Teil III

Ausgewählte Algorithmen

Überblick

Motivation

- Kennenlernen von Beispielalgorithmen, auch als Grundlage für praktische Übungen
- Vorbereitung der theoretischen Betrachtungen, etwa Komplexität von Algorithmen
- informelle Diskussion von Design-Prinzipien

Suchen in sortierten Folgen

- Suchen als eine der häufigsten Aufgaben in der Informatik
- Vielzahl von Lösungen für unterschiedlichste Aufgaben
- hier: Suchen in sortierten Folgen
- Annahmen
  - Folge F als Feld von numerischen Werten
  - Zugriff auf i-tes Element über $F[i]$
  - nur Berücksichtigung des *Suchschlüssels*
- Beispiel: Telefonbuch, Suche nach Namen

Sequenzielle Suche: Algorithmus

```
algorithm SeqSearch (F, k) → p
Eingabe: Folge F der Länge n, Schlüssel k

   for i := 1 to n do
      if F[i] = k then
         return i
      fi
   od;
   return NO_KEY
```

Sequenzielle Suche: Aufwand

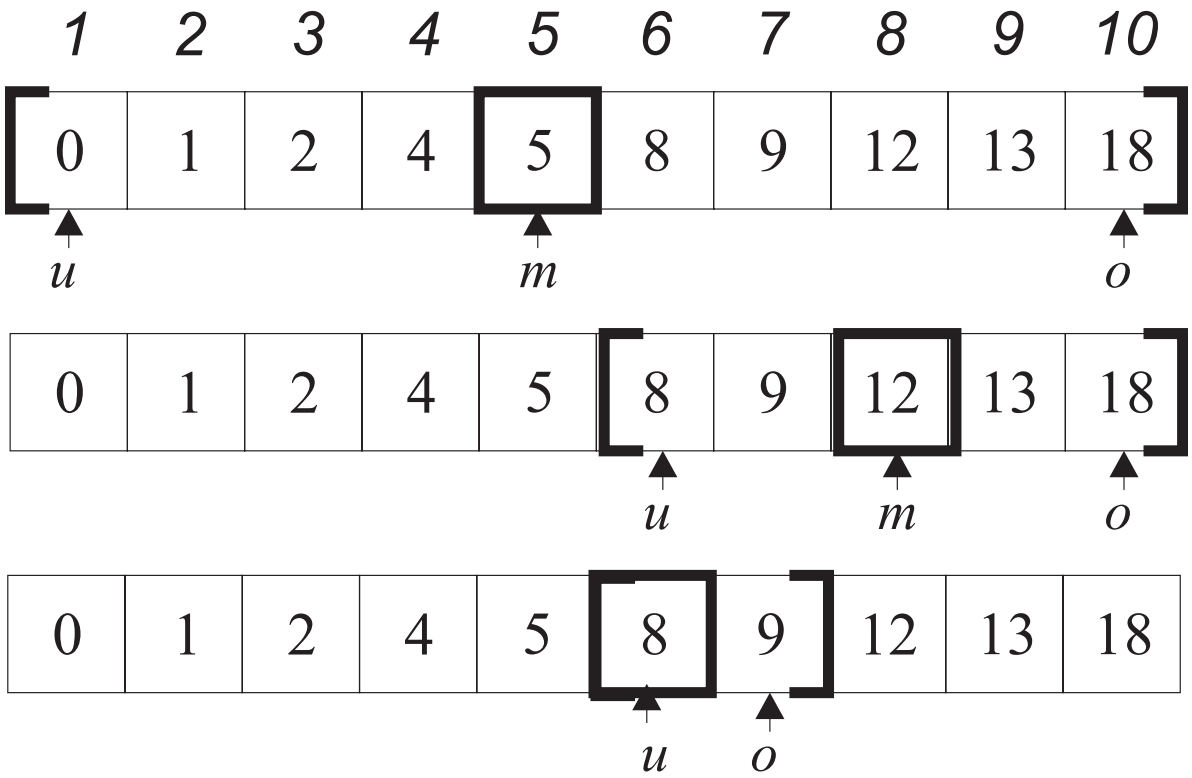
	Anzahl der Vergleiche
bester Fall	1
schlechtester Fall	n
Durchschnitt (erfolgreiche Suche)	$(n+1)/2$
Durchschnitt (erfolglose Suche)	n

Binäre Suche

- Prinzip:
  1. Wähle den mittleren Eintrag und prüfe ob gesuchter Wert in der ersten oder in der zweiten Hälfte der Liste ist.
  2. Fahre rekursiv mit der Hälfte vor, in der sich der Eintrag befindet.

Binäre Suche: Beispiel

Gesucht wird die 8:



Binäre Suche: Algorithmus

```
algorithm BinarySearch (F, k) → p
Eingabe: Folge F der Länge n, Schlüssel k

    u := 1, o := n;
    while u <= o do
        m := (u + o)/2;
        if F[m] = k then
            return m
        else if k < F[m] then o := m − 1
            else u := m + 1
        fi; fi;
    od;
    return NO_KEY
```

Binäre Suche: Aufwandsanalyse

- nach dem ersten Teilen der Folge: noch $n/2$ Elemente
- nach dem zweiten Schritt: $n/4$ Elemente
- nach dem dritten Schritt: $n/8$ Elemente
- ...
- allgemein: im i-ten Schritt max. $n/2^i$ Elemente $\rightsquigarrow \log_2 n$

Binäre Suche: Aufwand

	Anzahl der Schritte
bester Fall	1
schlechtester Fall	$\approx \log_2 n$
Durchschnitt (erfolgreiche Suche)	$\approx \log_2 n$
Durchschnitt (erfolglose Suche)	$\approx \log_2 n$

Vergleich der Suchverfahren

Verfahren	10	10^2	10^3	10^4
sequenziell $(n/2)$	≈ 5	≈ 50	≈ 500	≈ 5000
binär $(\log_2 n)$	≈ 3.3	≈ 6.6	≈ 9.9	≈ 13.3

Laufzeitanalyse/Komplexität eines Algorithmus

- Nach diesem einführenden Beispiel schauen wir uns die Grundlagen der Laufzeitanalyse etwas genauer an.
- Wir springen dazu zurück in das Kapitel “Eigenschaften von Algorithmen” auf die Folie 2-25.

Sortieren

- Grundlegendes Problem in der Informatik
- Aufgabe:
  - Ordnen von Dateien mit *Datensätzen*, die *Sortierschlüssel* enthalten
    - ⋆ z.B.: Name, Matrikelnummer, Datum, Dateigröße
  - Umordnen der Datensätze, so dass klar definierte Ordnung der Schlüssel (numerisch/alphabetisch) besteht
- Vereinfachung: nur Betrachtung der Schlüssel, z.B. Feld von `int`-Werten
- Konkrete Problemdefinition:
  - Eingabe: n Zahlen $(a_0, a_1, \ldots, a_{n-1})$
  - gewünschte Ausgabe: permutierte Zahlenfolge $(a_{\pi^{-1}(0)}, a_{\pi^{-1}(1)}, \ldots, a_{\pi^{-1}(n-1)})$ mit $a_{\pi^{-1}(i)} \leq a_{\pi^{-1}(i+1)}$
- Umordnen der Datensätze, so dass klar definierte Ordnung der Schlüssel (numerisch/alphabetisch) besteht

Sortieren: Grundbegriffe

- für den Schlüssel muss eine Vergleichsoperation (z.B. $\leq$) definiert sein.
- Duplikate sind bis auf weiteres zulässig.
- **Verfahren**
  - intern: in Hauptspeicherstrukturen (Felder, Listen)
  - extern: Datensätze auf externen Medien (Festplatte, Magnetband)

Stabilität

Ein Sortierverfahren heißt *stabil*, wenn es die relative Reihenfolge gleicher Schlüssel in der Datei beibehält.

- Beispiel: alphabetisch geordnete Liste von Personen soll nach Alter sortiert werden → Personen mit gleichem Alter weiterhin alphabetisch geordnet

Name	Alter
Endig, Martin	30
Geist, Ingolf	28
Höpfner, Hagen	24
Schallehn, Eike	28

Name	Alter
Höpfner, Hagen	24
Geist, Ingolf	28
Schallehn, Eike	28
Endig, Martin	30

Sortieren durch Einfügen (Insertion Sort): Prinzip

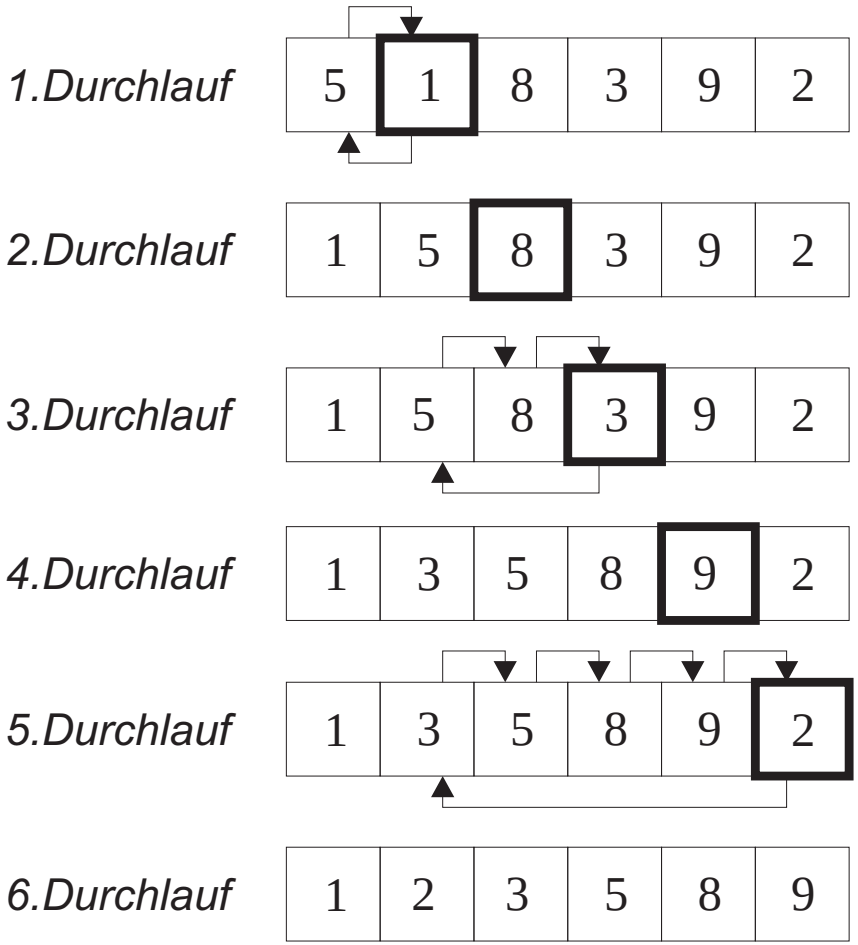
- **Idee**
  - Umsetzung der typischen menschlichen Vorgehensweise, etwa beim Sortieren eines Stapels von Karten:
    1. Starte mit der ersten Karte einen neuen Stapel
    2. Nimm jeweils nächste Karte des Originalstapels:
       füge diese an der richtigen Stelle in den neuen Stapel ein

Sortieren durch Einfügen: Algorithmus

```
algorithm InsertionSort (F)
Eingabe: zu sortierende Folge F der Länge n

    for i := 2 to n do
        m := F[i];  /* zu merkendes Element */
        j := i;
        while j > 1 do
            if F[j − 1] ≥ m then
                /* Verschiebe F[j − 1] nach rechts */
                F[j] := F[j − 1]; j := j − 1
            else
                Verlasse innere Schleife
            fi;
            F[j] := m  /* Einfügeposition */
        od
    od
```

Sortieren durch Einfügen: Beispiel



Analyse: InsertionSort

- Aufwand:
  - Anzahl der Vertauschungen
  - Anzahl der Vergleiche
  - Vergleiche **dominieren** Vertauschungen, d.h. es werden (wesentlich) mehr Vergleiche als Vertauschungen benötigt
- Ausserdem Unterscheidung:
  - Bester Fall: Liste ist schon sortiert
  - Mittlerer (zu erwartender) Fall: Liste ist unsortiert
  - Schlechtester Fall: z.B. Liste ist absteigend sortiert

Analyse: InsertionSort /2

- wir müssen in jedem Fall alle Elemente $i = 2$ bis n durchgehen, d.h. immer Faktor $n - 1$
- dann müssen wir zur korrekten Einfügeposition zurückgehen
- Bester Fall: Liste sortiert
  $\longrightarrow$ Einfügeposition ist gleich nach einem Schritt an Position $i - 1$
  - bei jedem Rückweg Faktor 1
  - Gesamtanzahl der Vergleiche:
    $(n - 1) * 1 = n - 1 = O(n)$

Analyse: InsertionSort /3

- Mittlerer (zu erwartender) Fall: Liste unsortiert
  $\rightarrow$ Einfügeposition wahrscheinlich auf der Hälfte des Rückwegs
  - bei jedem der $n-1$ Rückwege Faktor $(i-1)/2$
  - Gesamtanzahl der Vergleiche:

$$
\begin{aligned}
&(n-1)/2 + (n-2)/2 + (n-3)/2 + \cdots + 2/2 + 1/2 \\
&= \frac{(n-1) + (n-2) + \cdots + 2 + 1}{2} \\
&= \frac{1}{2} * \frac{n*(n-1)}{2} \\
&= \frac{n*(n-1)}{4} \\
&\approx \frac{n^2}{4} = O(n^2)
\end{aligned}
$$

Analyse: InsertionSort /4

- Schlechtester Fall: z.B. Liste umgekehrt sortiert
  $\longrightarrow$ Einfügeposition am Ende des Rückwegs bei Position 1
  - bei jedem der $n-1$ Rückwege Faktor $i-1$
  - analaog zu vorhergehenden Überlegungen, bloss doppelte Rückweglänge
  - Gesamtanzahl der Vergleiche:
    $\frac{n*(n-1)}{2} \approx \frac{n^2}{2} = O(n^2)$

Einschub: Java Collection Framework mit Generics

- Problem: Programme funktionieren nur für Elementtyp int, nicht für andere primitive Typen oder Klassen
- Ursachen:
  - Vergleichsoperator nötig, z.B. <
  - Deklaration von Hilfselementen, z.B. int element
  - Typangaben im Methodenkopf
- Ziel: Code-Wiederverwendung auch für unterschiedliche Typen
- allgemeine Lösung: generisches Programmieren (wird später behandelt)
- Spezialfall: Verwenden von Arrays.sort (aus java.util), alternativ mit:
  - Interface java.lang.Comparable oder
  - Interface java.util.Comparator

Das Interface `Comparable`

```
package java.lang;
public interface Comparable<T>{
   public int compareTo(T obj)
}
```

- obj1.compareTo(obj2) liefert ein x mit

$$x \begin{cases} < 0 & \text{falls } obj1 < obj2 \\ = 0 & \text{falls } obj1 == obj2 \\ > 0 & \text{falls } obj1 > obj2 \end{cases}$$

- ist in Integer, String etc. bereits implementiert
- ist von der Form “Vergleiche mich mit einem anderen Objekt gleichen Typs”
- kann in eigenen Klassen implementiert werden, s. nächste Folie

Beispiel zu `Comparable`

```
class Rectangle implements Comparable<Rectangle> {
   int length, width, x, y
   int area() { return length * width; }
   ...
   public int compareTo(Rectangle r) {
      return area() - r.area();
      }
}
```

Aufruf in anderer Klasse:

```
...
Rectangle[] myArray;
...
Arrays.sort(myArray); // aus java.util importieren
```

Das Interface `Comparator`

```
package java.lang;
public interface Comparator<T>{
   public int compare(T obj1, T obj2)
}
```

- compare(x,y) liefert ein x mit

$$x \begin{cases} < 0 & \text{falls } obj1 < obj2 \\ = 0 & \text{falls } obj1 == obj2 \\ > 0 & \text{falls } obj1 > obj2 \end{cases}$$

- kann mehrfach implementiert werden
  - bspw. wenn eine Menge von Strings einmal alphabetisch und einmal nach ihrer Länge sortiert werden sollen

Beispiel zu `Comparator`

```
class Myorder implements Comparator<Rectangle> {
   public int compare(Rectangle r1, Rectangle r2){
      return area() − r.area();
}
}
```

Aufruf in anderer Klasse:

```
...
Rectangle[] myArray;
...
Arrays.sort(myArray, new MyOrder());
// aus Paket java.util importieren
```

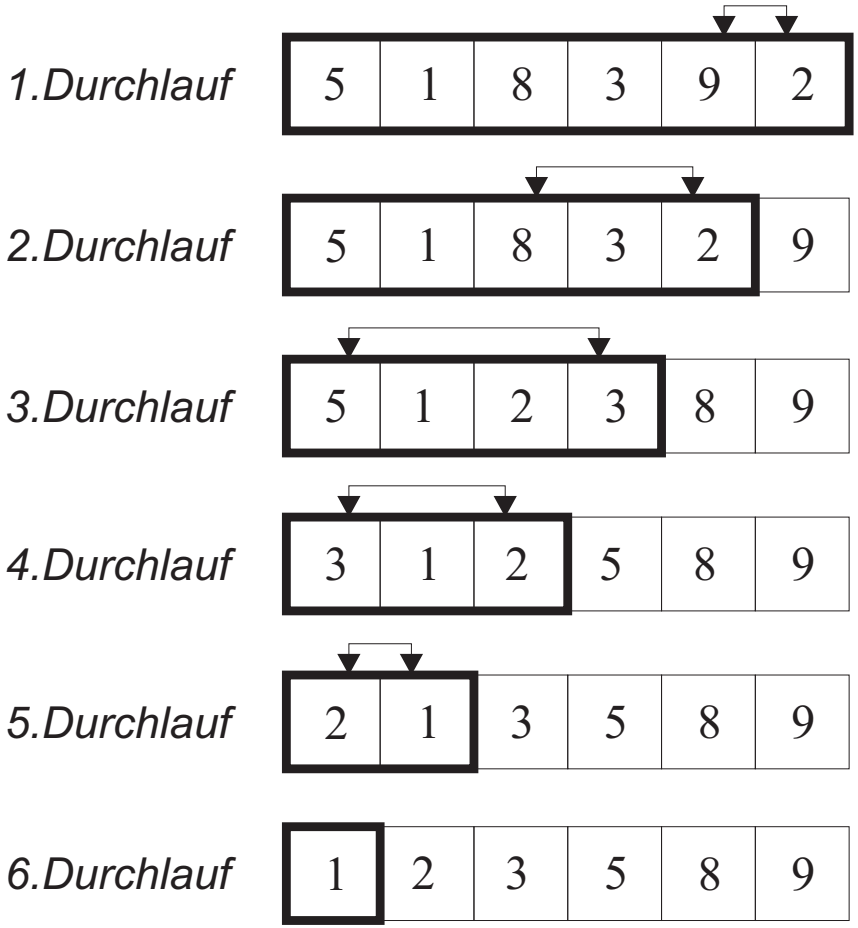
Sortieren durch Selektion

- **Idee:** Suche jeweils größten Wert, und tausche diesen an die letzte Stelle; fahre dann mit der um 1 kleineren Liste fort.

```
algorithm SelectionSort (F)
Eingabe: zu sortierende Folge F der Länge n

    p := n;
    while p > 0 do
        g := Index des größten Elementes aus F
             im Bereich 1...p;
        Vertausche Werte von F[p] und F[g];
        p := p − 1
    od
```

Sortieren durch Selektion: Beispiel



Analyse: SelectionSort

- in jedem Durchlauf das Element von $F[p]$ mit dem größten Element tauschen
- Variable p läuft von $n \ldots 1$
  $\rightsquigarrow$ daher n Vertauschungen
- in jedem Durchlauf das größte Element aus $1 \ldots p$ ermitteln
  $\rightsquigarrow$ $p - 1$ Vergleiche

$$(n-1) + (n-2) + (n-3) + \cdots + 2 + 1 = \frac{n(n-1)}{2} \approx \frac{n^2}{2} = O(n^2)$$

- **Anzahl Vergleiche identisch für besten, mittleren und schlechtesten Fall!**

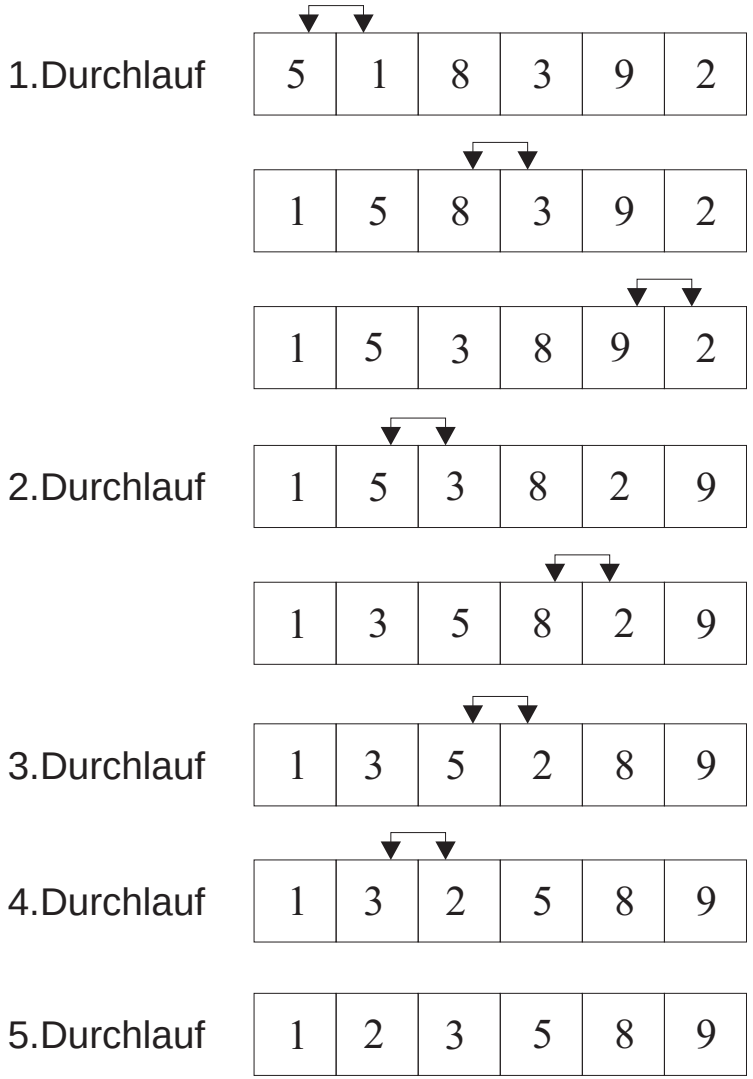
BubbleSort

- **Idee:** Verschieden große aufsteigende Blasen („Bubbles“) in einer Flüssigkeit sortieren sich quasi von allein, da größere Blasen die kleineren „überholen“.

```
algorithm BubbleSort (F)
Eingabe: zu sortierende Folge F der Länge n

   do
      for i := 1 to n − 1 do
         if F[i] > F[i + 1] then
            Vertausche Werte von F[i] und F[i + 1]
         fi
      od
   until keine Vertauschung mehr aufgetreten
```

BubbleSort: Beispiel



BubbleSort: Variante und Optimierung

Variante

- abwechselnd von oben und unten beginnen

Optimierung

- größte Zahl rutscht in jedem Durchlauf automatisch an das Ende der Liste
- im Durchlauf j reicht die Untersuchung bis Position $n - j$

Analyse: BubbleSort

- Bester Fall: n
- Durchschnittlicher Fall
  normal: n^2
  optimiert: $\frac{n^2}{2}$
  d.h. beides $O(n^2)$
- Schlechtester Fall
  normal: n^2
  optimiert: $\frac{n^2}{2}$
  d.h. beides $O(n^2)$

Vergleich der $O(n^2)$-Algorithmen

Bubblesort

- einfach zu kodieren
- schneller Abbruch für sortierte Folge
- hohe Konstante
- profitiert nur von wenigen Arten der Vorsortiertheit
- i.A. nicht zu empfehlen, ausser weil kurz

Selection Sort

- nur n Datentransporte
- gleicher Aufwand für bereits sortierte Folge
- kleine Konstante, da wenig Tauschoperationen

Insert Sort

- ist schneller je sortierter die Folge ist
- hoher Aufwand für Datentransporte, profitiert von Liste (vs. Array)

Vorteil der $O(n^2)$-Algorithmen

. . . gegenüber den folgenden $O(n \log n)$-Algorithmen:

- leicht zu kodieren
- geringer Overhead
- sind für kleine n schneller

MergeSort: Prinzip

Idee:

1. Teile die zu sortierende Liste in zwei Teillisten
2. Sortiere diese (rekursives Verfahren!)
3. Mische die Ergebnisse

Prinzip: Divide et Impera (Teile und Herrsche, Divide and Conquer)

1. Zerlege Problem P in Teilprobleme $P_1, \ldots, P_n$
2. Finde Lösungen $L_1, \ldots, L_n$ der Teilprobleme
3. Setze Lösung L von P als Kombination der $L_1, \ldots, L_n$ zusammen

Rekursive Anwendung bis die Teilprobleme trivial sind.

Mischen von zwei Folgen: Algorithmus

procedure Merge $(F_1, F_2) \rightarrow F$
Eingabe: zwei zu sortierende Folgen F_1, F_2
Ausgabe: eine sortierte Folge F

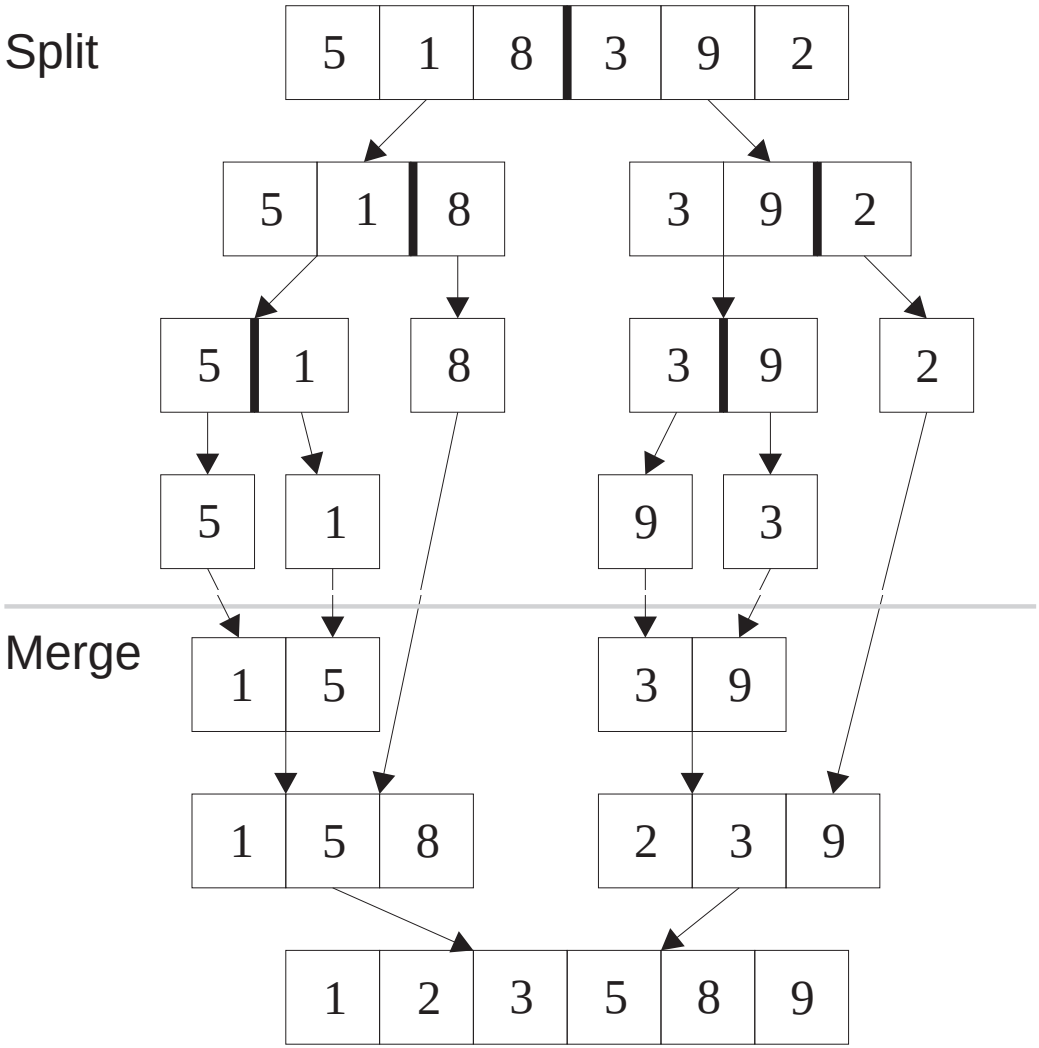
$F :=$ leere Folge;
while F_1 oder F_2 nicht leer **do**
  Entferne das kleinere der Anfangselemente
    aus F_1 bzw. F_2;
  Füge dieses Element an F an
od;
Füge die verbliebene nichtleere Folge F_1
  oder F_2 an F an;
return F

Sortieren durch Mischen: Algorithmus

algorithm MergeSort $(F) \rightarrow F_S$
Eingabe: eine zu sortierende Folge F
Ausgabe: eine sortierte Folge F_S

```
if F einelementig then
    return F
else
    Teile F in F1 und F2;
    F1 := MergeSort (F1);
    F2 := MergeSort (F2);
    return Merge (F1, F2)
fi
```

MergeSort: Beispiel



Analyse: MergeSort

- 1. Schritt:
  - ‘Ablesen’ der Rekursionsformel aus dem Diagramm.
  - Faktoren vor dem T dürfen nicht weggelassen werden.
- Vereinfachende Annahme: $n = 2^k$
- Dann gilt:

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ 2T(\frac{n}{2}) + cn & \text{falls } n > 1 \end{cases}$$

Analyse: MergeSort /2

- 2. Schritt: Herleiten der expliziten Rekursionsformel:

$$
\begin{aligned}
T(n) &\leq 2T(\tfrac{n}{2}) + cn \\
&\leq 2(2T(\tfrac{n}{4}) + c\tfrac{n}{2}) + cn \\
&\leq 4T(\tfrac{n}{4}) + 2cn \\
&\leq 8T(\tfrac{n}{8}) + 3cn \\
&\dots \\
&\leq 2^i T(\tfrac{n}{2^i}) + icn \\
&\dots \\
&\leq 2^k T(\tfrac{n}{2^k}) + kcn \\
&= nT(1) + (\log_2 n)cn \\
&= nc + cn \log n \\
&= O(n \log n)
\end{aligned}
$$

MergeSort-Variante

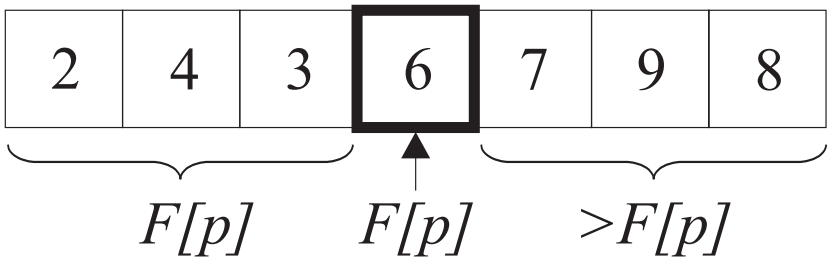
- Hilfsfeld mit Maximalgröße anlegen und wiederverwenden
- Abbruch bei Feldlänge k, darunter z.B. Insertion Sort
- nicht-rekursive Formulierung: Mischen sortierter Teilfolgen der Längen $1, 2, 4, \ldots, \frac{n}{2}$
- natürliches Mergesort: Start mit bereits sortierten Teilfolgen
- Aufteilung in mehr als 2 Teilfelder
- Mergesort eignet sich für externes Sortieren

Korrektheit des Quicksort-Algorithmus

- Am nächsten Algorithmus werden wir das Prinzip von Korrektheitsbeweisen anschauen.
- Für die nötige Theorie springen wir zunächst zurück in das Kapitel “Eigenschaften von Algorithmen”.

QuickSort: Prinzip

- **Idee:**
  - ähnlich wie MergeSort durch rekursive Aufteilung
  - Vermeidung des Mischvorgangs durch Aufteilung der Teillisten in zwei Hälften bezüglich eines Pivot-Elementes, wobei
    - ★ in der ersten Hälfte alle Elemente kleiner als das Referenzelement sind
    - ★ in der zweiten Hälfte alle Elemente größer sind



QuickSort: Algorithmus

```
algorithm QuickSort (F, left, right)
   Eingabe: eine Folge F,
      die untere (left) und obere Grenze (right)
      der zu sortierenden Teilfolge

   if left < right then
      pivIndex = Partition(F,left,right)
      QuickSort (F, left, pivIndex-1);
      QuickSort (F, pivIndex+1, right)
   fi
```

QuickSort: Partition

- Auswählen eines Pivot-Elements p und nach ganz rechts ("aus dem Weg") tauschen.
- Folge von links durchgehen.
- Immer wenn ein Element gefunden wird, das größer oder gleich $\leq p$ ist, dann dieses nach vorne in den Block der "$\leq p$-Elemente" tauschen.
- Siehe Tafelskizze.

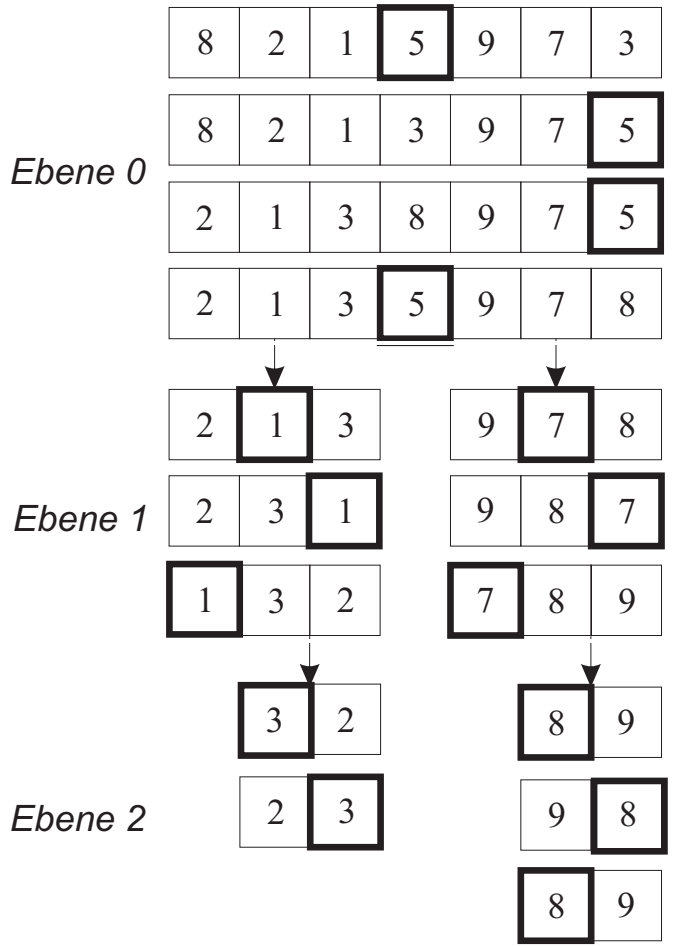
QuickSort: Zerlegen beim QuickSort

```
algorithm Partition(F,left,right)
   Eingabe: Folge F, Grenzen left bzw. right
   Ausgabe: Position i des Pivot-Elements

   swap(F, (left+right)/2, right)
   p := F[right] // Pivot-Element wird festgelegt
   i:= left
   for j := left to right do
      if F[j] ≤ p then swap(F, i, j); i:=i+1 fi
   od
   swap(F, i, right)
   return i;
```

Anm.: Die erste Zeile ist nicht nötig, aber in vielen Anwendungen liegen mittelgroße Werte (das sind die guten Pivot-Werte) eher in der Mitte.

QuickSort: Beispiel



Korrektheit der Partitionierung

1. Geeignete Schleifeninvariante P finden:
   $P := ((left \leq k < i \implies F[k] \leq p) \wedge (i \leq k < j \implies F[k] > p))$
2. Prüfen, dass P vor Schleife gilt:
   P gilt wegen $i = j = left$
3. Zeigen, dass $\{P \wedge B\}\ \beta\ \{P\}$ gilt:
   an Tafel
4. $P \wedge \neg B$ und **`swap`**(F, i, right) erzwingen Partitionierung:
   Gilt wegen P und $j = right$.

Analyse: QuickSort

- Aufwand analog zu MergeSort abschätzbar
  - Bester Fall: $n \log n$
  - Durchschnittlicher Fall: $n \log n$
  - Schlechtester Fall: $n \log n$
- aber: im Gegensatz zur MergeSort ist QuickSort durch Vorgehensweise bei Vertauschungen **instabil**

Aufwand der Sortierverfahren bis hier im Vergleich

Verfahren	Stabilität	Vergleiche (im Mittel)
SelectionSort	instabil	$\approx n^2/2 = O(n^2)$
InsertionSort	stabil	$\approx n^2/4 = O(n^2)$
BubbleSort	stabil	$\approx n^2/2 = O(n^2)$
MergeSort	stabil	$O(n \log n)$
QuickSort	instabil	$O(n \log n)$

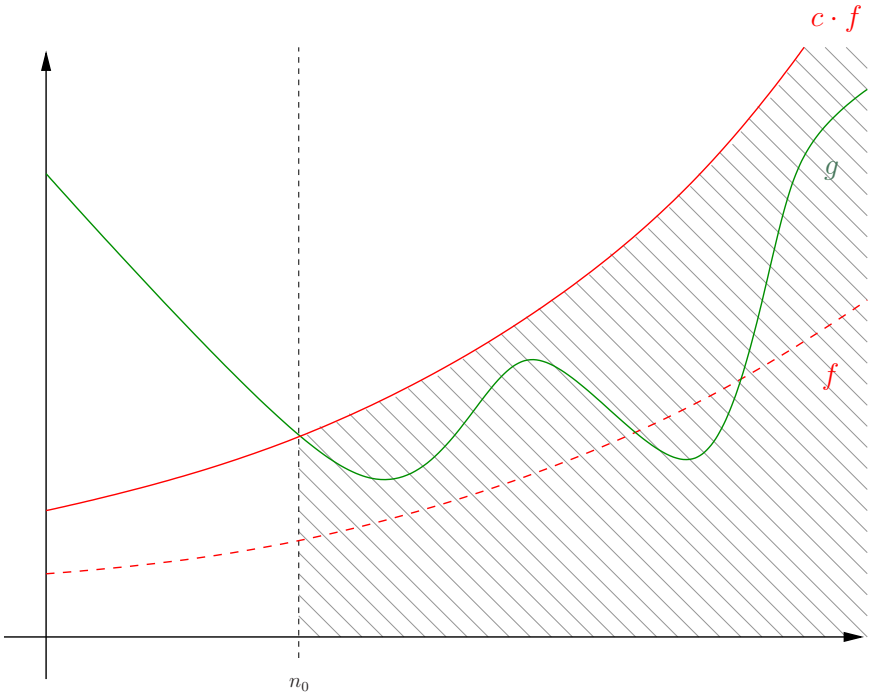
Untere Schranke: Einführung

- Laufzeiten fürs Sortieren:
  - $O(n^2)$ – Insertion Sort, Selection Sort, Bubblesort
  - $O(n \log n)$ – Mergesort, Quicksort
- Vermutung: schneller geht es nicht,
  - d.h., jeder Sortieralgorithmus benötigt mindestens die Zeit $cn \log n$ für ungünstige Eingaben.
  - d.h., $T(n) = \Omega(n \log n)$.

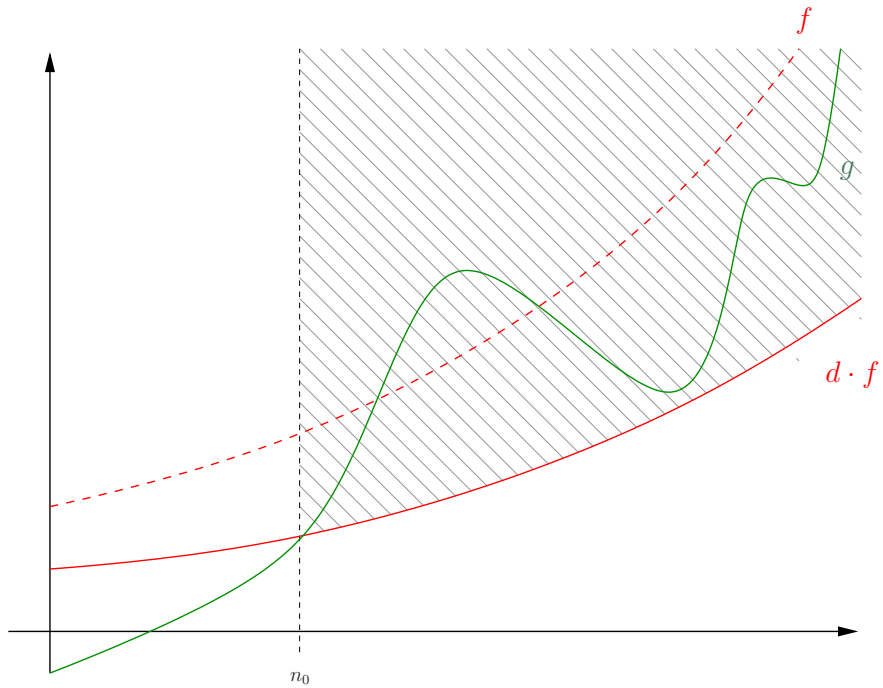
$$\Omega(g(n)) := \{f(n) \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : 0 \leq cg(n) \leq f(n)\}$$

Bemerkung: Die untere Schranke wird üblicherweise für die Betrachtung einer ganzen Algorithmen-/Problem-Klasse verwandt.

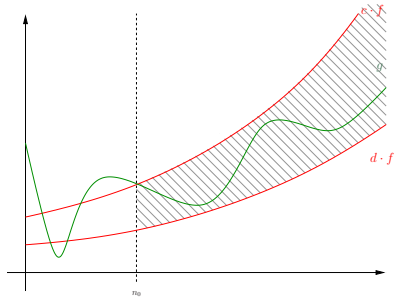
Obere und untere Schranken



obere Schranke: $O(f(n))$



untere Schranke: $\Omega(f(n))$



obere und untere Schranke: $\Theta(f(n))$

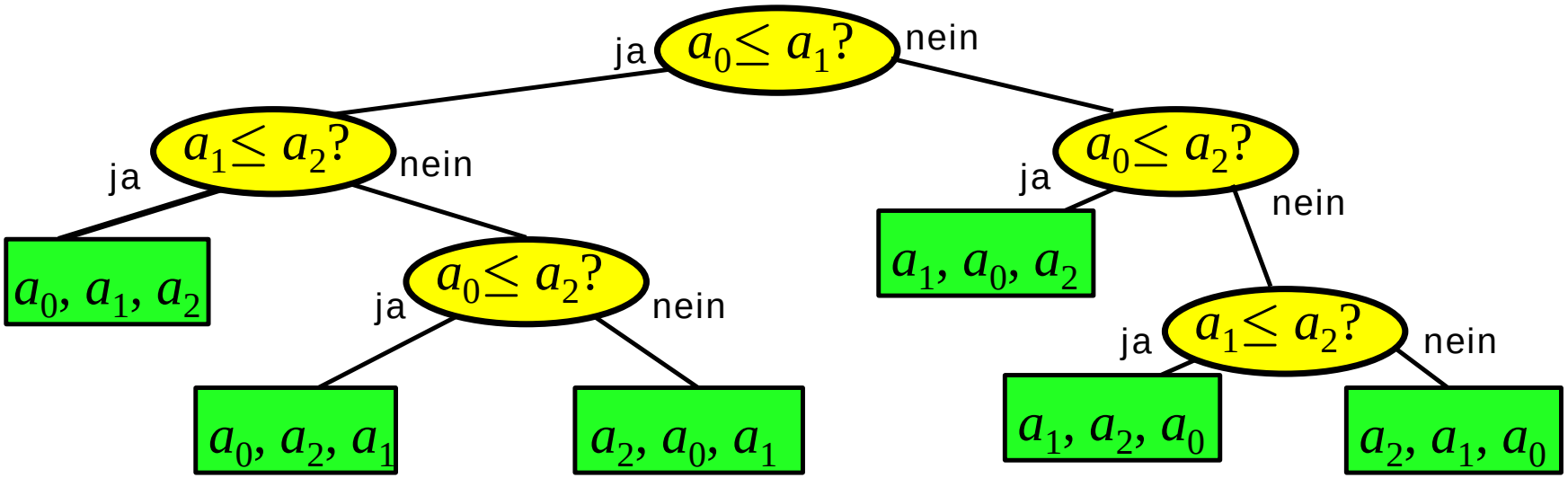
Untere Schranke: Behauptung

Behauptung: Für die worst-case-Laufzeit jedes vergleichsbasierten Sortieralgorithmus gilt $T(n) = \Omega(n \log n)$.

- Vergleichsbasiert: Die einzige Möglichkeit des Algorithmus, Informationen über die zu sortierenden Daten zu erhalten, sind Vergleiche der Form $a[i] \leq a[j]$.
- Alle bisherigen Algorithmen sind vergleichsbasiert.
- Die Behauptung besagt: Zu jedem vergleichsbasierten Sortieralgorithmus existieren Konstanten c und n_0, so dass es für jede Eingabegröße $n \geq n_0$ mindestens eine konkrete Eingabe gibt, für die der Algorithmus mindestens die Zeit $cn \log n$ benötigt.

Untere Schranke: Beweis

- Jeder vergleichsbasierte Algorithmus kann durch einen Entscheidungsbaum dargestellt werden.
- Beispiel: Insertion Sort für $n = 3$:



Distribution Sort: Einführung

- nicht vergleichsbasiert
- lineare Laufzeit
- verwandte Verfahren:
  - Radix Sort
  - Bucketsort
  - Binsort
  - Sortieren durch Fachverteilung
- Voraussetzung: Sortierschlüssel = “Ziffernfolge” fester Länge
  - z.B. Postleitzahl, ISBN-Nummer, IP-Adresse, ganze Zahl maximaler Größe
  - Damit ist die maximale Anzahl verschiedener Datensätze begrenzt.
- Literatur: Gumm et al.: Einführung in die Informatik

Distribution Sort: Algorithmus

Der Algorithmus verbindet drei Ideen:

- Sortieren erfolgt ziffernweise durch Einordnen in “Körbe”
  - je ein Korb für die Ziffern 0 bis 9
  - Zunächst nach einer Ziffer sortieren, dann nach der nächsten.
  - Dabei die vorige Reihenfolge beachten. (*)
- Verwenden eines großen Arrays statt Einzel-Körben
  - Zunächst zählen, wieviele Schlüssel in jeden der Körbe gehören
  - Daraus Berechnung der Anfangsadressen der Körbe im Array
  - Schließlich Einordnen der Schlüssel in die Körbe
- Mit letzter Ziffer beginnen
  - Spart Aufwand für die Verwaltung der Körbe im Array.
  - Wegen (*) ist das Einordnen in die Körbe stabil,
  - d.h. die Sortierung nach letzter Ziffer bleibt beim Sortieren nach vorderen Ziffern erhalten.

Distribution Sort: Beispiel

Fünf Matrikelnummern sollen sortiert werden: 94032, 83512, 90459, 56410, 53419.

- Sortieren nach letzter Ziffer:
  [0:56410] [1] [2:94032,83512] [3] [4] [5] [6] [7] [8] [9:90459,53419]
- Sortieren nach vorletzter Ziffer:
  [0] [1:56410,83512,53419] [2] [3:94032] [4] [5:90459] [6] [7] [8] [9]
- Sortieren nach vorvorletzter Ziffer:
  [0:94032] [1] [2] [3] [4:56410,53419,90459] [5:83512] [6] [7] [8] [9]
- Sortieren nach vorvorvorletzter Ziffer:
  [0:90459] [1] [2] [3:53419,83512] [4:94032] [5] [6:56410] [7] [8] [9]
- Sortieren nach erster Ziffer:
  [0] [1] [2] [3] [4] [5:53419,56410] [6] [7] [8:83512] [9:90459,94032]
- Endergebnis: 53419, 56410, 83512, 90459, 94032

Distribution Sort: Algorithmus

```
N := a.length
for k:=m downto 0 do     // für jede Ziffer
   for i:=0 to N-1 do       // Korbgrößen bestimmen
      count(a[i][k])++
   for z:=1 to d-1 do       // Korbgr. aufsummieren
      count(z)=count(z)+count(z-1)
   for z:=d-1 downto 1 do   // Fach für Zeichen z
      count(z)=count(z-1)   // beginnt jetzt an
      count[0]=0;           // Position count[z]
      od
   for i:=0 to N do         // Einsortieren
      z:=a[i][k]            // in Hilfs-
      b[Count[z]]:=a[i]     // Array b,
      Count[z]++            // dabei Korbgrenze anpassen
      od
od
b:=a;                       // Array zurückkopieren
```

Distribution Sort: Laufzeit

- pro Ziffer: cn
- insgesamt: $kcn = O(n)$ für n Schlüssel der Form $x_{k-1} \dots x_0$ mit $x_i \in \{0, \dots, d-1\}$
  - NB: maximal dk verschiedene Schlüssel möglich
- Konstanten d, k werden hinter O-Notation "versteckt"
- Verfahren ist für große Anzahl kurzer Schlüssel (die doppelt vorkommen können) effizient.
- Anderenfalls können die "unsichtbaren" Konstanten größer sein als der $\log(n)$-Faktor bei Quicksort.

Zusammenfassung

- Suchen und Sortieren als Grundaufgaben in der Informatik
- Grundalgorithmen
- Algorithmenmuster „Teile und herrsche“
- Abschätzung des Aufwands
- Literatur:
  - Saake/Sattler: *Algorithmen und Datenstrukturen*, Kap. 5
  - Gumm, Sommer: *Einführung in die Informatik*, Kap. 4