

# Sortier-Algorithmen

Wednesday, January 5, 2022 11:19 PM

## Sortieren

Gegeben:

Datenmenge, der eine Ordnungsrelation zugrunde liegt.

Ziel:

Sortieren der Datenelemente, sodass diese aufsteigend geordnet sind

Geeignete Datenstruktur: Ein Array A mit n natürlichen Zahlen.

### I. Einfache Sortierverfahren

#### → SelectionSort: Sortieren durch Auswählen (hier: „Minsort“)

Beispiel: n = 7

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Ausgangssituation	8	2	1	7	9	5	3
Nach dem 1. Durchgang	1	2	8	7	9	5	3
Nach dem 2. Durchgang	1	2	8	7	9	5	3
Nach dem 3. Durchgang	1	2	3	7	9	5	8
Nach dem 4. Durchgang	1	2	3	5	9	7	8
Nach dem 5. Durchgang	1	2	3	5	7	9	8
Nach dem 6. Durchgang	1	2	3	5	7	8	9

#### Vorgehensweise

Suche das kleinste Element im unsortierten Array und tausche es mit dem ersten Element des Arrays.

Betrachte anschließend nur noch das Array ohne das erste Element und gehe genauso vor.

Wiederhole diesen Vorgang, bis das zu betrachtende Array nur noch aus einem Element besteht.

#### Implementierung

```
public void selectionSort(){
    for (int i = 0; i < n-1; i++){
        int minpos = i;
        for (int j = i+1; j < n; j++){
            if (A[j] < A[minpos]){
                minpos = j;
            }
        }
        int temp = A[i];
        A[i] = A[minpos];
        A[minpos] = temp;
    }
}
```

#### Laufzeitverhalten

worst case:  $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$

best case:  $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$

## →InsertionSort: Sortieren durch Einfügen

Beispiel: n = 7

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Ausgangssituation	8	2	1	7	9	5	3
Nach dem 1. Durchgang	2	8	1	7	9	5	3
Nach dem 2. Durchgang	1	2	8	7	9	5	3
Nach dem 3. Durchgang	1	2	7	8	9	5	3
Nach dem 4. Durchgang	1	2	7	8	9	5	3
Nach dem 5. Durchgang	1	2	5	7	8	9	3
Nach dem 6. Durchgang	1	2	3	5	7	8	9

### Vorgehensweise

Man nimmt sich das zweite Element und fügt es in dem sortierten Teil des Arrays, der zu Beginn nur aus dem ersten Element besteht, an die passende Stelle ein. Als nächstes wird das dritte Element in den sortierten Teil eingefügt usw., bis das letzte Element eingefügt wird.

Mit jedem Schritt wird der sortierte Teil um 1 Element größer. Zu Beginn besteht er nur aus einem Element und am Ende aus allen Elementen des Arrays.

### Implementierung

```
public void insertionSort(){
    for (int i = 1; i < n; i++){
        int wert = A[i];
        int j = i;
        while(j > 0 && A[j-1] > wert){
            A[j] = A[j-1];
            j--;
        }
        A[j] = wert;
    }
}
```

### Laufzeitverhalten

worst case: Das Feld ist absteigend sortiert.  
 $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$

best case: Das Feld ist bereits aufsteigend sortiert.  
 $T(n) = n - 1$

## →BubbleSort: Sortieren durch Austauschen

Beispiel: n = 7

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	
Ausgangssituation	8 2	2 8	1 8	7 7	9 8	5 5	3 9	
Nach dem 1. Durchgang	2 1	1 2	7 7	8 8	5 5	3 3	9 9	
Nach dem 2. Durchgang	1 5	2 7	7 5	5 7	3 3	8 8	9 9	
Nach dem 3. Durchgang	1 5	2 5	5 3	3 7	7 8	8 9	9 9	
Nach dem 4. Durchgang	1 5	2 3	3 5	5 7	7 8	8 9	9 9	
Nach dem 5. Durchgang	1 1	2 2	3 3	5 5	7 7	8 8	9 9	Keine Vertauschung!

### Vorgehensweise

Das Array wird von vorne nach hinten durchlaufen, wobei jeweils zwei benachbarte Elemente miteinander verglichen werden. Diese zwei Elemente tauschen ihren Platz, wenn das vordere größer als das hintere ist.

Am Ende des ersten Durchlaufs steht das größte Element an der letzten Stelle, wodurch im zweiten Durchlauf das letzte Element ausgelassen werden kann. Das zu durchlaufende Feld verkleinert sich so in jedem Durchlauf um ein Element.

Das Durchlaufen mit entsprechendem Vertauschen wird solange wiederholt, bis nur noch 1 Element zu betrachten ist.

### Implementierung

```
public void bubbleSort(){
    int grenze = n-1;
    while (grenze > 0){
        for(int i=0; i < grenze; i++){
            if(A[i] > A[i+1]){
                int temp = A[i];
                A[i] = A[i+1];
                A[i+1] = temp;
            }
        }
        grenze--;
    }
}
```

### Laufzeitverhalten

worst case: Das Feld ist absteigend sortiert.  
 $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$

best case: Das Feld ist bereits aufsteigend sortiert.  
 $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$

### Bemerkungen

- 1) Ein großes Element zu Beginn des Arrays wandert nach wenigen Durchläufen nach hinten. Ein kleines Element am Ende kann pro Durchlauf nur um höchstens eine Position verrücken und wandert daher sehr langsam nach vorne.
- 2) Der Algorithmus kann beschleunigt werden, wenn man nur solange durchläuft, bis keine Vertauschung mehr stattgefunden hat.

### Implementierung

```
public void bubbleSort(){  
    boolean getauscht = true;  
    int grenze = n-1;  
    while((getauscht) && (grenze > 0)){  
        getauscht = false;  
        for(int i=0; i < grenze; i++){  
            if(A[i] > A[i+1]){  
                int temp = A[i];  
                A[i] = A[i+1];  
                A[i+1] = temp;  
                getauscht = true;  
            }  
        }  
        grenze--;  
    }  
}
```

### Laufzeitverhalten

- worst case: Das Feld ist absteigend sortiert.  
 $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$
- best case: Das Feld ist bereits aufsteigend sortiert.  
 $T(n) = n - 1$