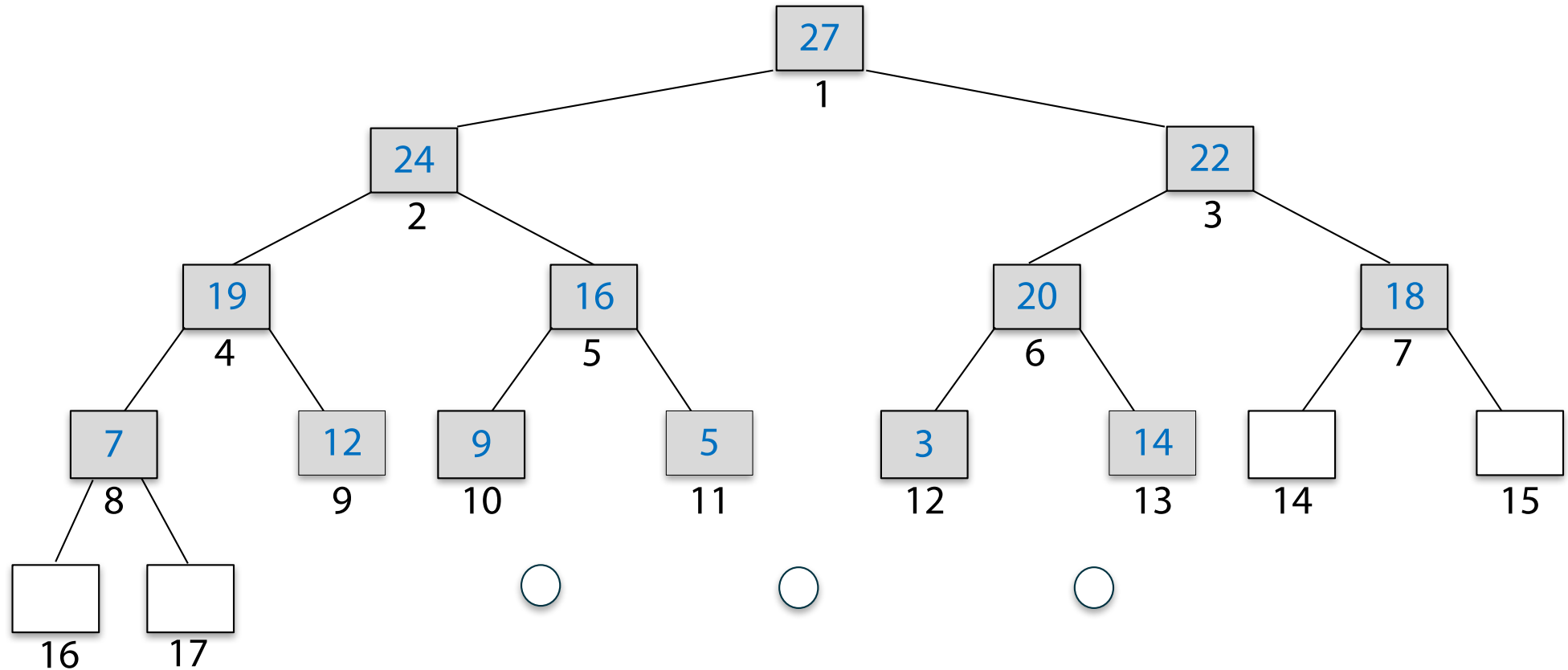
# ... mit Werten

**Beispiel:**  $A = [9, 16, 14, 7, 5, 3, 18, 19, 12, 27, 24, 20, 22]$  mit  $n = 13$



$A[1..13]$  in den „ersten“ 13 Knoten eines größeren binären Baums

# Umgestellt als Max-Heap

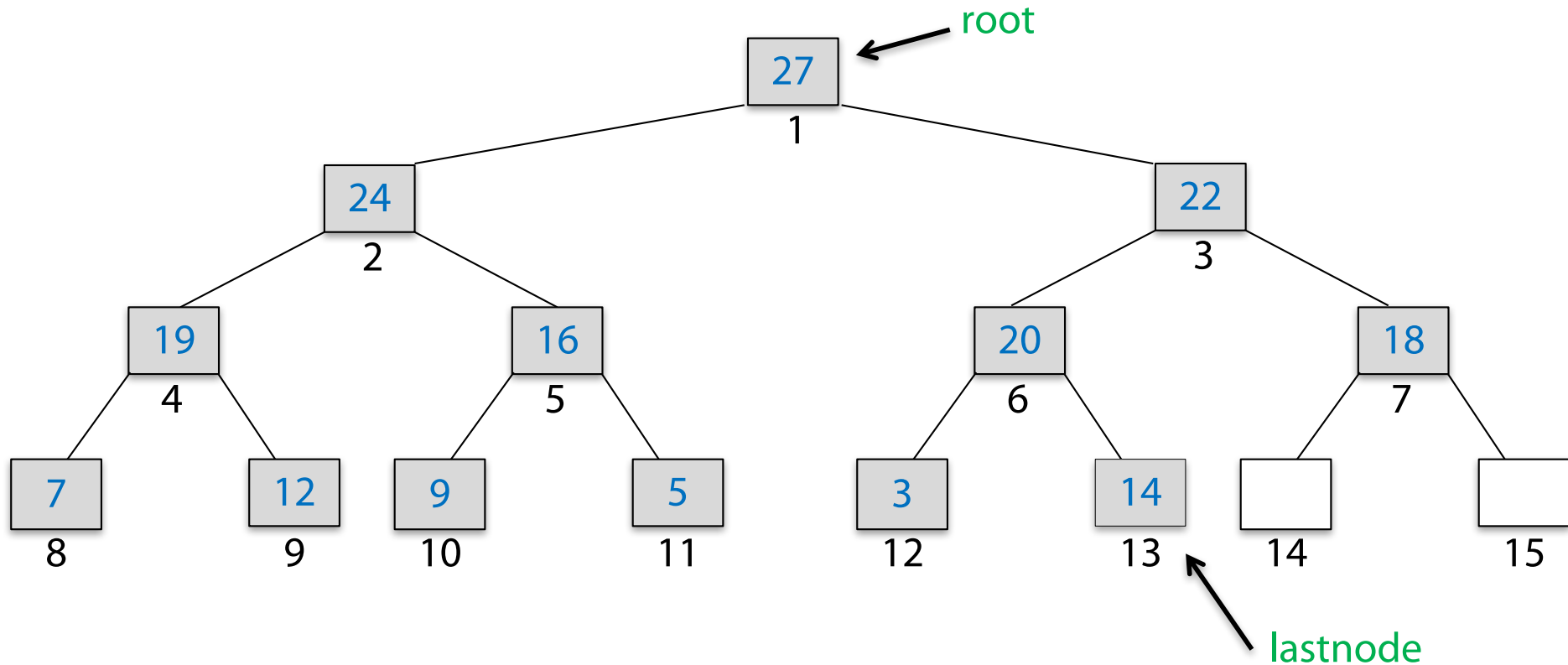


In einem Max-Heap gilt für **jeden** Knoten  $v$  die Eigenschaft:  
sein Schlüssel ist zumindest so groß wie der jedes seiner Kinder  
( für jedes Kind  $c$  von  $v$  gilt:  $key(v) \geq key(c)$  )

Im Max-Heap steht der größte Schlüssel immer an der Wurzel

# Sortierung mit einem Max-Heap

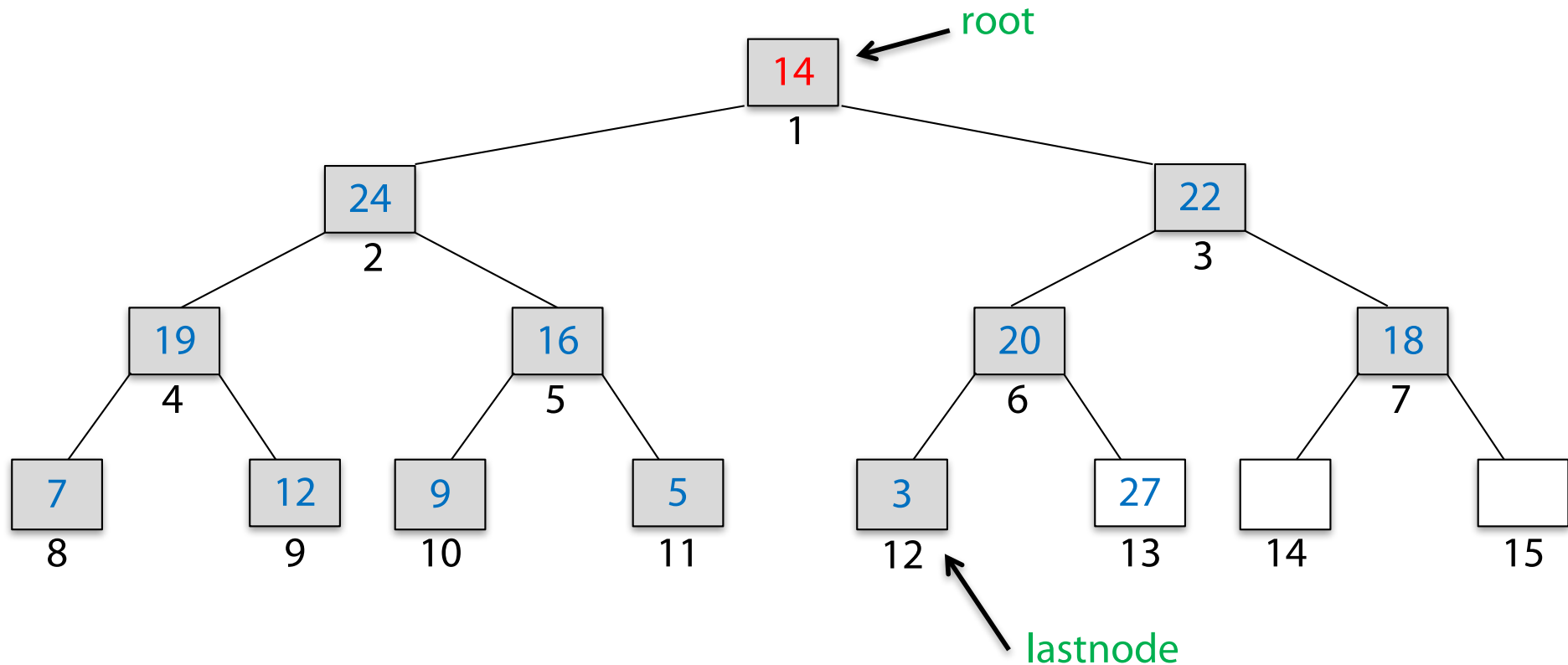
**Beachte:** In Max-Heap steht der größte Schlüssel immer bei der Wurzel.



**Idee:**

1. Tausche Schlüssel von **root** und **lastnode** und ziehe **lastnode** aus der Betrachtung

# Sortierung mit einem Max-Heap

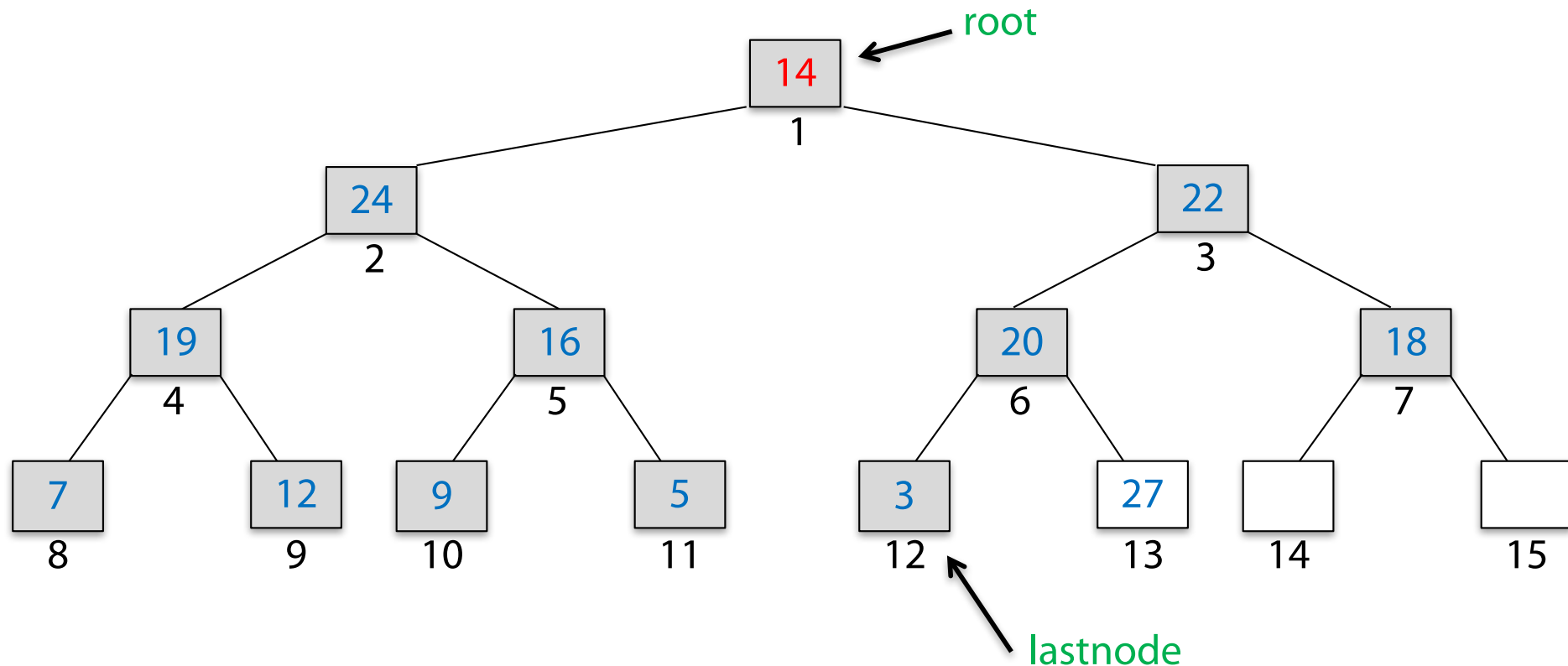


## Idee:

1. Tausche Schlüssel von **root** und **lastnode** und ziehe **lastnode** aus der Betrachtung



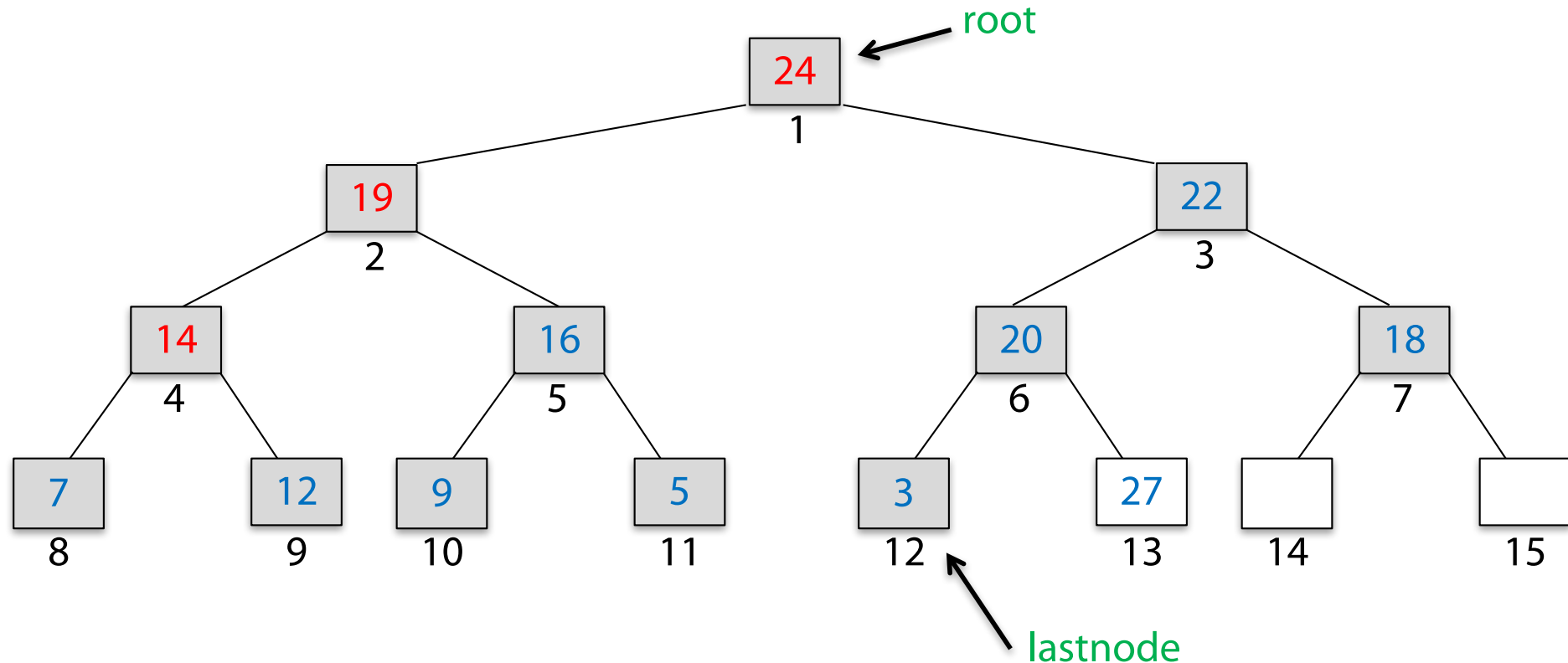
# Sortierung mit einem Max-Heap



## Idee:

1. Tausche Schlüssel von **root** und **lastnode** und ziehe **lastnode** aus der Betrachtung
2. Mache den "Beinahe-Max-Heap" (die Max-Heap-Eigenschaft ist bei der Wurzel verletzt) zu einem Max-Heap

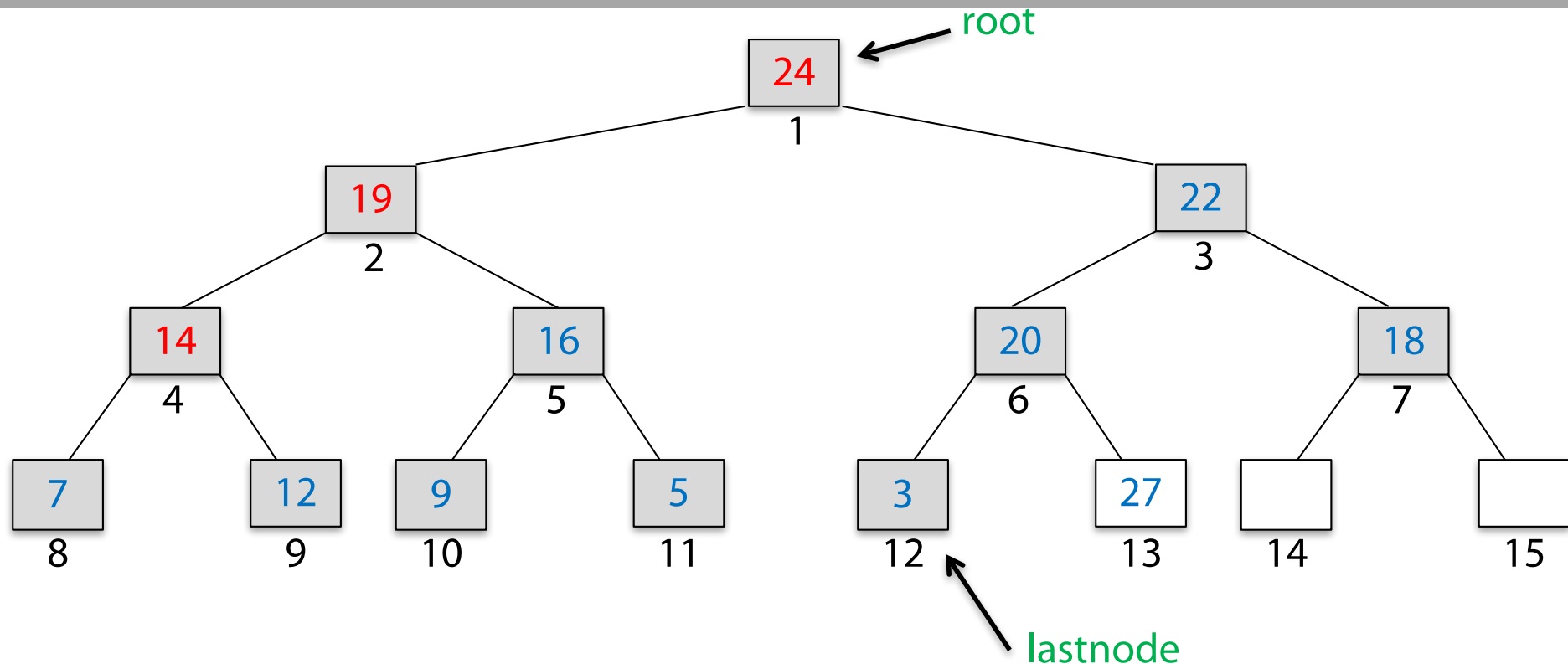
# Wiederherstellung des Max-Heaps: Einsieben



## Idee:

1. Tausche Schlüssel von **root** und **lastnode** und ziehe **lastnode** aus der Betrachtung
2. Mache den "Beinahe-Max-Heap" (die Max-Heap-Eigenschaft ist bei der Wurzel verletzt) zu einem Max-Heap (ggf. mit Einsieben in das Kind mit dem größten Schlüssel)

# Verkleinerungsprinzip + Max-Heap-Invariante

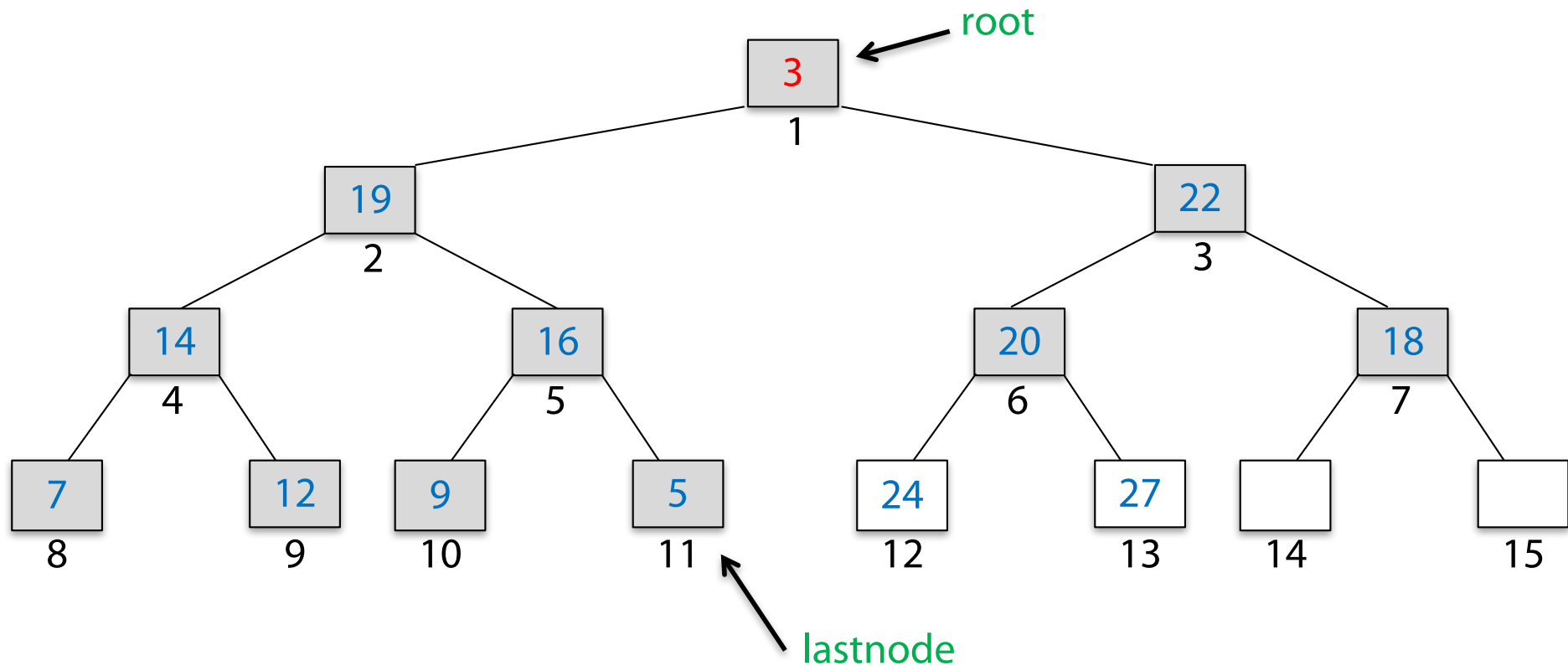


## Idee:

1. Tausche Schlüssel von **root** und **lastnode** und ziehe **lastnode** aus der Betrachtung.
2. Mache den "Beinahe-Max-Heap" (die Max-Heap-Eigenschaft ist bei der Wurzel verletzt) zu einem Max-Heap.

Der betrachtete, um eins kleinere Max-Heap enthält nur kleinere Schlüssel. **Diese müssen nun sortiert werden.** Dieses Sortieren kann durch Wiederholen der eben verwendeten Methode geschehen.

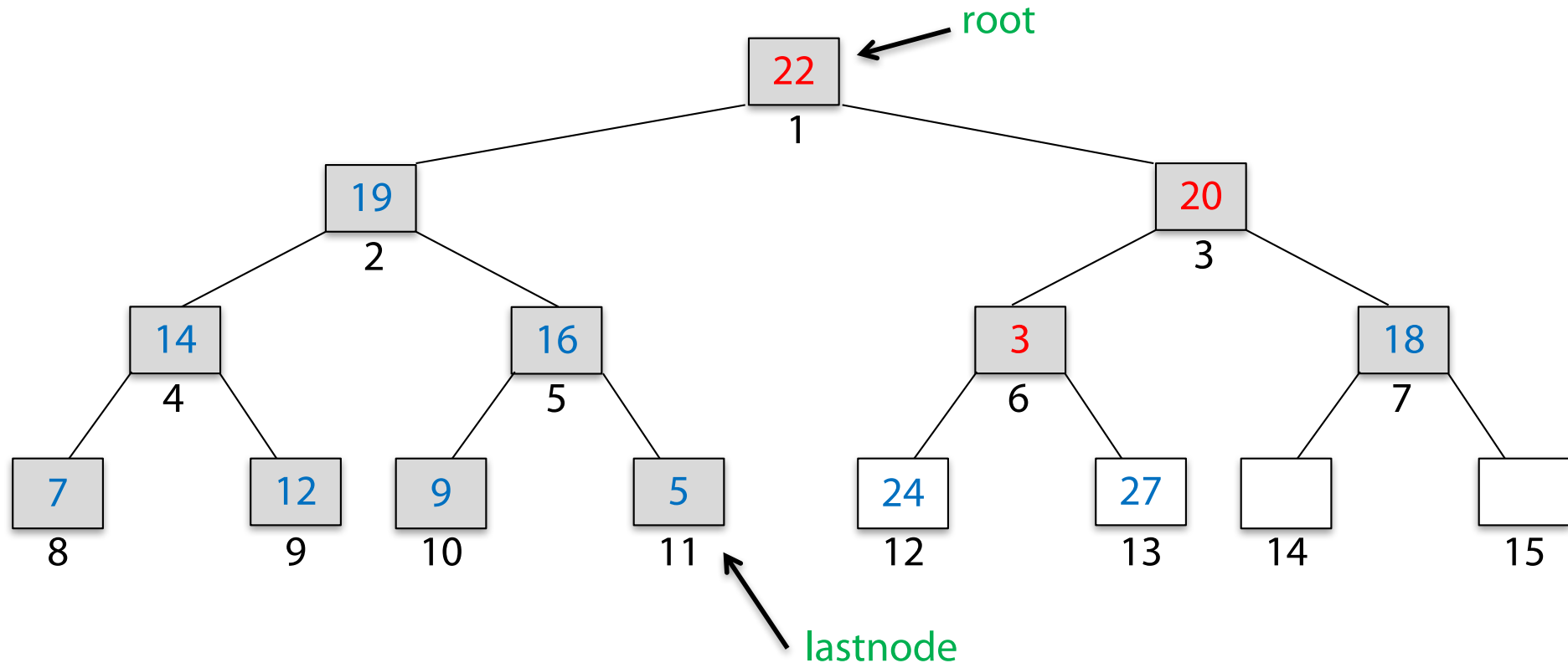
# Nach der Vertauschung...



## Idee:

1. Tausche Schlüssel von **root** und **lastnode** und ziehe **lastnode** aus der Betrachtung

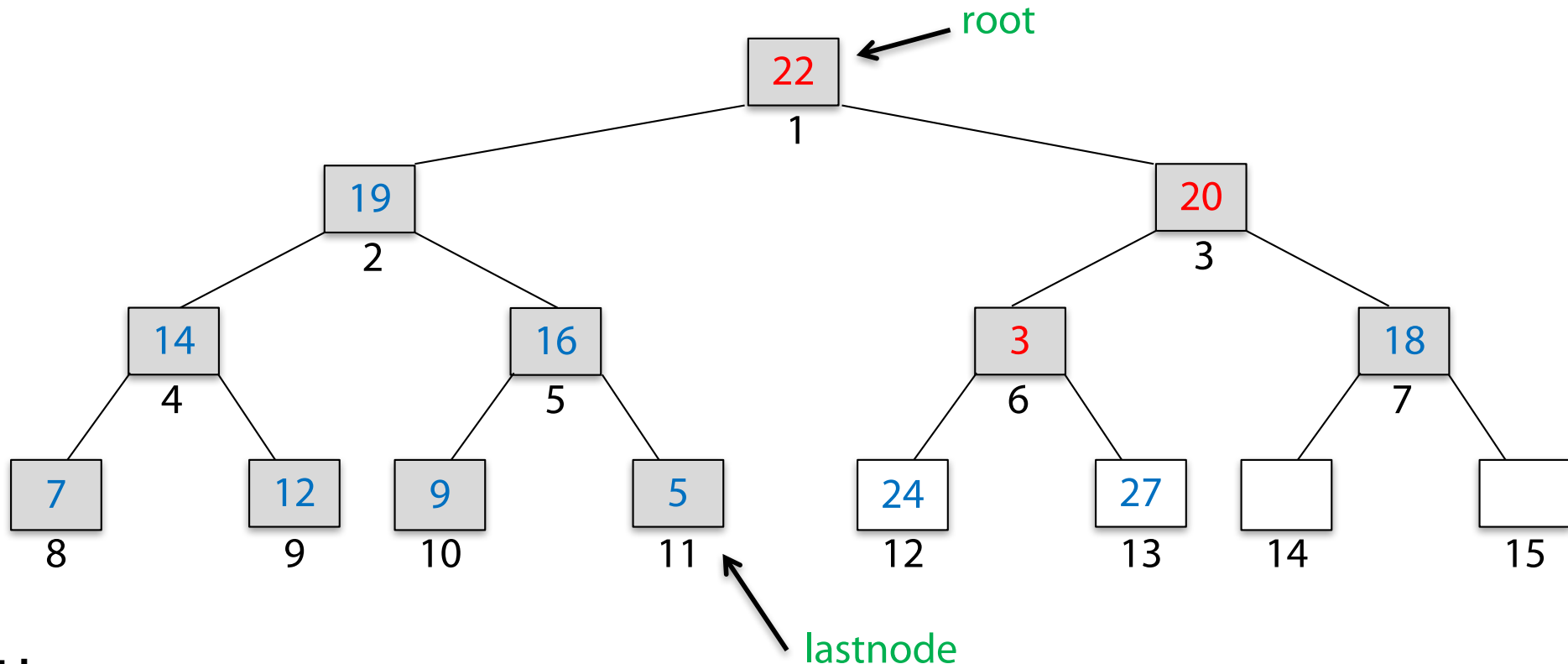
# ... und dem Einsieben



## Idee:

1. Tausche Schlüssel von **root** und **lastnode** und ziehe **lastnode** aus der Betrachtung
2. Mache den "Beinahe-Max-Heap" (die Max-Heap-Eigenschaft ist bei der Wurzel verletzt) zu einem Max-Heap

# Und weiter geht's ...



## Idee:

1. Tausche Schlüssel von **root** und **lastnode** und ziehe **lastnode** aus der Betrachtung.
2. Mache den "Beinahe-Max-Heap" (die Max-Heap-Eigenschaft ist bei der Wurzel verletzt) zu einem Max-Heap.

Der betrachtete, um eins kleinere Max-Heap enthält nur kleinere Schlüssel. **Diese müssen nun sortiert werden.** Dieses Sortieren kann durch Wiederholen der eben verwendeten Methode geschehen.

# Heap-Sort

```
1: procedure HEAP-SORT( $A$ )
2:    $lastnode \leftarrow length(A)$ 
3:   MAKEHEAP( $A, lastnode$ )
4:    $root \leftarrow 1$ 
5:   while  $lastnode \neq root$  do
6:      $temp \leftarrow A[root]$ 
7:      $A[root] \leftarrow A[lastnode]$ 
8:      $A[lastnode] \leftarrow temp$ 
9:      $lastnode \leftarrow lastnode - 1$ 
10:    HEAPIFY( $A, root$ )
```

Wir betrachten  $lastnode$  als globale Variable  
(besser: als Parameter in Unterfunktionen übergeben)

▷ swap key of  $root$  and  $lastnode$

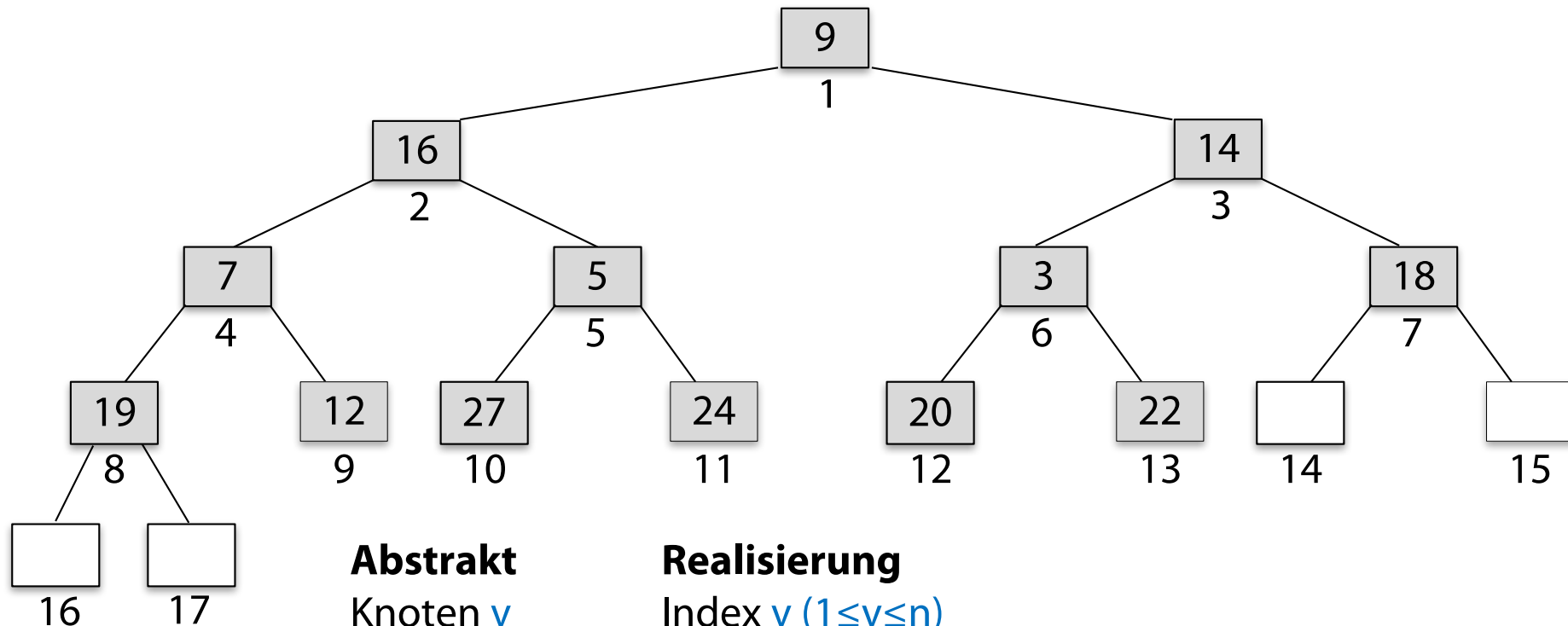
Robert W. Floyd: Algorithm 113: Treesort.  
In: Communications of the ACM. 5, Nr. 8, S. 434, **1962**

Robert W. Floyd: Algorithm 245: Treesort 3.  
In: Communications of the ACM. 7, Nr. 12, S. 701, **1964**

J. W. J. Williams: Algorithm 232: Heapsort.  
In: Communications of the ACM. 7, Nr. 6, S. 347-348, **1964**



# Realisierung des gewünschten Binärbaums im Feld $A[1..n]$



## Abstrakt

Knoten  $v$

$key(v)$

root

lastnode

leftchild( $v$ )

rightchild( $v$ )

parent( $v$ )

exist( $v$ )

is-leaf( $v$ )

## Realisierung

Index  $v$  ( $1 \leq v \leq n$ )

$A[v]$

1

$n$

$2 \cdot v$

$2 \cdot v + 1$

$\lfloor v/2 \rfloor$

$(v \leq n)$

$(v > n/2)$

- Realisierung von  $2 \cdot v$  für  $v$  eine natürliche Zahl?
- Realisierung von  $\lfloor v/2 \rfloor$ ?



# Make-Heap

---

```
1: procedure MAKEHEAP( $A, n$ )
2:    $p \leftarrow \text{PARENT}(n)$ 
3:    $root \leftarrow 1$ 
4:   for  $v \leftarrow p$  downto  $root$  do    ▷ Consider all inner nodes in descending order
5:     HEAPIFY( $A, v$ )
```

- Betrachte einen Knoten nach dem anderen, die Kinder sollten schon Heaps (Max-Heaps) sein
- Verwende Heapify um Beinahe-Heap zu Heap zu machen
- Kinder eines Knoten sind schon Wurzeln von Heaps, wenn man rückwärts vorgeht (beginnend beim Vater von lastnode)
- Zeitverbrauch: Sicherlich in  $O(n \log n)$

# Heapify

---

```
1: procedure HEAPIFY( $A, k$ )
2:   if not ISLEAF( $A, k$ ) then
3:      $left \leftarrow$  LEFTCHILD( $k$ )
4:      $right \leftarrow$  RIGHTCHILD( $k$ )
5:      $maxc \leftarrow left$ 
6:     if EXISTS( $A, right$ ) then
7:       if  $A[right] > A[left]$  then
8:          $maxc \leftarrow right$ 
9:     if  $A[maxc] > A[k]$  then
10:       $temp \leftarrow A[maxc]$ 
11:       $A[maxc] \leftarrow A[k]$ 
12:       $A[k] \leftarrow temp$ 
13:      HEAPIFY( $A, maxc$ )
```

▷  $k$ ... currently considered node

▷ determine the biggest child

▷ swap key of  $maxc$  and  $k$

Statt „Heapify“ wird oft auch der Ausdruck „Einsieben“ verwendet.

# Hilfsfunktionen

---

```
1: function LEFTCHILD(v)
2:   return  $2 * v$ 
3: function RIGHTCHILD(v)
4:   return  $2 * v + 1$ 
5: function PARENT(v)
6:   return  $\lfloor v/2 \rfloor$ 
7: function EXISTS(A, v)
8:   if  $v \leq \text{length}(A)$  then
9:     return  $v \leq \text{lastnode}$ 
10:  return false
11: function ISLEAF(A, v)
12:  if  $v > (\text{length}(A)/2)$  then
13:    return true
14:  return  $2*v > \text{lastnode}$ 
```

# Heap-Sort

```
1: procedure HEAP-SORT( $A$ )
2:    $lastnode \leftarrow length(A)$ 
3:   MAKEHEAP( $A, lastnode$ )
4:    $root \leftarrow 1$ 
5:   while  $lastnode \neq root$  do
6:      $temp \leftarrow A[root]$ 
7:      $A[root] \leftarrow A[lastnode]$ 
8:      $A[lastnode] \leftarrow temp$ 
9:      $lastnode \leftarrow lastnode - 1$ 
10:    HEAPIFY( $A, root$ )
```

▷ swap key of  $root$  and  $lastnode$

- $O(n \log n)$  für Make-Heap
- $O(n \log n)$  für While-Schleife
- Gesamtlaufzeit  $O(n \log n)$

Robert W. Floyd: Algorithm 113: Treesort.  
In: Communications of the ACM. 5, Nr. 8, S. 434, 1962

Robert W. Floyd: Algorithm 245: Treesort 3.  
In: Communications of the ACM. 7, Nr. 12, S. 701, 1964

J. W. J. Williams: Algorithm 232: Heapsort.  
In: Communications of the ACM. 7, Nr. 6, S. 347-348, 1964

# Wie "langsam" muss Sortieren sein?

**Frage:** Gibt es Sortieralgorithmen mit Laufzeit  $\underbrace{\quad}_{\leq n \log n}$  ?

Beschränke Betrachtung auf  
**Vergleichsbasierte Algorithmen**

- Vergleich ob  $<$  ,  $=$  ,  $>$  ist die einzige erlaubte Operation auf Schlüsseln  
(außer Kopieren oder im Speicher Bewegen)
- Algorithmus muss für jeden Typ von Schlüssel funktionieren, solange  $<$  ,  $=$  ,  $>$  definiert sind und eine totale Ordnung auf den Schlüsseln darstellen

z.B. unzulässig: arithmetische Operationen auf Schlüsseln,  
Verwendung von Schlüssel als Index in Feld



Wenn die Eingabegröße fixiert wird, kann jeder vergleichsbasierte Algorithmus als schleifenfreies Programm von **if-Statements** geschrieben werden

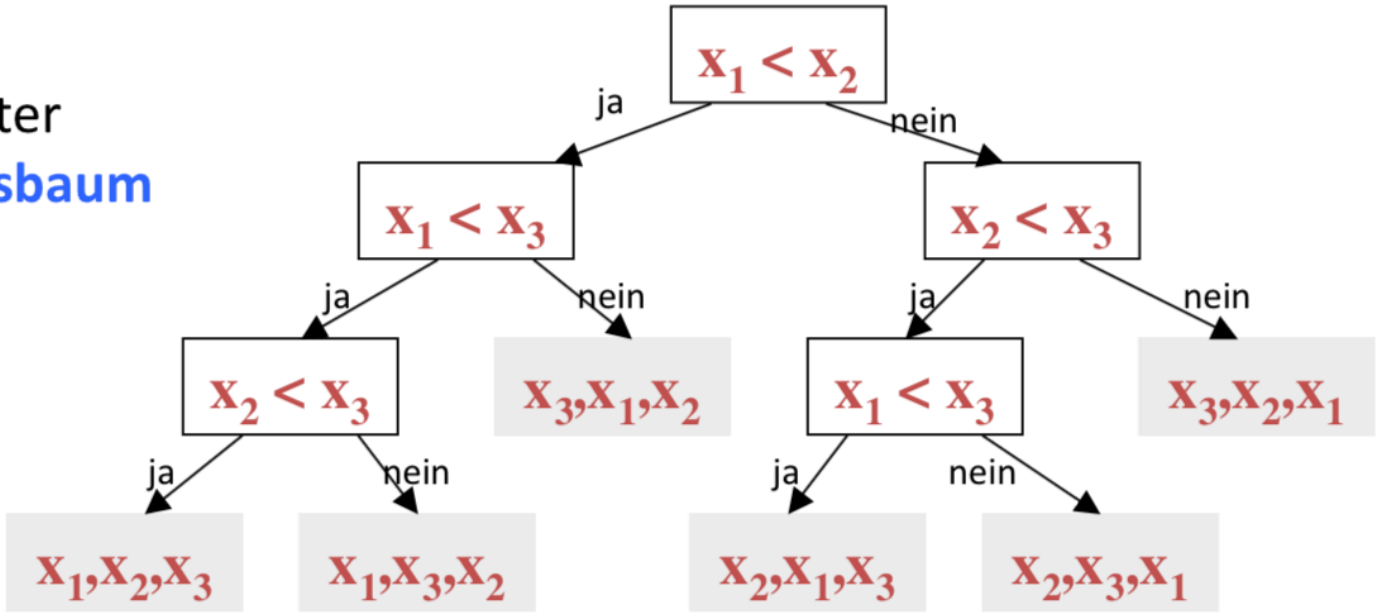
Bsp.: Programm um  $n=3$  Schlüssel  $x_1, x_2, x_3$  zu sortieren

```

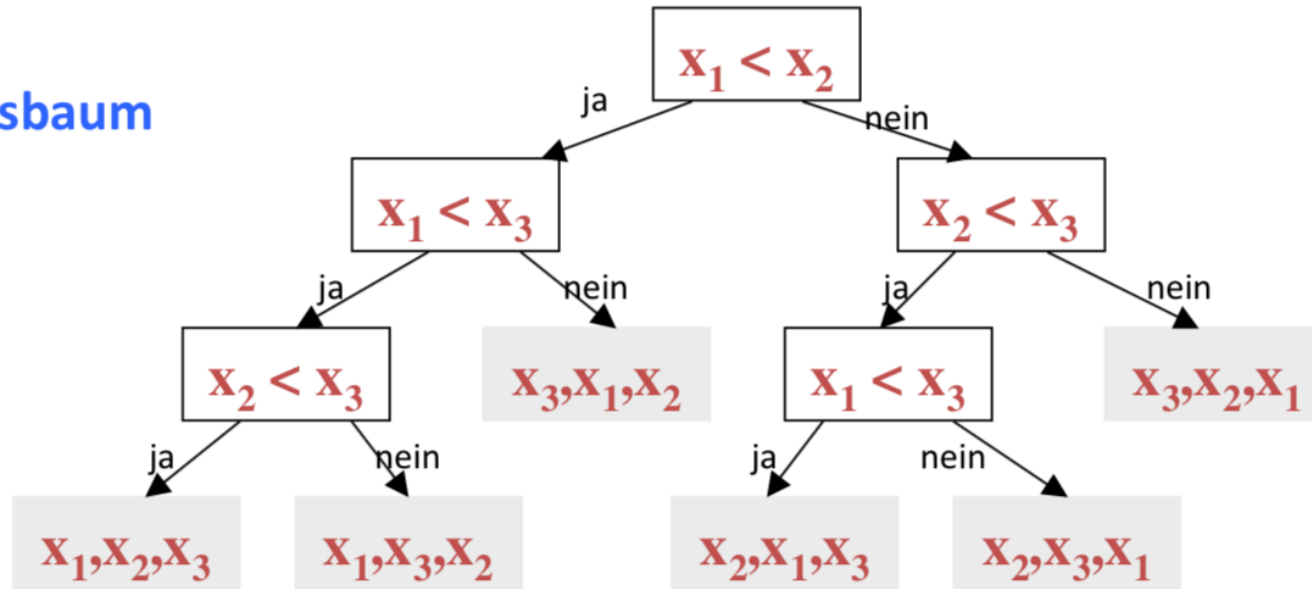
if  $x_1 < x_2$  then if  $x_1 < x_3$  then if  $x_2 < x_3$  then output  $x_1, x_2, x_3$ 
                                else output  $x_1, x_3, x_2$ 
                                else output  $x_3, x_1, x_2$ 
else if  $x_2 < x_3$  then if  $x_1 < x_3$  then output  $x_2, x_1, x_3$ 
                                else output  $x_2, x_3, x_1$ 
else output  $x_3, x_2, x_1$ 

```

Äquivalenter Entscheidungsbaum



## Entscheidungsbaum



- Programmdurchlauf entspricht Wurzel-Blatt Pfad
- Länge des Pfades entspricht Anzahl der Schlüsselvergleiche bei diesem Programmdurchlauf
- Blatt entspricht der (sortierten) Ausgabepermutation der Eingabe
- Worst-case Laufzeit des Programmes entspricht dem längsten Wurzel-Blatt Pfad im Baum, also der Höhe des Baums.
- Zu zeigen:  
**Jeder Entscheidungsbaum fürs Sortieren muss große Höhe haben.**

- Programmdurchlauf entspricht Wurzel-Blatt Pfad
- Länge des Pfades entspricht Anzahl der Schlüsselvergleiche bei diesem Programmdurchlauf
- Blatt entspricht der (sortierten) Ausgabepermutation der Eingabe
- Worst-case Laufzeit des Programmes entspricht dem längsten Wurzel-Blatt Pfad im Baum, also der Höhe des Baums.
- Zu zeigen:

**Jeder Entscheidungsbaum fürs Sortieren muss große Höhe haben.**

**B** Entscheidungsbaum, um  $n$  Schlüssel zu sortieren

$$\#Blätter(B) \geq n! \quad \boxed{\text{(mindestens ein Blatt für jede der } n! \text{ Eingabepermutationen)}}$$

$$\#Blätter(B) \leq 2^{\text{Höhe}(B)}$$

$$\begin{aligned} \text{Höhe}(B) &\geq \log_2 (\#Blätter(B)) \\ &\geq \log_2 n! \end{aligned}$$



# Komplexität des Problems „In-situ-Sortieren“

---

$$\log n! = \sum_{i=1}^n \log_2 i \geq \sum_{i=n/2}^n \log_2 i \geq \frac{n}{2} \log_2 \left(\frac{n}{2}\right)$$

$$\frac{n}{2} \log_2 \left(\frac{n}{2}\right) = \frac{n}{2} (\log_2(n) - 1) \in \Omega(n \log n)$$

- **Mindestens  $n \log n$**  viele Schritte im schlechtesten Fall
- Mit Heap-Sort haben wir auch festgestellt, dass nur **maximal  $n \log n$**  viele Schritte im schlechtesten Fall nötig sind
- Das In-situ-Sortierproblem ist in der **Klasse der Probleme, die deterministisch mit  $n \log n$  Schritten** gelöst werden können

# Einsichten

---

- Merge-Sort und Heap-Sort besitzen asymptotisch optimale Laufzeit
- Heapsort in  $O(n \log n)$ , aber aufwendige Schritte
- Wenn die erwartete Aufwandsfunktion von Quicksort in  $O(n \log n)$ , dann einfachere Schritte
  - Quicksort dann i.a. schneller ausführbar auf einem konkreten Computer

# Randbemerkung: Timsort

---

- Von Merge-Sort und Insertion-Sort abgeleitet (2002 von Tim Peters für Python)
- Mittlerweile auch in Java SE 7 und Android genutzt
- Idee: Ausnutzung von Vorsortierungen

## Komplexität und Effizienz [\[Bearbeiten\]](#)

---

Wie Mergesort ist Timsort ein **stabiles, vergleichsbasiertes Sortierverfahren** mit einer Best-Case-Komplexität von  $\Theta(n)$  und einer Worst- und Average-Case-Komplexität von  $\mathcal{O}(n \cdot \log(n))$ .<sup>[4]</sup>

Nach der **Informationstheorie** kann kein vergleichsbasiertes Sortierverfahren mit weniger als  $\Omega(n \log n)$  Vergleichen im Average-Case auskommen. Auf realen Daten braucht Timsort oft deutlich weniger als  $\Omega(n \log n)$  Vergleiche, weil es davon profitiert, dass Teile der Daten schon sortiert sind.<sup>[5]</sup>

- Man sieht also:  $\mathcal{O}$ ,  $\Omega$ , und  $\Theta$  werden tatsächlich in der öffentlichen Diskussion verwendet, sollte man also verstehen.

<http://de.wikipedia.org/wiki/Timsort>

# Zusammenfassung

---



- Problemspezifikation
  - Beispiel Sortieren mit Vergleichen
- Problemkomplexität
- Algorithmenanalyse:
  - Asymptotische Komplexität (O-Notation)
  - Bester, typischer und schlimmster Fall
- Entwurfsmuster für Algorithmen
  - Ein-Schritt-Berechnung (nicht immer einfach zu sehen, dass es geht)
  - Verkleinerungsprinzip + Invarianten
  - Teile und Herrsche