Algorithmen und Datenstrukturen

Sortierung durch Vergleichen: Quicksort, Ω-Notation: Mindestaufwand

Prof. Dr. Ralf Möller

Universität zu Lübeck

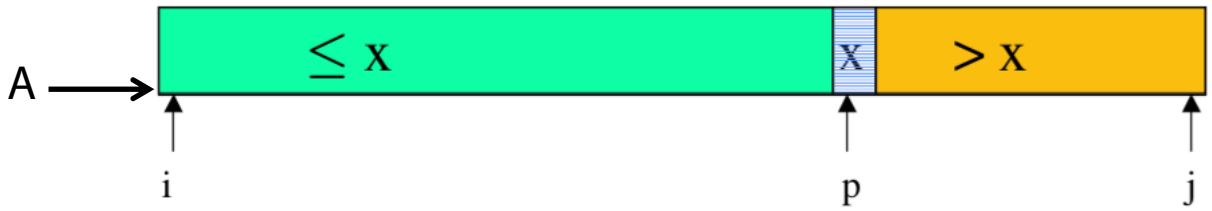
Institut für Informationssysteme

Felix Kuhr (Übungen)

sowie viele Tutoren

Quicksort: Vermeidung des Mischspeichers

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



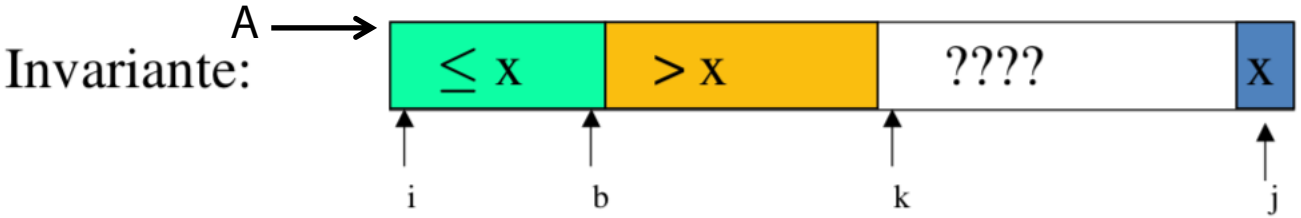
sortiere Teilfeld der „kleinen“ Elemente ($\leq$ x) rekursiv

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

```
1: procedure QS(A)
2:     n ← length(A)
3:     QUICKSORT(A, 1, n)
4: procedure QUICKSORT(A, i, j)
5:     if i < j then
6:         p ← PARTITION(A, i, j)
7:         QUICKSORT(A, i, p − 1)
8:         QUICKSORT(A, p + 1, j)
```

C. A. R. Hoare: *Quicksort*.
In: *The Computer Journal*. 5(1) , S. 10–15, **1962**

Partitionierung

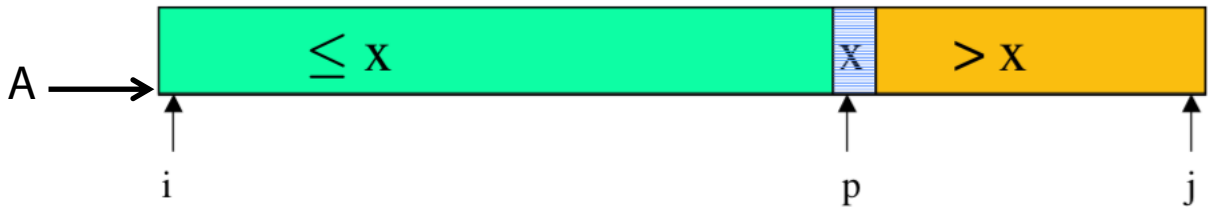



```
1:  function PARTITION(A, i, j)
2:      x ← A[j]        ▷ Es muss das letzte sein, damit der Code funktioniert.
3:      b ← i − 1
4:      for k ← i to j do
5:          ▷ swap A[k] and A[b + 1]
6:          temp ← A[k]
7:          A[k] ← A[b + 1]
8:          A[b + 1] ← temp
9:          if A[b + 1]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

$T_{partition}(n) \in O(n)$

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 ← length(A)
3:     QUICKSORT(A, 1, n)
4: procedure QUICKSORT(A, i, j)
5:     if i < j then
6:         p ← PARTITION(A, i, j)
7:         QUICKSORT(A, i, p − 1)
8:         QUICKSORT(A, p + 1, j)
```

C. A. R. Hoare: *Quicksort*.
In: *The Computer Journal*. 5(1) , S. 10–15, **1962**

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
- Im typischen Fall liegt die Wahrheit irgendwo dazwischen

Lampsort: Es geht auch nicht-rekursiv

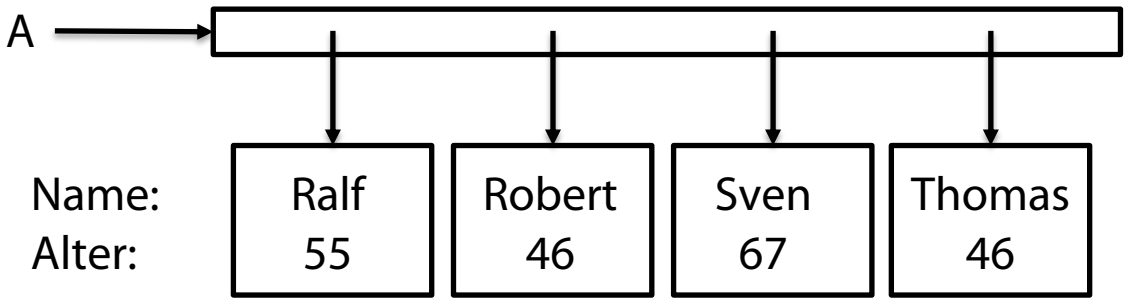
- Führe eine Agenda von Indexbereichen eines Feldes (am Anfang [1, n]), auf denen Partition arbeiten muss
- Solange noch Einträge auf der Agenda:
  - Nimm Indexbereich von der Agenda,
    wenn ein Element im Indexbereich partitioniere und
    setze zwei entsprechende Einträge auf die Agenda

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

Leslie Lamport, Thinking for Programmers
http://channel9.msdn.com/Events/Build/2014/3-642

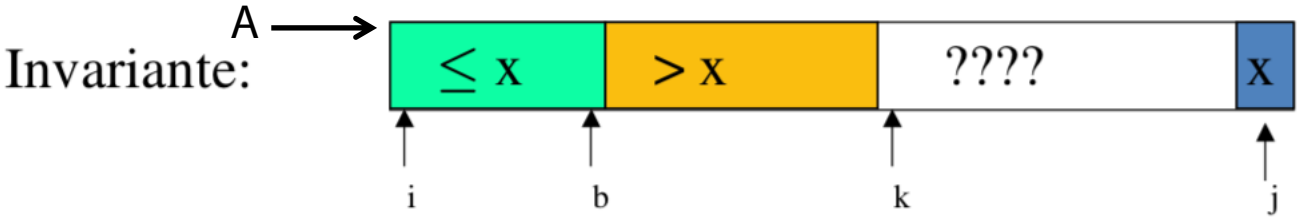
Stabilität eines Sortierverfahrens

- In den Feldern seien komplexe Objekte enthalten
- Sortierung nach vorgegebenem „Schlüssel“ (Name, Alter, …)



- Annahme: Sortierung nach Name sei gegeben
- Dann: Sortierung nach Alter
- Bei gleichem Sortierschlüsselwerte soll die Reihenfolge der Objekte bestehen bleiben (**Stabilität**)
  - Bei Sortierung nach Alter bleibt Robert vor Sven

Ist die Partitionierung von Quicksort stabil?




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

Charakterisierung von Sortierfunktionen

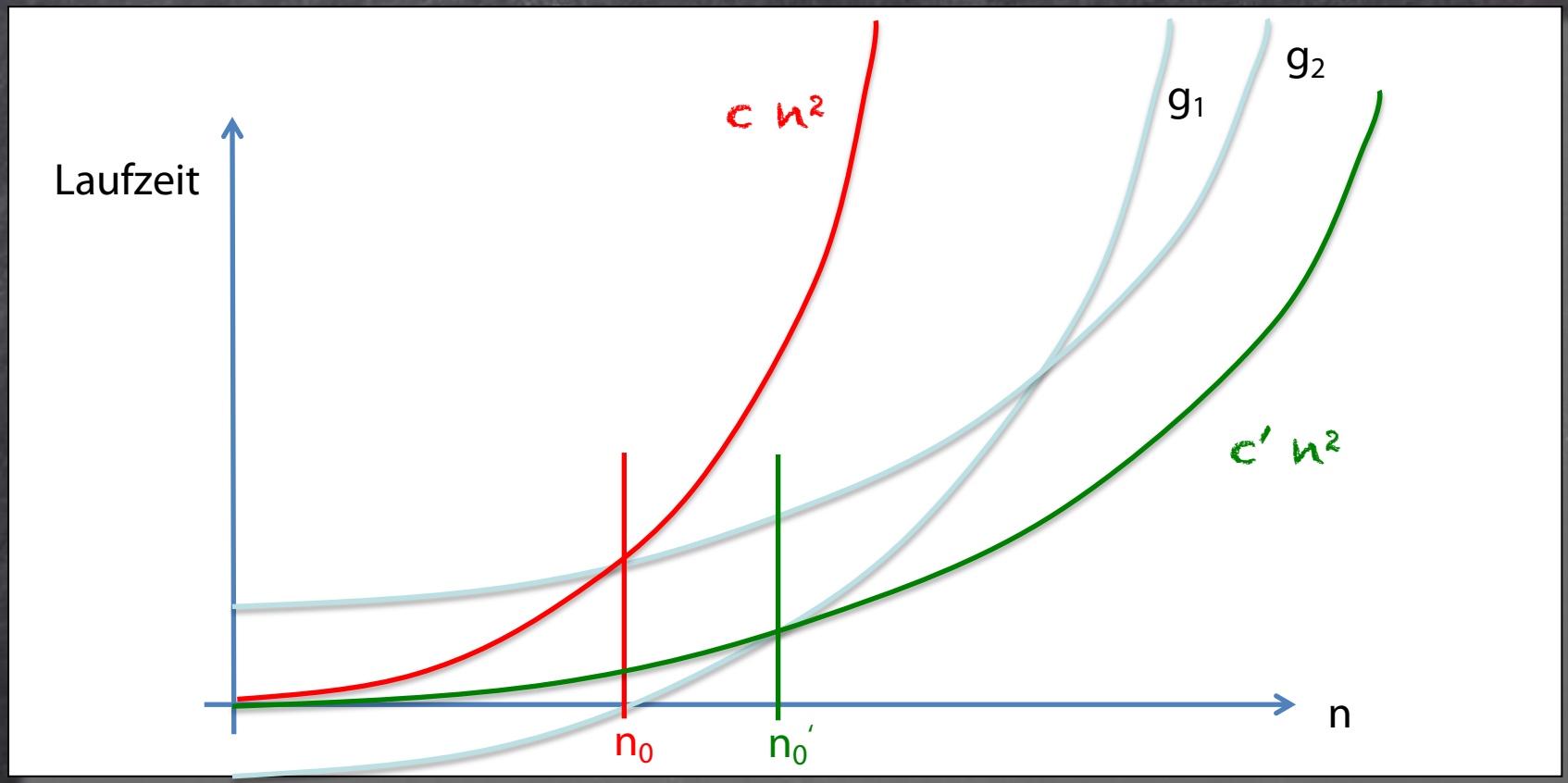
- Asymptotische Komplexität: O-Notation (oberer Deckel)
  - Relativ einfach zu bestimmen für Algorithmen basierend auf dem Verkleinerungsprinzip
  - Nicht ganz einfach für Algorithmen, die nach dem Teile-und-Herrsche-Prinzip arbeiten:

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

    - Substitutionsmethode
      (Ausrollen der Rekursion, Schema erkennen, ggf. Induktion)
    - Master-Methode (kommt später im Studium)
- Stabilität
  - Nicht offensichtlich und auch nicht immer gegeben

Noch einmal: Aufwandsbetrachtung

- 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

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

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 $g(n) = O(n^2)$

Zusammenfassung

- Quicksort
  - $T_{Quicksort}(n)$ im besten Fall in $O(n \log n)$
- Stabilität
- Ω-Notation: Mindestaufwand
  - $T_{Quicksort}(n)$ im besten Fall nicht in $\Omega(n^2)$