# Sortieralgorithmen

by

Dr. Günter Kolousek

# Einführung

- zugrundeliegendes Problem
  - Sequenz von Daten (z.B. ein Array mit ganzen Zahlen)
  - nach einem bestimmten Kriterium zu sortieren
  - Meist: Schlüssel $k$ und Nutzdaten
- aufsteigend vs. absteigend
  - je zwei Schlüssel müssen vergleichbar sein bzgl. $<$ bzw. $>$
- Sequenz $S$ von Datensätzen (items): $S = s_0, s_1, \ldots, s_{n-1}$
- Gesucht ist eine Sequenz $S' = s_{\pi(0)}, s_{\pi(1)}, \ldots, s_{\pi(n-1)}$
  - für die gilt, dass die Schlüssel $k_{\pi(i)}$ aufsteigend sortiert sind:

    $$k_{\pi(0)} \leq k_{\pi(1)} \leq \ldots \leq k_{\pi(n-1)}$$

    wobei $\pi$ eine Permutation der Zahlen $0$ bis $n-1$ sein soll.

# Kriterien

- Ort der Daten
  - passt in Hauptspeicher: wahlfreier Zugriff
  - oder nicht (z.B. Band): seq. Zugriff (ext. Sortierverfahren)
- Anzahl der zu sortierenden Datensätze
- Operation
  - Müssen die Datensätze pyhsisch bewegt werden oder
  - Information berechnen, sodass sortiertes Durchlaufen

# Kriterien - 2

- Speicherplatzbedarf
  - zusätzlicher Speicher zum Sortieren nötig? wieviel?
- Stabilität
  - Veränderung der Reihenfolge von Elementen mit **gleichem** Schlüssel (nicht stabil) oder nicht (stabil)?
- Laufzeitverhalten
  - bei steigender Anzahl der Datensätze?
  - bzgl. der Anordnung (zufällig, vorsortiert, gegenteilig)?

# Bubble-Sort

- Idee
  1. Beginne vorne.
  2. Vergleiche jeweils 2 benachbarte Zahlen und vertausche diese, wenn diese nicht in der richtigen Reihenfolge sind.
     - Danach ist sicher die größte Zahl am rechten Ende, aber alle anderen Zahlen sind unsortiert.
  3. Beginne deshalb wieder von vorne
- Name
  - Jeweils größtes Element steigt wie Blase an Oberfläche
- Beispiel
  - `[15, 2, 43, 17, 47, 8, 4]`
  - `[2, 15, 43, 17, 47, 8, 4]`
  - `[2, 15, 43, 17, 47, 8, 4]`
  - `[2, 15, 17, 43, 47, 8, 4]`
  - ...

# Bubble-Sort – 2

- Prinzip

  ```
  für jedes x der Liste bis zum Vorletzten:
   für jedes y der Liste bis zum Vorletzten:
    wenn y größer ist als dessen Nachfolger:
     vertausche y mit seinem Nachfolger
  ```

- Algorithmus

```python
def bubble_sort1(lst):
    for x in lst[:-1]:
        for j in range(len(lst) - 1):
            if lst[j] > lst[j + 1]:
                lst[j], lst[j+1] = lst[j+1], lst[j]
    return lst
```

# Bubble-Sort – 3

- Verbesserungsmöglichkeit?
  - Warum beim inneren Durchlauf die Liste nochmals bis zum ”Ende” durchlaufen, wenn das letzte Element sicher schon richtig ist?

# Bubble-Sort – 4

- Prinzip

  ```
  für jedes El. top vom Letzten zum Ersten:
   für jedes y bis zum Vorletzten (vor top):
    wenn y größer ist als dessen Nachfolger:
     vertausche y mit seinem Nachfolger
  ```

- Algorithmus

```python
def bubble_sort2(lst):
    for i_top in range(len(lst)-1, 0, -1):
        for j in range(i_top):
            if lst[j] > lst[j + 1]:
                lst[j], lst[j+1] = lst[j+1], lst[j]
    return lst
```

# Bubble-Sort – 4

- Verbesserungsmöglichkeit?
  - Wenn die Daten schon weitgehend sortiert sind, wie in z.B. in [1, 9, 2, 3, 4, 5]? Ein Durchgang reicht und die Liste ist schon sortiert.

# Bubble-Sort – 5

- Prinzip
  - Wenn sich offensichtlich in einem weiteren Durchgang nichts ändert, dann kann die äußere Schleife abbrechen.
- Algorithmus

```
def bubble_sort3(lst):
    for i_top in range(len(lst)-1, 0, -1):
        changed = False
        for j in range(i_top):
            if lst[j] > lst[j + 1]:
                lst[j], lst[j+1] = lst[j+1], lst[j]
                changed = True
        if not changed:
            break
    return lst
```

# Bubble-Sort – 6

- Verbesserungsmöglichkeit?
  - Warum beim inneren Durchlauf bis zum *jeweils* letzten Element durchlaufen, wenn die letzte Änderung beim vorhergehenden Durchlauf schon früher stattgefunden hat.

# Bubble-Sort – 6

- Verbesserungsmöglichkeit?
  - Warum beim inneren Durchlauf bis zum *jeweils* letzten Element durchlaufen, wenn die letzte Änderung beim vorhergehenden Durchlauf schon früher stattgefunden hat.
- Prinzip
  - Kombination der letzten beiden Optimierungen: Heruntersetzen der oberen Grenze, auf die letzte Position an der noch eine Vertauschung stattgefunden hat. Damit wird keine Vertauschung mehr durchgeführt, wenn diese obere Grenze der ”Anfang” ist.

# Bubble-Sort – 7

```
def bubble_sort4(lst):
  n = len(lst)
  while True:
    new_n = 1
    for y in range(n - 1):
      if lst[y] > lst[y + 1]:
        lst[y], lst[y + 1] = lst[y + 1], lst[y]
        new_n = y + 1
    n = new_n
    if n == 1:
      break
  return lst
```

# Bubble-Sort – 8

- Bewertung Warum ist dieser Algorithmus denn eigentlich so schlecht?
  - Weil bei n Elemente größenordnungsmäßig $n^2$ Vergleiche und Vertauschoperationen notwendig sind,
    - da zwei verschachtelte Schleifen jeweils für (fast) alle Elemente durchlaufen werden.
- Vorteil: nur 3 Zeilen Code (je nach Programmiersprache)
- Noch schlechtere Algorithmen?
  - Bilde alle Permutationen und finde geordnete Liste
    - maximale Speicherplatzverschwendung
  - Solange Liste nicht sortiert, vertausche 2 **beliebige** Elemente
    - maximales Glücksspiel

# Selection-Sort

- Idee
  1. Lege neue Ergebnisliste an
  2. Finde das kleinste Element in der Liste.
  3. Hänge dieses Element an die Ergebnisliste
  4. entferne es aus der Liste
  5. wenn die Liste noch nicht leer ist, gehe zu Schritt 1) zurück.
- Name
  - Auswahl des jeweils kleinstem Elementes aus der Liste
- Beispiel
  - `[15, 2, 43, 17, 47, 8, 4]`
  - `[2],[15, 43, 17, 47, 8, 4]`
  - `[2, 4],[15, 43, 17, 47, 8]`
  - ...

# Selection-Sort – 2

- Prinzip
  1. Finde Position $j_0$ des kleinsten Elementes von $a[0], ..., a[n-1]$ und verstausche $a[0]$ mit $a[j_0]$.
  2. Finde Position $j_1$ des kleinsten Elementes von $a[1], ..., a[n-1]$ und vertausche $a[1]$ mit $a[j_1]$.
     - das ist das Element mit dem zweitkleinstem Schlüssel unter allen n Elementen
  3. Das wird solange durchgeführt bis alle Elemente an ihrem richtigen Platz stehen.
- Frage: Wo ist da die Ergebnisliste?
  - Geht auch *mit* Ergebnisliste
  - je nach Programmiersprache mit Array (je nach Aufgabenstellung: effizienter)

# Selection-Sort – 3

- Algorithmus

```
def selection_sort(seq):
    n = len(seq)
    for i in range(n - 1): # i von 0 bis n-2
        min = i
        # j von i + 1 bis n - 1
        for j in range(i + 1, n):
            if seq[j] < seq[min]:
                min = j
        seq[min], seq[i] = seq[i], seq[min]
    return seq
```

- Aufgabe: schreibe eine rekursive Variante `rec_selection_sort`

# Insertion-Sort – 1

- Idee
  1. Lege eine neue Ergebnisliste mit dem ersten Element der zu sortierenden Liste an.
  2. Gehe alle Elemente der zu sortierenden Liste von Position 1 bis zum Ende durch und füge das aktuelle Element in der Ergebnisliste an der richtigen Position ein.
- Name
  - Einfügen in Ergebnisliste
- Beispiel
  - `[15, 2, 43, 17, 47, 8, 4]`
  - `[15],[2, 43, 17, 47, 8, 4]`
  - `[2, 15],[43, 17, 47, 8, 4]`
  - `[2, 15, 43],[17, 47, 8, 4]`
  - ...

# Insertion-Sort – 2

- Prinzip

```
Für jede Position i von 1 bis zur Letzten:
  Wert val mit dem Wert von i belegen
  Index j mit i belegen
  Endlosschleife
    Wenn j == 0, dann Schleife beenden
    Wenn Wert an j-1 <= val
      dann Schleife beenden
    Element an Pos. j mit Pos. j-1 belegen
    j dekrementieren
  Liste an der Position j mit val belegen
```

# Insertion-Sort – 3

- Algorithmus

```
def insertion_sort(seq):
    # vom zweiten Element bis zum letzten El.
    for i in range(1, len(lst)):
        val = lst[i]
        j = i
        while True:
            if j == 0 or lst[j - 1] <= val:
                break
            lst[j] = lst[j - 1]
            j -= 1
        lst[j] = val
    return lst
```

- Aufgabe: schreibe eine rekursive Variante `rec_insertion_sort`

# Quick-Sort

- Idee
  1. sortiere grob in 2 Teile, indem ein Trennwert gewählt wird
     - Partition!
     - Heuristik: letzten Wert in Sequenz als Trennwert nehmen
  2. sortiere den ersten Teil
  3. sortiere den zweiten Teil
  4. setze die Teile zusammen
- Name…
- Beispiel
  - `[15, 2, 43, 17, 4, 8, 47]` $\rightarrow$ Pivot: 47
  - `[15, 2, 43, 17, 4, 8]`, `[]` $\rightarrow$ Pivot: 8
  - `[2, 4]`, `[17, 15, 43]` $\rightarrow$ Pivot: 4 und 43
  - `[2]`, `[]`
  - `[17, 15]`, `[]` $\rightarrow$ Pivot: 15
  - `[]`, `[17]`

# Quick-Sort – 2

- Algorithmus - 1

```
def partition(lst, left, right):
    pivot = lst[right]
    i = left - 1
    j = right
    while True:
        i += 1
        while i <= right and lst[i] < pivot:
            i += 1
        j -= 1
        while j >= left and lst[j] > pivot:
            j -= 1
        if i < j:
            lst[i], lst[j] = lst[j], lst[i]
        else:
            break
    lst[i], lst[right] = lst[right], lst[i]
    return i
```

# Quick-Sort – 3

- Algorithmus - 2

```
def quicksort_(lst, start, end):
  if start < end:
    p = partition(lst, start, end)
    quicksort_(lst, start, p - 1)
    quicksort_(lst, p + 1, end)

def quicksort(lst):
  quicksort_(lst, 0, len(lst) - 1)
```

# Quick-Sort – 4

- Ordnung: $n \log(n)$
  - im schlechtesten Fall: $n^2$
    - wenn Pivot-Element schlecht gewählt: wenn letztes Element und sortierter Liste
- rekursiver Algorithmus kann in iterativen umformuliert werden
  - Rekursionen können gespart werden, wenn bei Teillisten mit Länge $\leq 5$ Insertion-Sort verwendet wird.

# Laufzeitvergleiche

- 1000, 2000, 4000 Zahlen auf einem Subnotebook
  - Zeit in Sekunden, gerundet auf 4 Nachkommastellen
  - Abhängig von: hauptsächlich CPU, Speicher, aktuelle Auslastung, aktuellen Daten (zufällige Zahlen!).

| | | | |
|---|---|---|---|
| bubble1 | 0.6176 | 2.5324 | 10.1482 |
| bubble2 | 0.3845 | 1.6154 | 6.4617 |
| bubble3 | 0.4020 | 1.6620 | 6.6788 |
| bubble3/sortiert | 0.0004 | 0.0009 | 0.0018 |
| selection | 0.1787 | 0.7077 | 2.8736 |
| selection/sortiert | 0.1763 | 0.7098 | 2.8491 |
| insertion | 0.2134 | 0.8688 | 3.5896 |
| insertion/sortiert | 0.0011 | 0.0018 | 0.0009 |
| quicksort | 0.0087 | 0.0188 | 0.0412 |
| quicksort/sortiert | 0.3945 | 1.5862 | 6.3666 |

quicksort $\rightarrow$ sys.setrecursionlimit(5000)