LMU LUDWIG-MAXIMILIANS-UNIVERSITY MUNICH
DEPARTMENT INSTITUTE FOR INFORMATICS
DATABASE SYSTEMS GROUP

Algorithmen und Datenstrukturen
Kapitel 3: Sortierverfahren

Skript zur Vorlesung

Algorithmen und Datenstrukturen

Sommersemester 2015

Ludwig-Maximilians-Universität München

basierend auf dem Skript von Prof. Dr. Martin Ester, Prof. Dr. Daniel A. Keim, Dr. Michael Schiwietz und Prof. Dr. Thomas Seidl

Motivation fürs Sortieren

Sortieren: Anordnen einer gegebenen Menge von Objekten in einer bestimmten Ordnung

- Sortierte Folgen von Objekten → deutliche **Vereinfachung für den Zugriff** auf einzelne Elemente der Objektmenge
- Sortieren ist daher eine grundlegende Tätigkeit in vielen Anwendungsbereichen angewandt wird, z.B.:
- .

Telefonbücher / Wörterbücher / Büchereikataloge	Sortierung nach Buchstaben
Zugfahrpläne / Flugpläne / Terminkalender	Sortierung nach Datum/Uhrzeit
Kundenlisten / Inventarlisten	Sortierung nach Schlüsseln
LRU- (least recently used) Schlange	Sortierung nach Zeitpunkt der letzten Nutzung

Problemstellung

Allgemeine Problemstellung:

Gegeben:

n Objekte $a_1, a_2, \ldots, a_n$ mit ihren Sortierschlüsseln $k_1, k_2, \ldots, k_n$

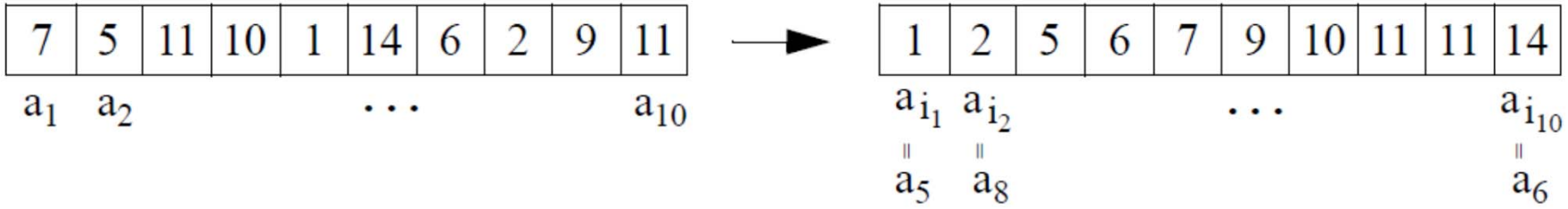
Gesucht:

Eine Anordnung $a_{i_1}, a_{i_2} \ldots, a_{i_n}$ mit:

- $(i_1, i_2, \ldots, i_n)$ ist eine Permutation von $(1,2, \ldots, n)$ und
- $k_{i_1} \leq k_{i_2} \leq \cdots \leq k_{i_n}$ (Sonderkriterium)

Beispiel

Beispiel:



Anmerkung:

- Sortierschlüssel müssen einem Datentyp angehören, auf den das Sortierkriterium anwendbar ist, d.h. auf dem eine „<“-Relation definiert ist.
- Gilt in natürlicher Weise für atomare Datentypen (wie **int**, **float** oder **char**)

Beispiel: Allgemeine Klassendeklaration

Sei die folgende allgemeine Klassendeklaration vorgegeben:

```
class Entry {
   public int key;
   …
}
```

```
class SortableArray {
   public Entry a[];
   // Im Konstruktor initialisieren mit new Entry[n]
   // Die Schlüssel sind in a[0].key bis a[n-1].key gespeichert
   protected int n = a.length;
   protected void vertausche(int k, int l) {
      Entry t = a[k]; a[k] = a[l]; a[l] = t;
   }
}
```

Allgemeiner Sortieralgorithmus

while

$\exists(i,j) : (i < j) \wedge \left(k_i > k_j\right)$

do

vertausche a_i und a_j

end

Anmerkungen:

- Nicht deterministisch
- Geeignete Kontrollstruktur (Algorithmus) nötig!

Zwei Kategorien von Sortierverfahren

- Allgemeine Aufteilung in zwei Kategorien:
  - Sortieren von Arrays (**internes Sortieren**)
  - Sortieren sequentieller Files (**externes Sortieren**)

- Externe Sortierverfahren kommen für sehr große Datenmengen zur Anwendung, die nicht komplett in den Hauptspeicher eingelesen werden können.

internes Sortieren: alle Objekte sind bekannt

externes Sortieren:

zu jedem Zeitpunkt ist nur ein Teil der Objekte bekannt

Stabilität von Sortierverfahren

Definition:
Ein Sortierverfahren heißt **stabil**, wenn die relative Ordnung von Elementen mit gleichen Schlüsselwerten beim Sortieren erhalten bleibt, d.h. wenn für die sortierte Folge $k_{i_1}, k_{i_2}, \ldots, k_{i_n}$ gilt: $k_{i_j} = k_{i_l}$ und $j < l \Rightarrow i_j < i_l$

Weitere Klassifikationen von Sortierverfahren sind möglich:

- Über ihr **methodisches Vorgehen** (Sortieren durch Einfügen oder durch Auswählen; Divide-and-Conquer-Verfahren; …),
- Über ihr **Laufzeitverhalten** (im schlechtesten Fall / Durchschnitt),
- Über ihren **Speicherplatzbedarf** (*in situ* oder zweites Array / zusätzlicher Speicher nötig)

- Für uns wesentliches Kriterium → **Laufzeitverhalten**

Vergleich verschiedener Verfahren

- **Vergleich unterschiedlicher Verfahren:**
  Zählen der Anzahl der bei einer Sortierung von n Elementen durchzuführenden Operationen (in Abhängigkeit von n).
- Die beiden **wesentlichen Operationen** der internen Sortierverfahren sind:
  - **Vergleiche von Schlüsselwerten**
    → Informationen über die vorliegende Ordnung
  - **Zuweisungsoperationen**, z.B. Vertauschungsoperationen oder Transpositionen von Objekten (im allg. innerhalb eines Arrays)
    → Herstellen der Ordnung

Im Folgenden bezeichne

- $C(n)$ die Anzahl der Vergleiche
- $M(n)$ die Anzahl der Zuweisungsoperationen,

die bei der Sortierung von n Elementen notwendig sind.

3.1. Interne Sortierverfahren

Sortiergeschwindigkeit

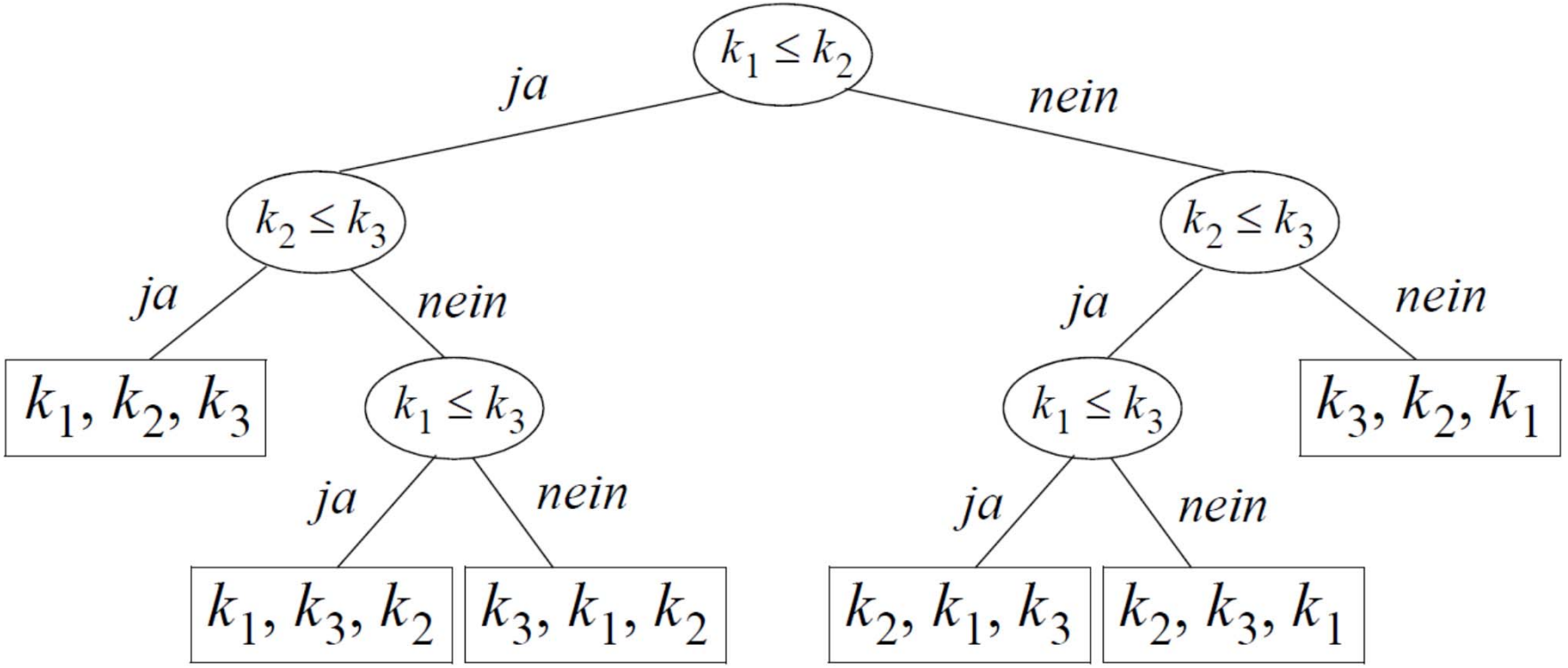
Frage:

Wie schnell kann man überhaupt sortieren?

- Beschränkung auf solche Algorithmen, die nur Schlüsselvergleiche und Transpositionen verwenden
- **Gesucht:**
  Eine untere Schranke für die Anzahl $C_{max}(n)$ von Schlüsselvergleichen, die im schlechtesten Fall notwendig sind.

Entscheidungsbaum I

Entscheidungsbaum für 3 Schlüssel k_1, k_2, k_3:



Entscheidungsbaum II

- In einem Entscheidungsbaum ohne redundante Vergleiche entspricht jedes Blatt einer der $n!$ verschiedenen Permutationen von n Elementen.
  → Entscheidungsbaum für n Elemente hat eine **minimale Höhe** von
  $$\lceil \log_2(n!) \rceil ,$$
  da es sich um einen binären Baum handelt.

- Damit gilt für einen beliebigen Sortieralgorithmus:
  $$C_{max}(n) = \Omega(n \cdot \log n)$$

Entscheidungsbaum III

Begründung:

$$C_{max}(n) \geq \lceil \log_2(n!) \rceil$$

und

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

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

$$\rightarrow C_{max}(n) \geq O(n \cdot \log n) \Leftrightarrow C_{max}(n) = \Omega(n \cdot \log n)$$

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

Einfache Sortieralgorithmen

Anmerkungen:

- Algorithmen, die diese Laufzeit auch tatsächlich erreichen, sind recht komplex.
- Deshalb gehen wir zunächst nur auf **einfache Sortierverfahren** ein.
- Für ein kleines n (z.B. $n < 100$) erreichen einfache Sortierverfahren meist eine ausreichende Performance!
- Einfache Sortierverfahren besitzen meist eine Laufzeit von $O(n^2)$.

Sortieren durch Abzählen I

Sortieren durch Abzählen:

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.

```
int zaehler[] = new int[n];
// Die Schlüssel sind in a[0].key bis a[n-1].key gespeichert.
```

Sortieren durch Abzählen II

Ziel:

zaehler[i] = (Anzahl der Elemente a_j mit $a_j.key < a_i.key$)

Start:

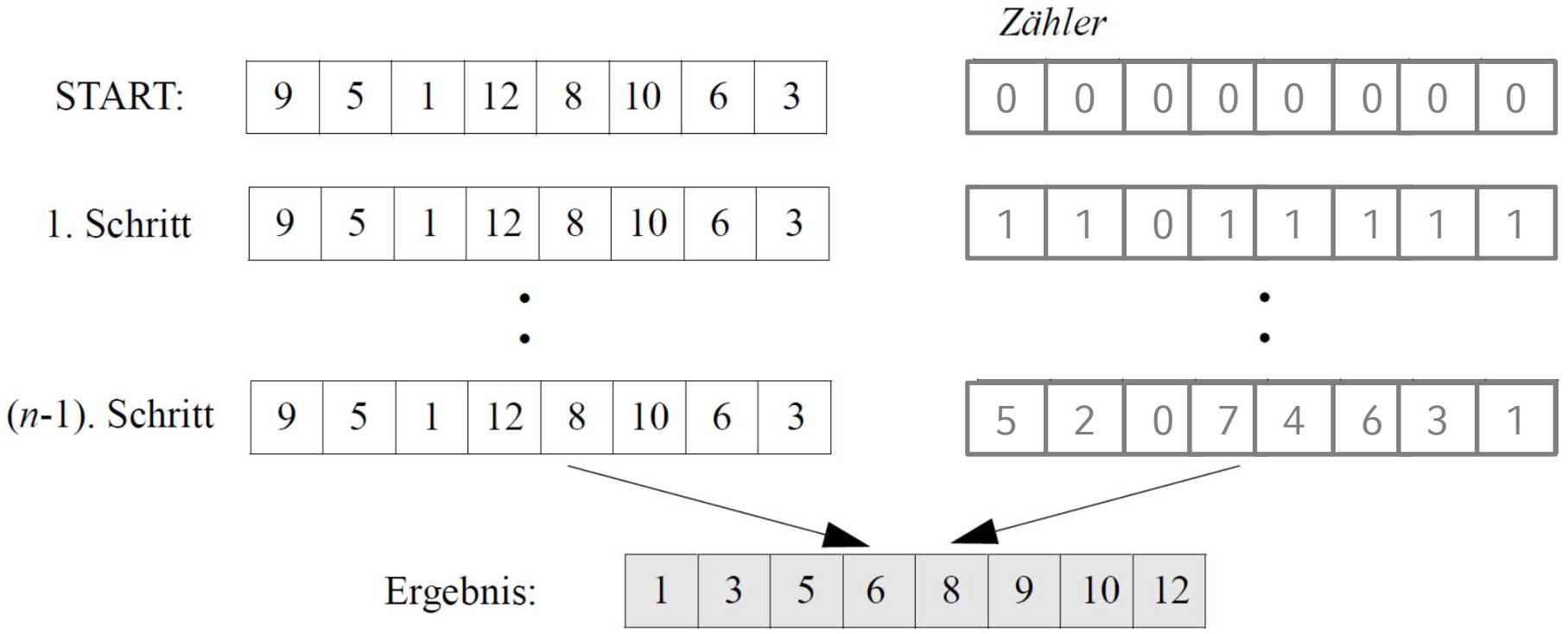
zaehler[] = {0, 0, …, 0}

Programmstück:

```
for (int i = n-1; i >= 1; i--) {
   for (int j = i – 1; j >= 0; j--) {
      if (a[i].key < a[j].key)
         zaehler[j]++;
      else
         zaehler[i]++;
   }
}
```

Sortieren durch Abzählen III

Veranschaulichung:



Sortieren durch Abzählen IV

- Algorithmus ist stabil.
- Algorithmus wird instabil, falls ‚<' mit ‚≤' ersetzt wird.

- Für die benötigte Anzahl von Vergleichen gilt:

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

- Die Anzahl der Zuweisungsoperationen für den Zähler ist identisch zur Anzahl der Schlüsselvergleiche. Hinzu kommt ein linearer Aufwand, da jedes Objekt jetzt noch an die richtige Stelle platziert werden muss.

$$\rightarrow M(n) = O(n^2)$$

Selection Sort I

Sortieren durch direktes Auswählen (Selection Sort):

Prinzip:

- Suche aus allen n Elementen das kleinste und setze es an die erste Stelle.
- Wiederhole dieses Verfahren für die verbleibenden $n - 1, n - 2, \ldots$ Elemente.
- Nach $n - 1$ Durchläufen ist die Sortierung abgeschlossen.

Selection Sort II

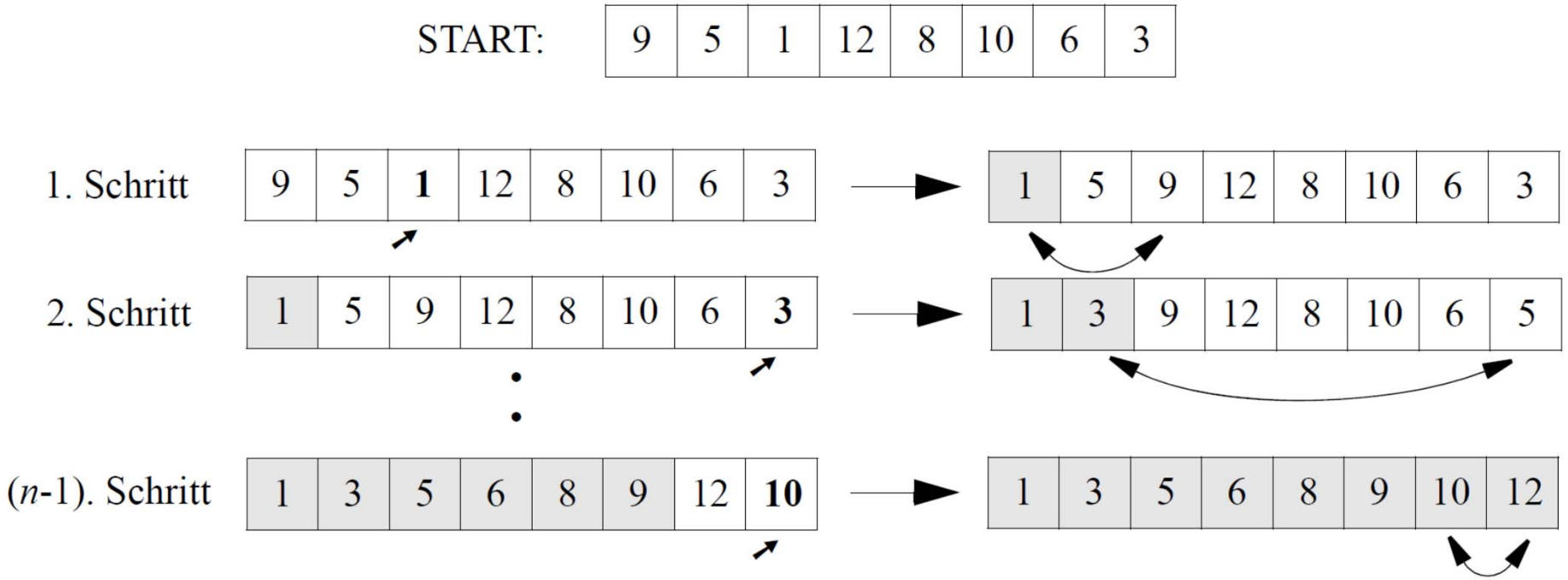
Algorithmus:

```
void selectionSort() {
   int minindex;
   for (int i = 0; i <= n-2; i++) {
      minindex = i;

      // minimales Element suchen
      for (int j = i+1; j < n; j++) {
         if (a[j].key < a[minidex].key) {
            minindex = j;
         }
      }
      vertausche(i, minindex);
   }
}
```

Selection Sort III

Veranschaulichung:



Selection Sort IV

- **Anzahl der Vergleichoperationen**:

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

- **Durchschnittliche Anzahl von Zuweisungen**: *(ohne Beweis)*

$$M_{\emptyset}(n) = O(n \cdot \log n)$$

(Anzahl von Vertauschungen: $O(n)$)

- Wesentlicher Aufwand
  → Bestimmung des Minimums der verbleibenden Elemente

- **Lemma:**
  Jeder Algorithmus zum Auffinden des Minimums von n Elementen, der auf Vergleichen von Paaren von Elementen basiert, benötigt mindestens $n - 1$ Vergleiche.

Bubble Sort I

Sortieren durch direktes Austauschen (Bubble Sort):

Prinzip: Das Verfahren basiert darauf, die relative Reihenfolge benachbarter Elemente zu überprüfen und diese so zu vertauschen, dass kleine Elemente an den Anfang und größere Elemente an das Ende des Arrays wandern.

- 1. Durchlauf: Bearbeitung der Paare $(a_{n-1}, a_n), (a_{n-2}, a_{n-1})$
- Das kleinste Element wandert an die an die erste Position im Array; es wird nicht weiter betrachtet.
- So entsteht eine immer längere sortierte Folge von Elementen am Anfang des Arrays
- Sortierung ist abgeschlossen nach $n - 1$ Durchläufen.

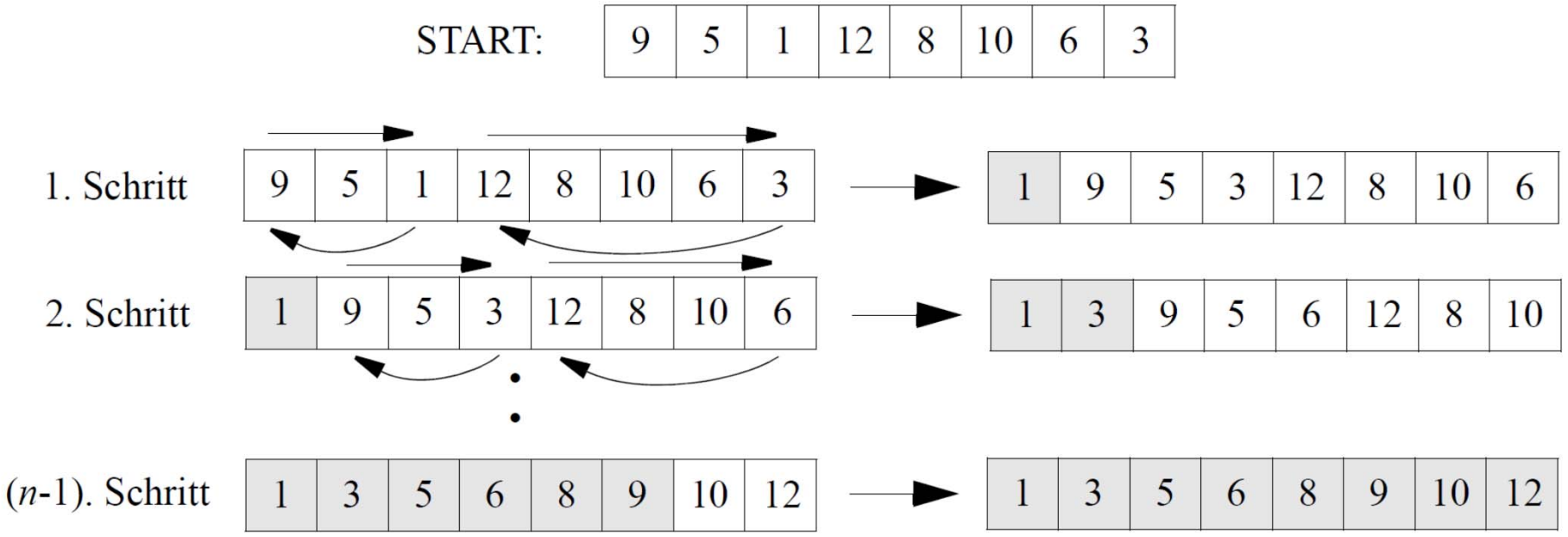
Bubble Sort II

Algorithmus:

```
void bubbleSort() {
   for (int i = 0; i < n-1; i++) {
      for (int j = n-1; j >= i; j--) {
         if (a[j].key < a[j-1].key) {
            vertausche(j, j-1);
         }
      }
   }
}
```

Bubble Sort III

Veranschaulichung:



Bubble Sort IV

- **Anzahl der Vergleichoperationen**:

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

- **Durchschnittliche Anzahl von Vertauschungen**:

$$M_{\emptyset}(n) = \frac{n \cdot (n-1)}{4} = O(n^2)$$

- **Verbesserungen:**
  - Durchlauf der inneren for-Schleife ohne jegliche Vertauschung zweier Elemente ermöglicht frühzeitigen Abbruch des Algorithmus
  - Beobachtung: Asymmetrisches Verhalten!
    - kleine Schlüssel springen schnell an den Anfang des Array
    - große Schlüssel wandern nur schrittweise ans Ende

    → Änderungen der Richtung aufeinanderfolgender Durchläufe („Shaker Sort“)

→ Verbesserungen sparen nur Schlüsselvergleiche, keine Vertauschungsoperationen!

Insertion Sort I

Sortieren durch direktes Einfügen (Insertion Sort):

Prinzip:

- Element a_1 beschreibt für sich genommen stets eine korrekt sortierte ‚Folge'.
- Seien nun für $j = 2,3, \ldots$ die ersten $j - 1$ Elemente $a_1, a_2, \ldots, a_{j-1}$ korrekt sortiert.
- Im j-ten Durchlauf wird das Element a_j an die richtige Stelle der vorsortierten Folge eingefügt.
- Nach $n - 1$ Durchläufen ist die Sortierung abgeschlossen.

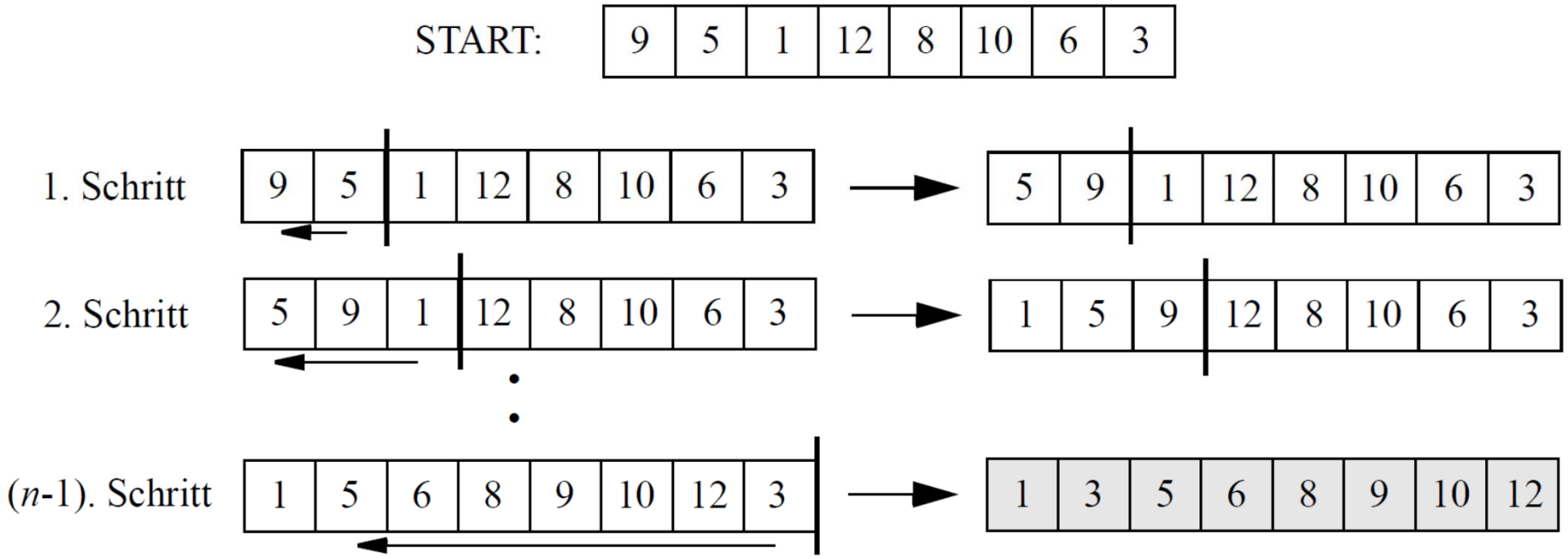
Insertion Sort II

Algorithmus:

```
void insertionSort() {
   Entry x = new Entry;
   for (int j = 1; j < n; j++) {
      x = a[j];
      i = j-1;
      while ((i >= 0) && (x.key < a[i].key)) {
         a[i+1] = a[i]; // Transposition der Array-Elemente
         i = i-1;
      }
      a[i+1] = x;
   }
}
```

Insertion Sort III

Veranschaulichung:



Insertion Sort IV

Laufzeitanalyse:

- Äußere (for-) Schleife wird stets $n-1$ mal durchlaufen
- Beim Einfügen des j-ten Elementes ($(j-1)$-ter Durchlauf der for-Schleife) werden benötigt:
  - Worst Case: $j-1$ Schlüsselvergleiche und j Transpositionen
  - Durchschnitt: $\frac{j-1}{2}$ Schlüsselvergleiche und $\frac{j+1}{2}$ Transpositionen

$$\rightarrow C_{\emptyset}(n) = \sum_{j=2}^{n} \frac{j-1}{2} = \frac{1}{2}\sum_{j=2}^{n} j - \frac{n-1}{2} = \frac{n^2+2-2}{4} - \frac{2n-2}{4} = \frac{1}{4}(n^2-n) = O(n^2)$$

$$C_{worst}(n) = \sum_{j=2}^{n} j - 1 = \sum_{j=1}^{n-1} j = \frac{1}{2} \cdot (n^2-n) = O(n^2)$$

Und ebenfalls: $O(n^2)$ Transpositionen.

Insertion Sort V

Verbesserung:

- Beim Einfügen: Binäres Suchen statt while-Scheife
- Dadurch: Anzahl der Schlüsselvergleiche: $O(n \cdot \log n)$ (sowohl im Worst Case als auch im Durchschnitt)
- Aber: Anzahl der Transpositionen bleibt erhalten → Aufwand $O(n^2)$ bleibt

Insertion Sort VI

Bemerkung:

Ein Sortierverfahren, das Objekte immer nur um eine Position verschiebt, hat eine durchschnittliche Laufzeit von mindestens $O(n^2)$.

Begründung (Lemma):

Sei $k_1, k_2, \ldots, k_n$ eine zufällige Permutation von $\{1,2,\ldots,n\}$. Dann gilt für die durchschnittliche Anzahl $M_\emptyset(n)$ der Stellen, über die die n Objekte bewegt werden:

$$M_\emptyset(n) > \Theta(n^2)$$

Folgerung:

Sortierverfahren, die im Durchschnitt schneller als $O(n^2)$ sind, müssen Transpositionen in Sprüngen statt nur stellenweise vornehmen (wie z.B. beim Selection Sort)

Shell Sort I

Verfeinerung des Insertion Sort (Shell Sort):
(Shell 1959, „*diminishing increment sort*“)

Prinzip:

- Sei $n = 2^k, k \in \mathbb{N}$.
- Das Array wird in jedem Schritt als Menge kleinerer Gruppen von Elementen betrachtet, beschrieben durch eine vorgegebene Schrittweite.
- Diese werden getrennt voneinander (→ schneller) über Insertion Sort sortiert.
- Beispiele für Schrittweitenfolge: $\frac{n}{2}, \frac{n}{4}, \ldots, 1$

Shell Sort II

Prinzip: (Fortsetzung)

- 1. Schritt: Bildung von $\frac{n}{2}$ Gruppen mit jeweils 2 Elementen
- 2. Schritt: Bildung von $\frac{n}{4}$ Gruppen mit jeweils 4 Elementen
- … (jeder Schritt nutzt die bereits bestehende Vorsortierung aus)

- Nach $\log(n)$ Schritten: Gesamte Folge ist sortiert.
- Für allgemeines n ergeben sich u.U. um eins größere Gruppen

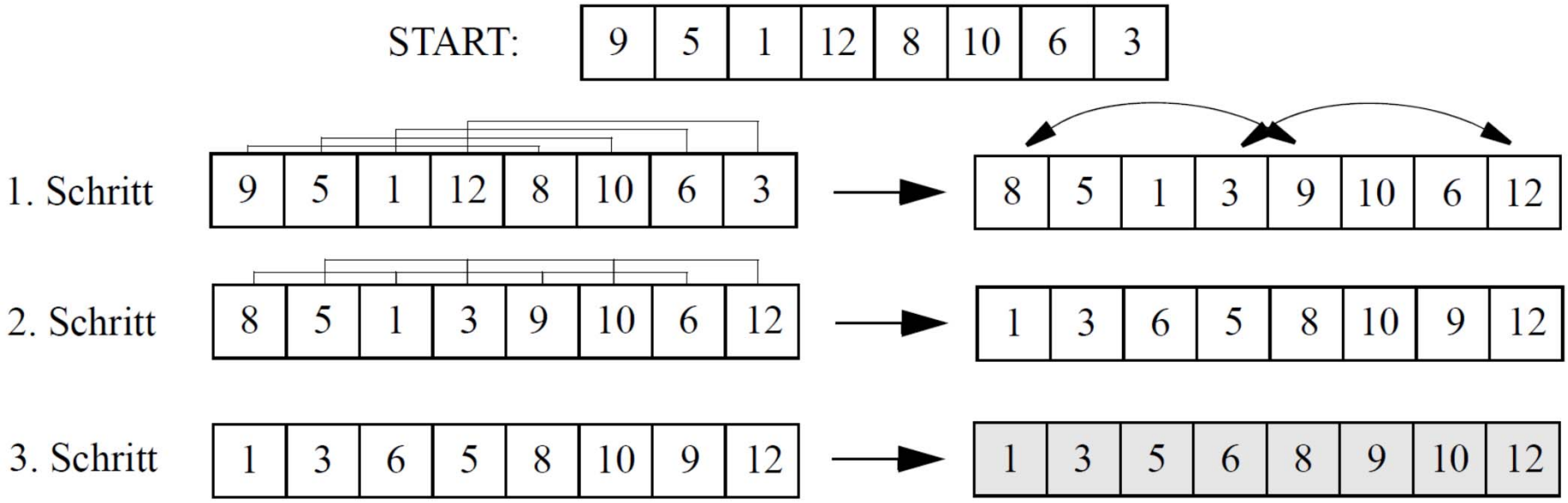
Shell Sort III

Algorithmus: ($\forall n \in \mathbb{N}$ korrekt)

```
void shellSort() {
   int incr, j;
   incr = n/2; // Distanz der Elemente eines Teilarrays
   while (incr > 0) {
      for (int i = incr; i < n; i++) { // einzelne Teilarrays betr.
         j = i - incr;
         while (j >= 0) {
            if (a[j].key > a[j+incr].key) {
               vertausche(j, j+incr);
               if (j >= incr) j = j - incr;
                  else break;
            }
            else break;
         }
      }
      incr = incr / 2;
   }
}
```

Shell Sort IV

Veranschaulichung:



Shell Sort V

Laufzeitanalyse:

- 1. Schritt:
  Für $\frac{n}{2}$ Teilarrays höchstens 1 Vertauschung
- 2. Schritt:
  Für $\frac{n}{4}$ Teilarrays höchstens $1 + 2$ Vertauschungen
- Im $\log n = k$-ten Schritt:
  Höchstens $1 + 2 + \cdots + \frac{n}{2}$ Vertauschungen

→ Gesamtlaufzeit im Mittel nahe bei $O(n \cdot \sqrt{n}) = O(n^{1,5})$
Laufzeit im Worst Case offentlichtlich $O(n^2)$.

(ohne Beweis)

Shell Sort VI

Bemerkungen:

- Schrittweiten, die keine Potenzen von 2 sind, liefern in der Regel bessere Ergebnisse

- Experimentelle Ermittlung von Knuth:
  - (umgekehrte) Schrittweitenfolge $1, 4, 13, 40, 121, \ldots$
  - allgemein: $h_1 = 1, h_{s+1} = 3 \cdot h_s + 1$
    und beginne mit Schrittweite $\tilde{s}$, für die gilt: $h_{(\tilde{s}+2)} \geq n$.
  - Dadurch Laufzeit von $O(n^{1,25})$

- Unabhängig von der Schrittweite:
  Laufzeit nicht besser als $O(n^{1,2})$ $(> O(n \cdot \log n))$.