Bubblesort I – Grundprinzipen

- paarweiser Vergleich und Vertauschen von Elementen des Feldes
- nach jedem Vergleich wird Zähler um eins erhöht und nächste Paarung betrachtet
- pro Durchlauf wird somit das jeweils größte Element an seine Position in der Unterliste verschoben
- Algorithmus ist beendet, wenn in einem Durchlauf keine Elemente vertauscht werden mussten

Bubblesort II – Erklärungsbeispiel

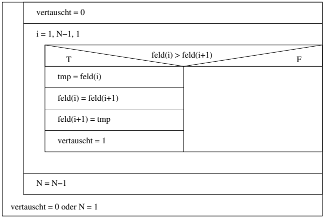
1	7	5	3	2
1	7	5	3	2
1	5	7	3	2
1	5	3	7	2
1	5	3	2	7
1	5	3	2	7
1	5	3	2	7
1	3	5	2	7
1	3	2	5	7
1	3	2	5	7
1	3	2	5	7
1	2	3	5	7
1	2	3	5	7
1	2	3	5	7

Vergleich ohne Vertauschung
Vergleich mit Vertauschung
Endpostion des Elementes

Bubblesort III – Regieanweisungen

1. Führe solange a) bis c) aus, bis keine Vertauschung innerhalb eines Durchganges mehr vorgenommen wurde
   a) Beginne an der linken Seite mit den ersten zwei Personen
   b) Vergleiche paarweise die benachbarten Personen und lasse die Plätze tauschen, wenn die rechte Person kleiner ist als die linke Person
   c) Laufe eine Person weniger weit, als du im letzten Durchgang gegangen bist

Bubblesort IV – Struktogramm



Durch die äußere ($\sum_1^{N-1}$) und innere ($\sum_{i=1}^{N-1}$) Schleife ergibt sich im schlechtesten Fall eine Anzahl von *FLOP*s (FLoating point OPerations) in der Größenordnung von $\mathcal{O}(N^2)$

Bubblesort V – Vor- und Nachteile

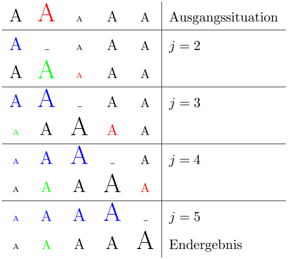
- Vorteile
  - einfach zu programmieren
- Nachteile
  - langsam
  - geringe Leistungsfähigkeit

Insertsort I – Grundprinzipen

Idee: Sortierung durch Einfügen an richtiger Stelle

- Feld wird elementweise von links nach rechts durchlaufen, beginnend bei zweitem Element
- linkes Feld ist schon sortiert
- Vergleich jedes Elements mit den linken Nachbarelementen
- Größere Elemente werden nach rechts verschoben
- Element wird in enstandene Lücke eingefügt
- alle Elemente links von dem betrachteten Element sind sortiert

Insertsort II – Erklärungsbeispiel



entnehmen, einfügen, betrachtete Unterliste

Insertsort III – Regieanweisungen

1. Für alle Personen von der Zweiten von links bis zur Rechten arbeite folgende Unterliste ab:
   1. Stelle die zunächst betrachtete Person gesondert ab
   2. Füge die betrachtete Person in die linke Seite ein, indem man alle größeren Personen (aus der linken Seite) einen Platz nach rechts gehen lässt und die betrachtete Person dann einordnet
   3. Erhöhe die einzuordnende Position um eins

Insertsort IV – Struktogramm

```
j = 2, N, 1
    tmp = feld(j)
    i = j – 1
    i >= 1 und feld(i) > tmp
        feld(i+1) = feld(i)
        i = i – 1
    feld(i+1) = tmp
```

Durch die äußere $(\sum_{j=2}^{N})$ und innere $(\sum_{1}^{i=j-1})$ Schleife ergibt sich im schlechtesten Fall eine Anzahl von *FLOP*s in der Größenordnung von $\mathcal{O}(N^2)$

Insertsort V – Vor- und Nachteile

- Vorteile
  - einfache Implementierung
  - minimaler Speicherverbrauch, da ortsfest (in-place)
  - stabiler Sortieralgorithmus
- Nachteile
  - bei steigenden Datenmengen, verringernde Effizienz

Quicksort I – Grundpinrizpen

- "divide and conquer"
- Wahl eines sog. Pivotelements (meist willkürlich das Element mit dem größten Index)
- Sortierung bzw. Aufteilung des Feldes mit Hilfe des Pivotelementes
- nach Tauschen mit Pivotelement ist das *alte* Pivotelement an richtiger Stelle
- rekursive Anwendung auf Teilfelder mit Änderung der Laufindizes
- Sortierung ist beendet, wenn alle Teilfelder sortiert sind

Quicksort II – Erklärungsbeispiel

```
4 2 1 7 3     1 2 4 7 3     1 2     7 4      4|7
    ⇓           k g         g k     g k      k g
4 2 1 7 3         ⇓
g       k     1 2|3|7 4     1 2     7 4      rot: groesser
                k g         k g     g k      blau: kleiner
4 2 1 7 3                    ⇓       ⇓       gruen: Pivot
g   k                       1|2     4 7
    ⇓                       k g     g k
1 2 4 7 3
g   k                               4 7
                                    k g
                                     ⇓
```

Quicksort III – Regieanweisungen Teil 1

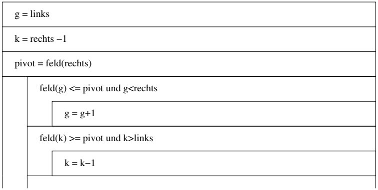
1. Person ganz rechts nimmt Blatt **P** (→ **P**ivotelement/Vergleichselement)
2. Person ganz links nimmt Blatt **g** (größer als)
3. Person links von **P** nimmt Blatt **k** (kleiner als)
4. Führe die Zeilen *a.* bis *c.* solange aus, bis Blatt **g** rechts von Blatt **k** oder genau gleich
   ...
5. Tausche Person mit **g** mit der Person mit **P** (Blätter mitnehmen), somit ist bekannt, dass Person mit **P** nun an der sortierten Stelle steht und links bzw. rechts von **P** sich neue Unterlisten bilden
6. Führe den Algorithmus (Punkt *1.* bis *6.*) für die entstandenen Unterlisten solange durch, bis die Anzahl der Listenelemente 1 beträgt

Quicksort IV – Regieanweisungen Teil 2

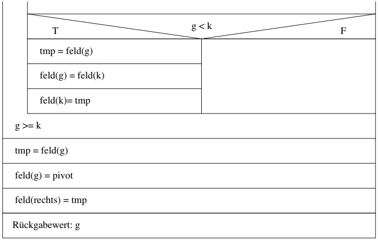
4. ...
   a. Lasse das Blatt **g** solange nach rechts laufen bis die Person (die gerade das **g** hält) größer ist als die Person mit dem **P** oder man bei der Person mit **P** angekommen ist
   (**g** kann auch bei der linken Person bleiben, falls diese größer ist)
   b. Lass das Blatt **k** solange nach links laufen bis die Person (die gerade das **k** hält) kleiner ist als die Person mit dem **P** oder das Blatt mit dem **k** am Anfang angekommen ist
   (**k** kann auch bei der rechten Person bleiben, falls diese kleiner ist)
   c. Falls die Person mit dem **g** links von der Person mit dem **k** steht, tausche die beiden Personen (aber lasse die Blätter an der Stelle)
5. ...

Quicksort V – Struktogramm

teile(feld, links, rechts)

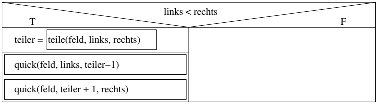


Quicksort VI – Struktogramm



Quicksort VII – Struktogramm

quick(feld, links, rechts)



Durch die Schleife in *teile()* und die rekursiven Aufrufe in *quick()* ergibt sich eine Fehlerordnung von durchschnittlich $\mathcal{O}(n \cdot \log(n))$

Quicksort VIII – Vor- und Nachteile

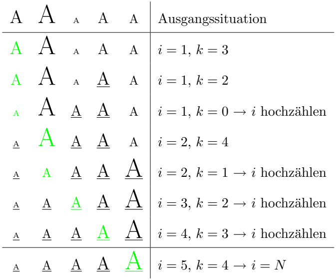
- Vorteile
  - schnell (vor allem bei großen Feldern)
  - effizient
  - einfach zu implementieren
- Nachteile
  - sehr störanfällig
  - langsam bei kleinen Feldern, Feldern aus Elementen mit oftmals gleichen Schlüsselwerten und vorsortierten Feldern
  - durch rekursiven Aufbau erhöhter Speicherbedarf für Stack, was zu Programmabstürzen führen kann

Swapsort I – Grundprinzipen

Idee: Vertauschen von Elementen, sodass getauschtes Element an endgültiger Position steht

- Feld wird elementweise von links nach rechts durchlaufen; beginnend bei erstem Element
- Elemente, die kleiner als das aktuelle Element sind, werden gezählt (Zähler k)
- aktuelles Element wird mit dem Element an der Stelle *k + 1* getauscht

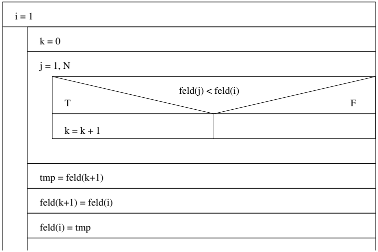
Swapsort II – Erklärungsbeispiel



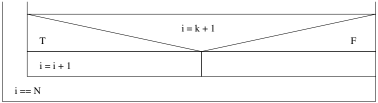
Swapsort III — Regieanweisungen

1. Setze i auf eins
2. Setze k auf null
   a. Zähle alle Personen die kleiner sind als die Person an Stelle i ($\rightarrow k$ pro kleinerer Person um eins hochzählen)
   b. Tausche die Person an der Stelle i mit der Person an der Stelle $k{+}1$
3. Falls i den gleichen Wert hat wie $k{+}1$, zähle i um eins hoch
4. Führe die Punkte 2. und 3. solange durch, bis i der Anzahl der Personen entspricht

Swapsort IV — Struktogramm



Swapsort V — Struktogramm



Durch die äußere und innere ($\sum_N^{j=1}$) Schleife ergibt sich immer eine Anzahl der FLOPs in der Größenordnung von $\mathcal{O}(n^2)$

Swapsort VI — Vor- und Nachteile

- Vorteile
  - einfache Implementierung
  - minimaler Speicherverbrauch, da ortsfest (in-place)
  - stabiler Sortieralgorithmus
- Nachteile
  - jedes Element darf nur einmal vorkommen (sonst keine Terminierung)

Gegenüberstellung

Sortieralgorithmus	*Best-Case*	*Average-Case*	*Worst-Case*
Bubblesort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Insertsort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Quicksort	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n^2)$
Swapsort			$\mathcal{O}(n^2)$

Weiterführendes

Weiterführende Informationen findet ihr unter:

- Übersicht diverser Sortieralgorithmen: http://www.sorting-algorithms.com/
- Veranschaulichung mit Hilfe von Quelltext: http://www.bluffton.edu/ nesterd/java/SortingDemo.html