LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Skript zur Vorlesung:

Einführung in die Programmierung

WiSe 2009 / 2010

Skript © 2009 Christian Böhm, Peer Kröger, Arthur Zimek

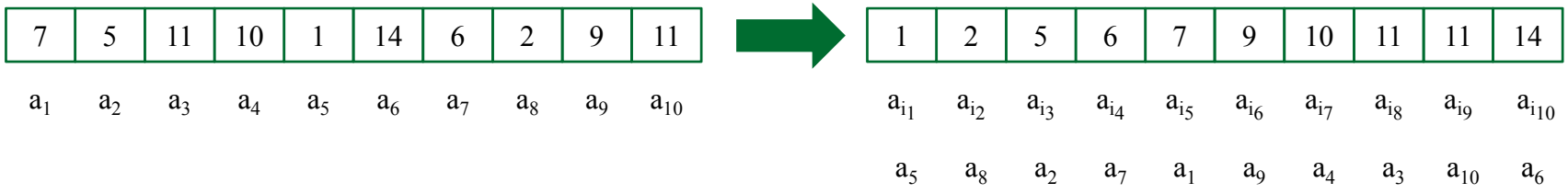
Prof. Dr. Christian Böhm
Annahita Oswald
Bianca Wackersreuther

Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Gliederung

Motivation

- Sortieren ist eine wichtige Operation auf Mengen von Objekten.
- Sortierte Mengen erlauben deutlich effizienteren Zugriff auf einzelne Elemente (z.B. durch binäre Suche).
- Problemstellung:
  - Gegeben: n Objekte $a_1, \ldots, a_n$ mit ihren Schlüsseln $k_1, \ldots, k_n$, anhand derer die Objekte sortiert werden sollen.
  - Gesucht: Eine Anordnung $a_{i_1}, \ldots, a_{i_n}$ mit
    - $(i_1, \ldots, i_n)$ ist eine Permutation von $(1, \ldots, n)$
    - $k_{i_1} \leq k_{i_2} \leq \ldots \leq k_{i_n}$ (*Sortierkriterium*).
- Beispiel:



Allgemeines

- Der Typ der Schlüssel k_i muss entweder primitiv sein, oder das Interface `Comparable` implementieren.
- Im folgenden betrachten wir folgende Vereinfachungen:
  - Wir verwenden die Schreibweisen $k_i < k_j$, $k_i > k_j$, usw., bzw. $k_i = k_j$ unabhängig davon, ob k_l einen primitiven Datentyp oder einen Objekttyp besitzt (im letzteren Fall müssten wir eigentlich mit den Methoden `compareTo` bzw. `equals` arbeiten).
  - Wir nehmen an, dass die Datenstruktur zur Verwaltung der Menge ein einfaches Array ist.
- Allgemeiner Sortieralgorithmus:
  while $\exists(i, j) : (i < j) \wedge (k_i > k_j)$ **do** vertausche a_i und a_j;
- Problem: Algorithmus nicht deterministisch.
- Daher: Speziellere Algorithmen nötig.

Stabilität von Sortierverfahren

Definition (*Stabilität* von Sortierverfahren)

Ein Sortierverfahren heißt *stabil*, wenn die relative Ordnung von Elementen mit gleichen Schlüsselwerten beim Sortieren erhalten bleibt, d.h. wenn für die sortierte Menge $k_{i_1}, \ldots k_{i_n}$ gilt:
$k_{i_j} = k_{i_l}$ und $j < l \Rightarrow i_j < i_l$.

Definition (*in situ* Sortierverfahren)

Ein Sortierverfahren heißt *in situ*, wenn zusätzlich zu dem zu sortierenden Array *kein* weiterer Speicherplatz benötigt wird. Dabei wird angenommen, dass die zu sortierenden Objekte die Indexpositionen $1, \ldots n$ belegen und das Feld mit Index 0 für Vertauschungen genutzt werden kann.

Klassifikation von Sortierverfahren

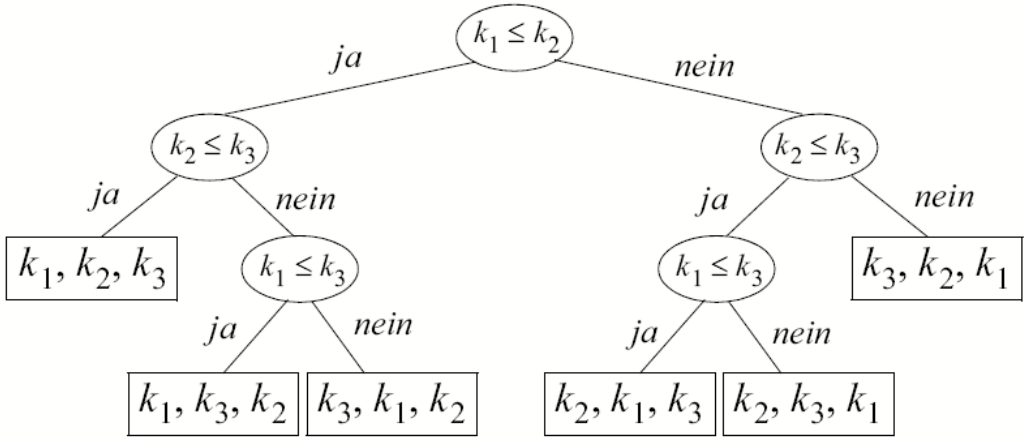
- Es gibt verschiedene Kriterien, um Sortieralgorithmen zu klassifizieren, z.B. ihre Stabilität oder ihren Speicherplatzbedarf.
- Das in dieser Vorlesung wesentliche Kriterium ist das Laufzeitverhalten.
- Um das Laufzeitverhalten unterschiedlicher Verfahren miteinander zu vergleichen, zählen wir die Anzahl der bei einer Sortierung von n Objekten durchzuführenden Operationen (in Abhängigkeit von n).

Laufzeitverhalten von Sortierverfahren

- Die beiden wesentlichen Operationen der meisten Sortierverfahren sind:
  - *Vergleiche* von Schlüsselwerten um Informationen über die vorliegende Ordnung zu erhalten. Im folgenden bezeichnen wir die Anzahl dieser Schlüssel-Vergleiche mit $C(n)$.
  - *Zuweisungsoperationen*, z.B. Vertauschungoperationen oder Tranpositionen von Objekten (i.a. innerhalb eines Arrays) zur Herstellung der Sortierordnung. Im folgenden bezeichnen wir die Anzahl dieser Zuweisungsoperationen mit $M(n)$.
- Wir beschränken uns hier auf Algorithmen, die nur Schlüsselvergleiche und Transpositionen verwenden.

Wie schnell kann man sortieren?

- Gesucht: Eine untere Schranke für die Anzahl $C_{max}(n)$ von Schlüsselvergleichen, die im schlechtesten Fall notwendig sind, um n Objekte zu sortieren.
- Entscheidungsbaum für 3 Schlüssel k_1, k_2 und k_3:



- In einem Entscheidungsbaum ohne redundante Vergleiche entspricht jedes Blatt einer der $n!$ verschiedenen Permutationen.
- Da der Entscheidungsbaum ein binärer Baum ist, hat er für n Elemente eine minimale Höhe von $\lceil \log(n!) \rceil$.

Wie schnell kann man sortieren?

- Damit gilt für einen beliebigen Sortieralgorithmus:

$$C_{max}(n) \geq O(n \cdot \log n), \text{ denn}$$

$$C_{max}(n) \geq \lceil \log(n!) \rceil \quad \text{und}$$

$$n! \geq n \cdot (n-1) \cdot \ldots \cdot (\lceil n/2 \rceil) \geq (\frac{n}{2})^{\frac{n}{2}}$$

$$\Rightarrow \log_2(n!) \geq \frac{n}{2} \cdot \log_2(\frac{n}{2}) = O(n \cdot \log n).$$

- Resultat: Sortierverfahren haben mindestens eine Laufzeit von $O(n \cdot \log n)$.

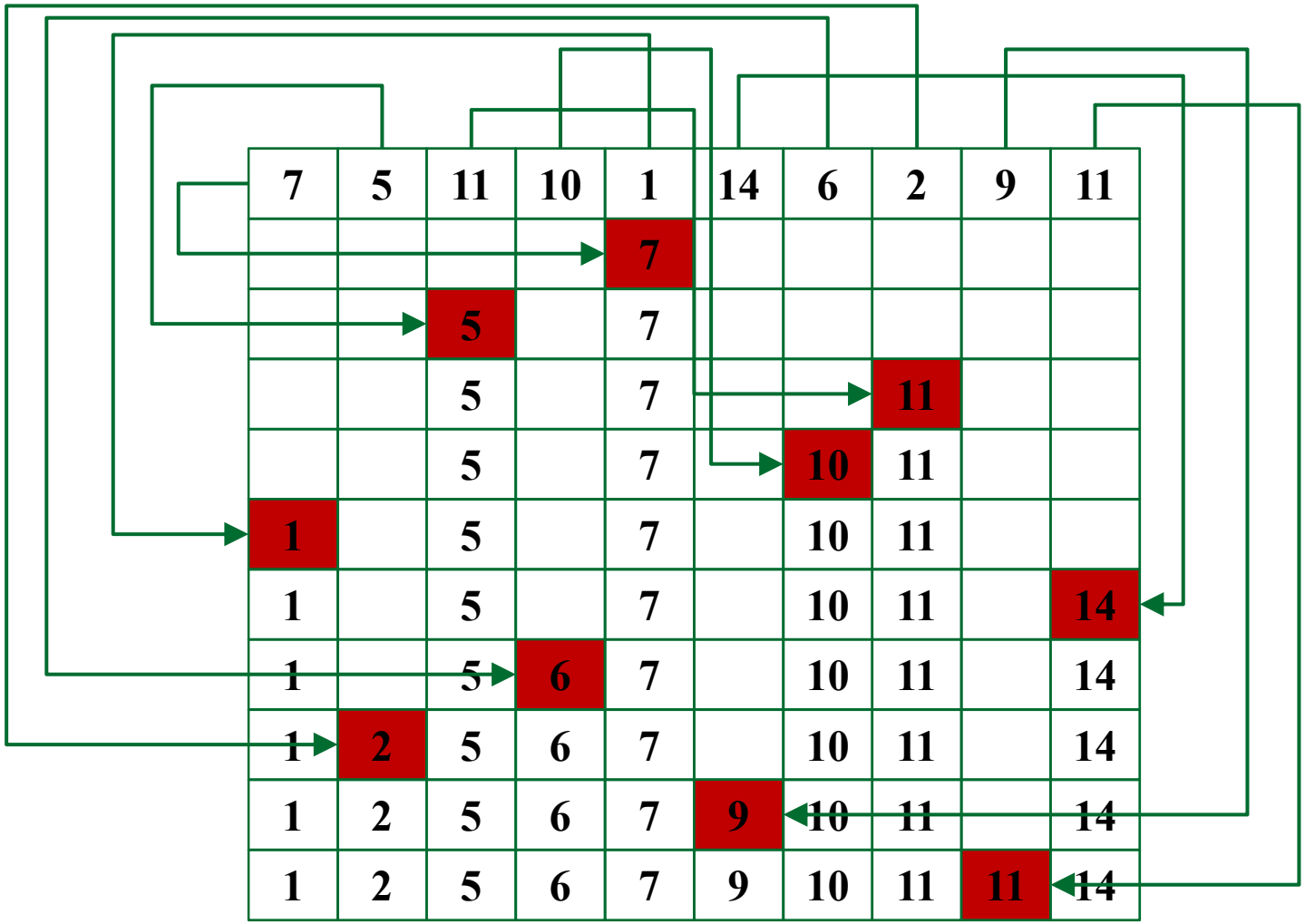
Überblick

- Einfache Sortierverfahren besitzen meist eine Laufzeit von $O(n^2)$.
- Für wenige Objekte (kleineWerte von n) ist dies meist noch akzeptabel, für große Mengen (z.B. $n > 100$) allerdings nicht mehr.
- Im folgenden besprechen wir einige einfache Sortierverfahren informell. Diese Algorithmen wurden in den Übungen für den primitiven Datentyp `int` implementiert. Nun sollte es Ihnen nicht mehr allzu schwer fallen, diese Algorithmen für beliebige Typen in einem Java-Programm zu implementieren.

Sortieren durch Abzählen: *Count Sort*

- Prinzip:
  Der j-te Schlüssel der sortierten Folge ist größer als $j-1$ der übrigen Schlüssel. Die Position eines Schlüssels in der sortierten Folge kann damit durch Abzählen der kleineren Schlüssel ermittelt werden.
- Vorgehen:
  Für jedes Objekt o im ursprünglichen Array: Zähle die Objekte o mit $o' < o$.

Beispiel: *Count Sort*



Diskussion: *Count Sort*

- Laufzeit:

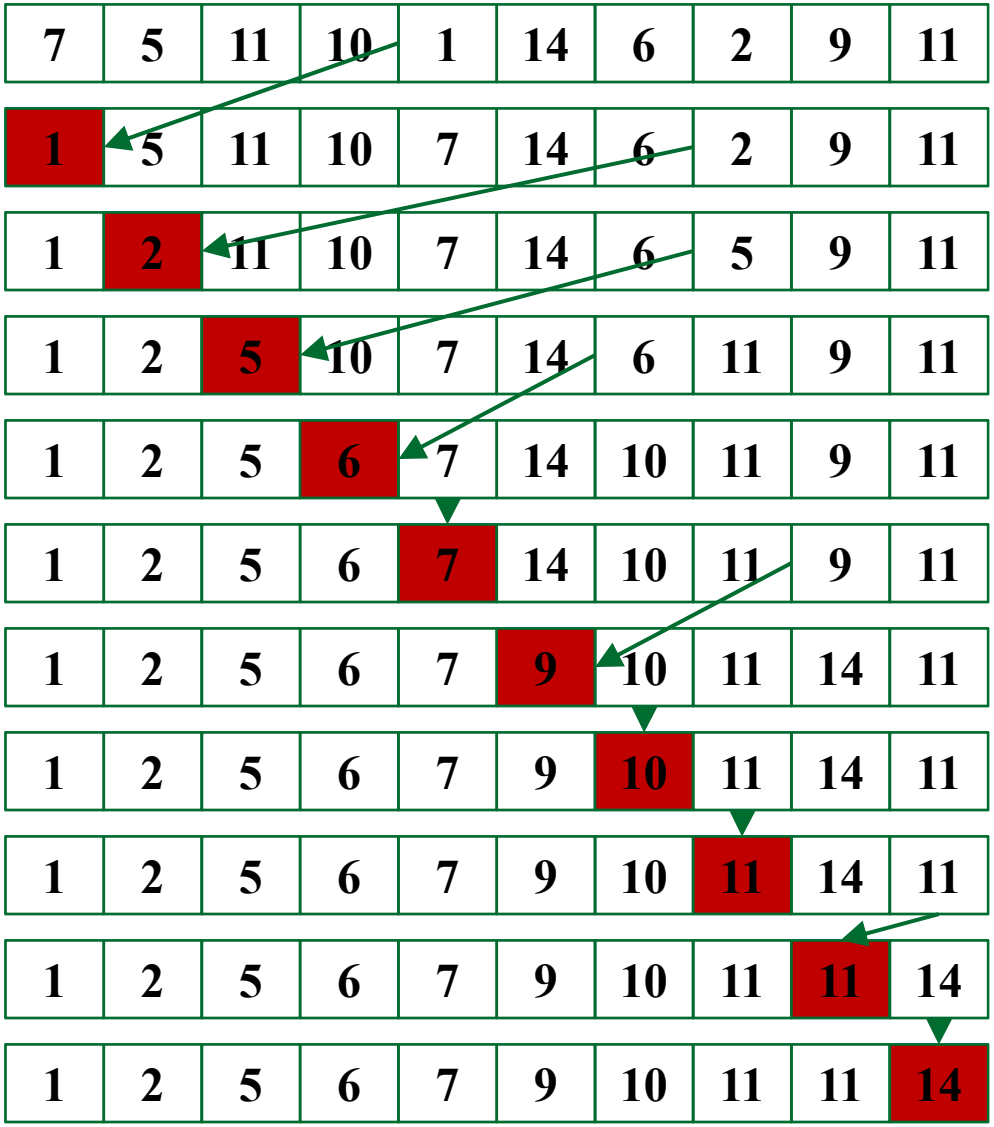
$$C(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n \cdot (n-1)}{2} = O(n^2)$$
$$M(n) = C(n) = O(n^2)$$

- Das Verfahren ist nicht in situ.
- Das Verfahren ist stabil (wird instabil falls Test $o' \leq o$).

Sortieren durch direktes Auswählen: *Selection Sort*

- Prinzip / Vorgehen:
  Suche aus allen *n* Elementen das kleinste und setze es an die erste Stelle. Wiederhole dieses Verfahren für die verbleibenden Elemente. Nach *n*-1 Durchläufen ist die Folge sortiert.

Beispiel: *Selection Sort*



Diskussion: *Selection Sort*

- Laufzeit:

$$C(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n \cdot (n-1)}{2} = O(n^2)$$
$$M_{\emptyset}(n) = O(n \log n) \text{ (ohne Beweis)}$$

- Das Verfahren ist in situ.
- Je nach Implementierung ist das Verfahren stabil.

Sortieren durch direktes Austauschen: *Bubble Sort*

- Prinzip:
  Vertausche die relative Reihenfolge benachbarter Elemente, sodass kleine Elemente nach vorne und größere Elemente nach hinten wandern.
- Vorgehen:
  Im ersten Durchlauf werden die Paare (a_{n-1}, a_n), (a_{n-2}, a_{n-1}), … bearbeitet. Dadurch wandert das kleinste Element an die erste Position im Array. Nach n-1 Durchläufen ist die Sortierung abgeschlossen.

Beispiel: *Bubble Sort*

7	5	11	10	1	14	6	2	9	11
1	7	5	11	10	2	14	6	9	11
1	**2**	7	5	11	10	6	14	9	11
1	2	**5**	7	6	11	10	9	14	11
1	2	5	**6**	7	9	11	10	11	14
1	2	5	6	**7**	9	10	11	11	14
1	2	5	6	7	**9**	10	11	11	14
1	2	5	6	7	9	**10**	11	11	14
1	2	5	6	7	9	10	**11**	11	14
1	2	5	6	7	9	10	11	**11**	14
1	2	5	6	7	9	10	11	11	**14**

Diskussion: *Bubble Sort*

- Laufzeit:

$$C(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n \cdot (n-1)}{2} = O(n^2)$$
$$M_{\emptyset}(n) = O(n^2) \text{ (ohne Beweis)}$$

- Das Verfahren ist in situ.
- Je nach Implementierung ist das Verfahren stabil.

Das Divide-and-Conquer-Prinzip

- Analyse:
  Einfache Sortierverfahren reduzieren die Größe des noch zu sortierenden Arrays in jedem Schritt lediglich um eins.
- Idee:
  Reduziere das noch zu sortierende Array in jedem Schritt um die Hälfte. Dadurch wird eine Laufzeit von $O(n \log n)$ ermöglicht.
- Das allgemeine Algorithmus-Prinzip von Quicksort heißt *Divide-and-Conquer* ("teile-und-beherrsche").
- Divide-and-Conquer-Algorithmen zerlegen ein gegebenes Problem solange in kleinere Probleme, bis diese beherrschbar sind. Die globale Lösung ergibt sich durch Verschmelzen der einzelnen (Teil-)Lösungen.

Divide-and-Conquer-Sortieralgorithmen

- Allgemeines Schema eines Divide-and-Conquer-Sortieralgorithmus:
  - Wenn die Objektmenge klein genug ist, löse das Problem direkt.
  - Ansonsten:
    DIVIDE: Zerlege die Menge in möglichst gleich große Teile.
    CONQUER: Löse das Problem für jede Teilmenge.
    MERGE: Berechne aus den Teil- die Gesamt-Lösung.
- Wichtige Eigenschaft: Jedes Divide-and-Conquer-Sortierverfahren, dessen Divide- und Merge-Schritt in $O(n)$ Zeit durchgeführt werden können und das eine balancierte Unterteilung des Problems garantiert, besitzt eine Laufzeit von $O(n \log n)$.

Prinzip: *Quicksort*

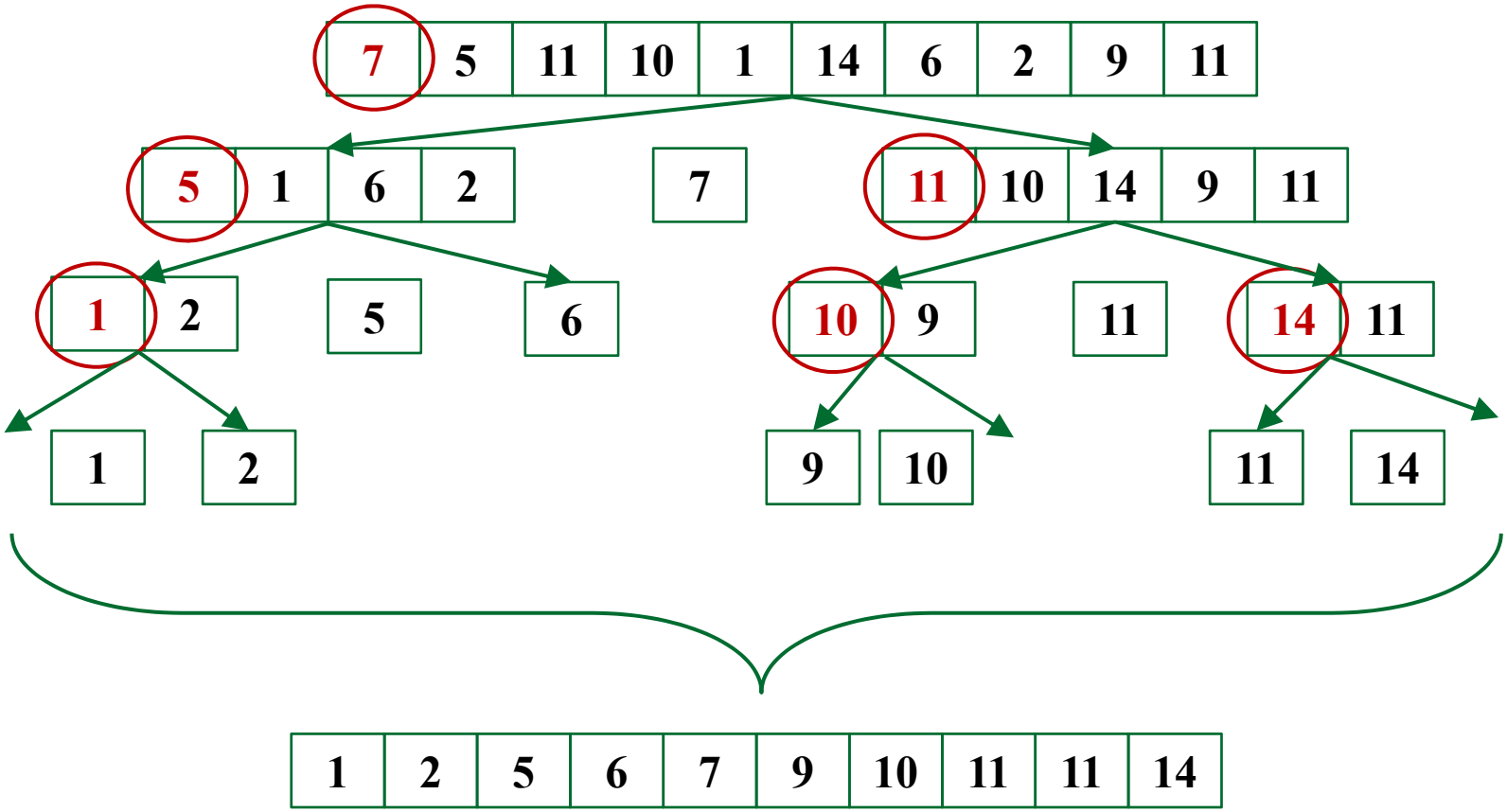
- Ein (prinzipiell beliebiger) Schlüssel x aus dem Array wird ausgewählt (das sog. *Pivot-Element*).
- Divide-Schritt: Das Array wird in Schlüssel $\geq x$ und Schlüssel $< x$ zerlegt.
- Conquer-Schritt: Die beiden resultierenden Teilarrays werden *rekursiv* bis auf Elementebene in gleicher Weise behandelt.
- Merge-Schritt: Entfällt, durch entsprechende Speicherung der Teilarrays innerhalb des Arrays.

Algorithmus: *Quicksort*

```
ALGORITHM QuickSort(ARRAY S)
  IF S.length = 1
  THEN RETURN S;
  ELSE
    // DIVIDE:
    Waehle ein Pivot-Element x aus S;
    Initialisiere zwei Teilfolgen S1 und S2;
    FOR EACH y IN S DO
      IF y < x
      THEN fuege x zu S1 hinzu;
      ELSE fuege x zu S2 hinzu;
      ENDIF
    ENDFOR
    // CONQUER:
    S1' = QuickSort(S1);
    S2' = QuickSort(S2);
    // MERGE:
    RETURN Konkatenation aus S1' und S2';
  ENDIF
END
```

- Arbeit im Divide-Schritt: $O(n)$.
- Arbeit im Merge-Schritt: $O(1)$.

Beispiel: *Quicksort*



Laufzeit: *Quicksort*

- Im besten Fall:
  In jedem Divide-Schritt wird immer in zwei gleich große Teilarrays geteilt: $O(n \log n)$.
- Im Durchschnitt: $O(n \log n)$.
- Im schlechtesten Fall:
  Degenerierung zur linearen Liste, d.h. es wird stets das größte / kleinste Element als Pivot gewählt: $O(n^2)$.

Mehr dazu in der “Algorithmen und Datenstrukturen”.

Schlussbemerkungen: *Quicksort*

- Austausch von Schlüsseln über große Distanzen ⇒ Array ist schnell “nahezu” sortiert.
- Wie alle komplexen Sortierverfahren ist Quicksort schlecht für kleine n.
- Es gibt verschiedene Methoden, die Wahrscheinlichkeit des schlechtesten Falls zu vermindern; dennoch bleibt Quicksort immer eine Art “Glückspiel”.
- Quicksort war lange Zeit das im Durchschnittsverhalten beste bekannte Sortierverfahren.
- Die statischen Methoden `java.util.Arrays.sort` implementieren Quicksort.