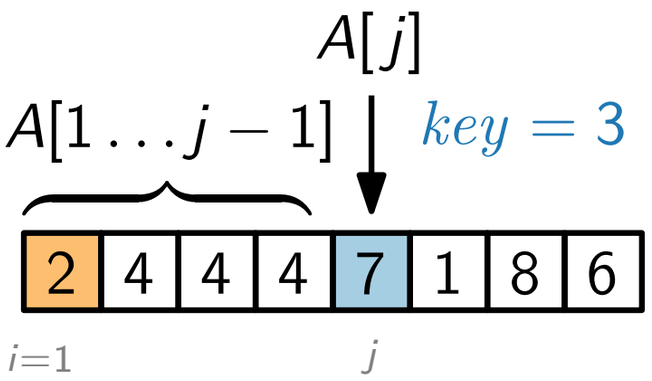
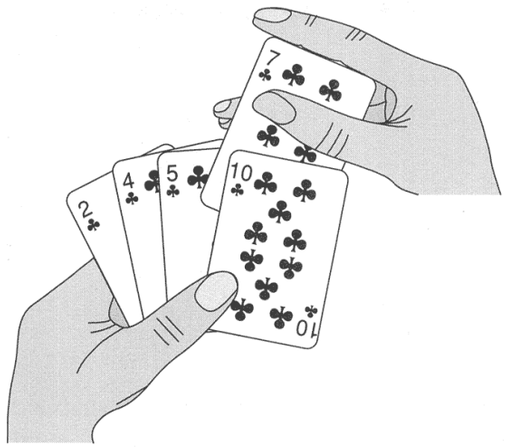
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Algorithmen und Datenstrukturen



Vorlesung 1:
Sortieren



Alexander Wolff

Wintersemester 2025

Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Umordnung

Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Eingabe

Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Umordnung

Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Eingabe

Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Umordnung

Ausgabe

Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Eingabe

Umordnung

Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Ausgabe

Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Eingabe

Algorithmus

Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Umordnung

Ausgabe

Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Eingabe

Algorithmus

Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Umordnung

Ausgabe

Beachte. Computerinterne Zahlendarstellung hier unwichtig!

Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Eingabe

Algorithmus

Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Umordnung

Ausgabe

Beachte. Computerinterne Zahlendarstellung hier unwichtig!

Wichtig:

Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Eingabe

Algorithmus

Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Umordnung

Ausgabe

Beachte. Computerinterne Zahlendarstellung hier unwichtig!

Wichtig:
- Je zwei Zahlen lassen sich vergleichen.

Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Eingabe

Algorithmus

Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Umordnung

Ausgabe

Beachte. Computerinterne Zahlendarstellung hier unwichtig!

Wichtig:
- Je zwei Zahlen lassen sich vergleichen.
- Ein Vergleich dauert **konstante Zeit**, d.h. die Dauer ist unabhängig von n.

Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Eingabe

Algorithmus

Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

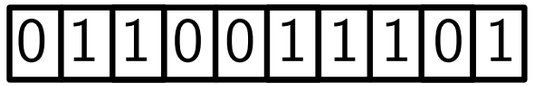
Umordnung

Ausgabe

Beachte. Computerinterne Zahlendarstellung hier unwichtig!

Wichtig:
- Je zwei Zahlen lassen sich vergleichen.
- Ein Vergleich dauert **konstante Zeit**, d.h. die Dauer ist unabhängig von n.

Noch was:



Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Eingabe

Algorithmus

Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Umordnung

Ausgabe

Beachte. Computerinterne Zahlendarstellung hier unwichtig!

Wichtig:
- Je zwei Zahlen lassen sich vergleichen.
- Ein Vergleich dauert **konstante Zeit**, d.h. die Dauer ist unabhängig von n.

Noch was:

Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Eingabe

Algorithmus

Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

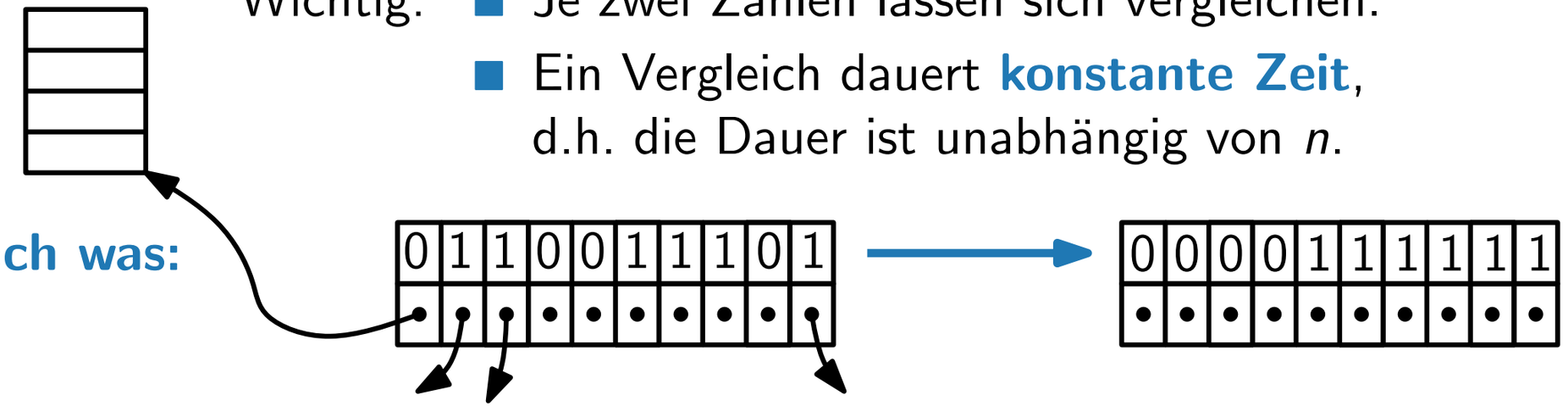
Umordnung

Ausgabe

Beachte. Computerinterne Zahlendarstellung hier unwichtig!

Wichtig:
- Je zwei Zahlen lassen sich vergleichen.
- Ein Vergleich dauert **konstante Zeit**, d.h. die Dauer ist unabhängig von n.

Noch was:



Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Eingabe

Algorithmus

Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

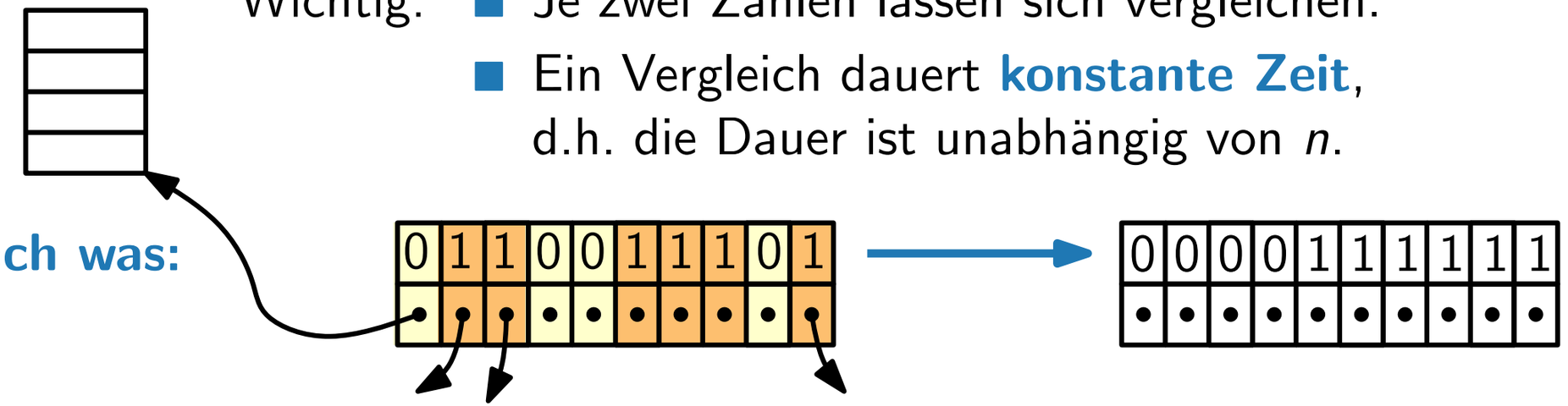
Umordnung

Ausgabe

Beachte. Computerinterne Zahlendarstellung hier unwichtig!

Wichtig:
- Je zwei Zahlen lassen sich vergleichen.
- Ein Vergleich dauert **konstante Zeit**, d.h. die Dauer ist unabhängig von n.

Noch was:



Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Eingabe

↓ **Algorithmus**

Umordnung

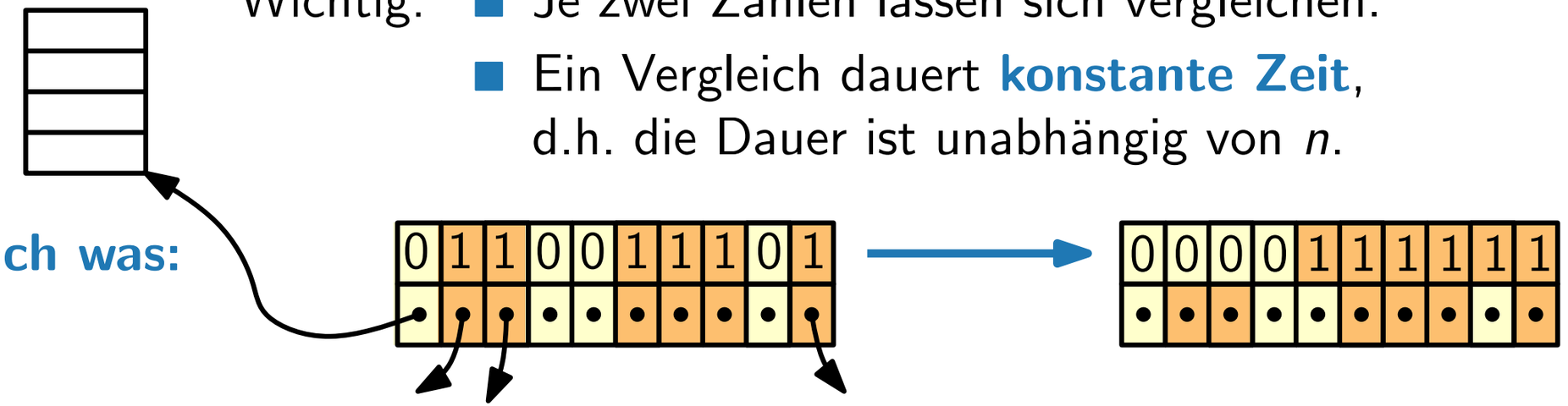
Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Ausgabe

Beachte. Computerinterne Zahlendarstellung hier unwichtig!

Wichtig:
- Je zwei Zahlen lassen sich vergleichen.
- Ein Vergleich dauert **konstante Zeit**, d.h. die Dauer ist unabhängig von n.

Noch was:



Das Problem

Gegeben. eine Folge $A = \langle a_1, a_2, \ldots, a_n \rangle$ von n Zahlen

Eingabe

↓ **Algorithmus**

Umordnung

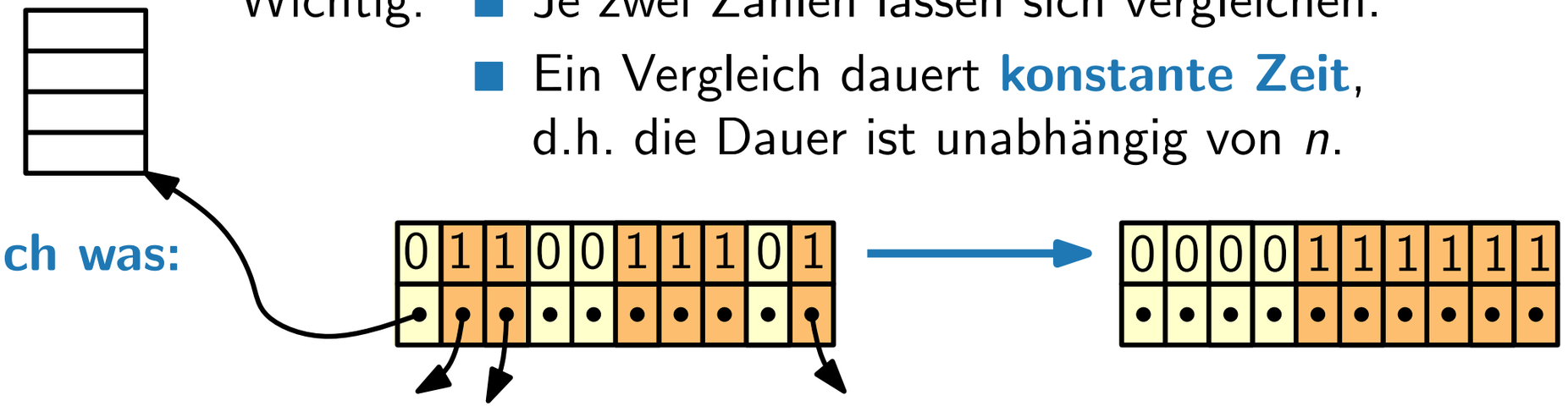
Gesucht. eine **Permutation** $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Ausgabe

Beachte. Computerinterne Zahlendarstellung hier unwichtig!

Wichtig:
- Je zwei Zahlen lassen sich vergleichen.
- Ein Vergleich dauert **konstante Zeit**, d.h. die Dauer ist unabhängig von n.

Noch was:



Frage an Alle

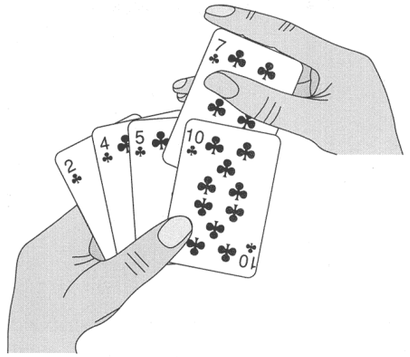
Frage an Alle

Bild von Kundan Kumar

Wie sortieren Sie Karten?

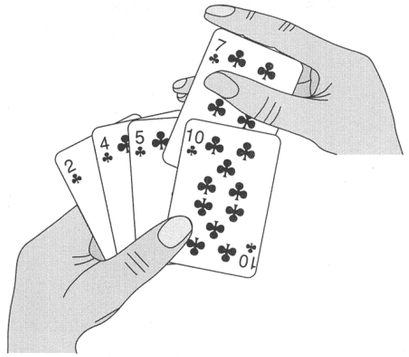
Mögliche Lösungen

InsertionSort

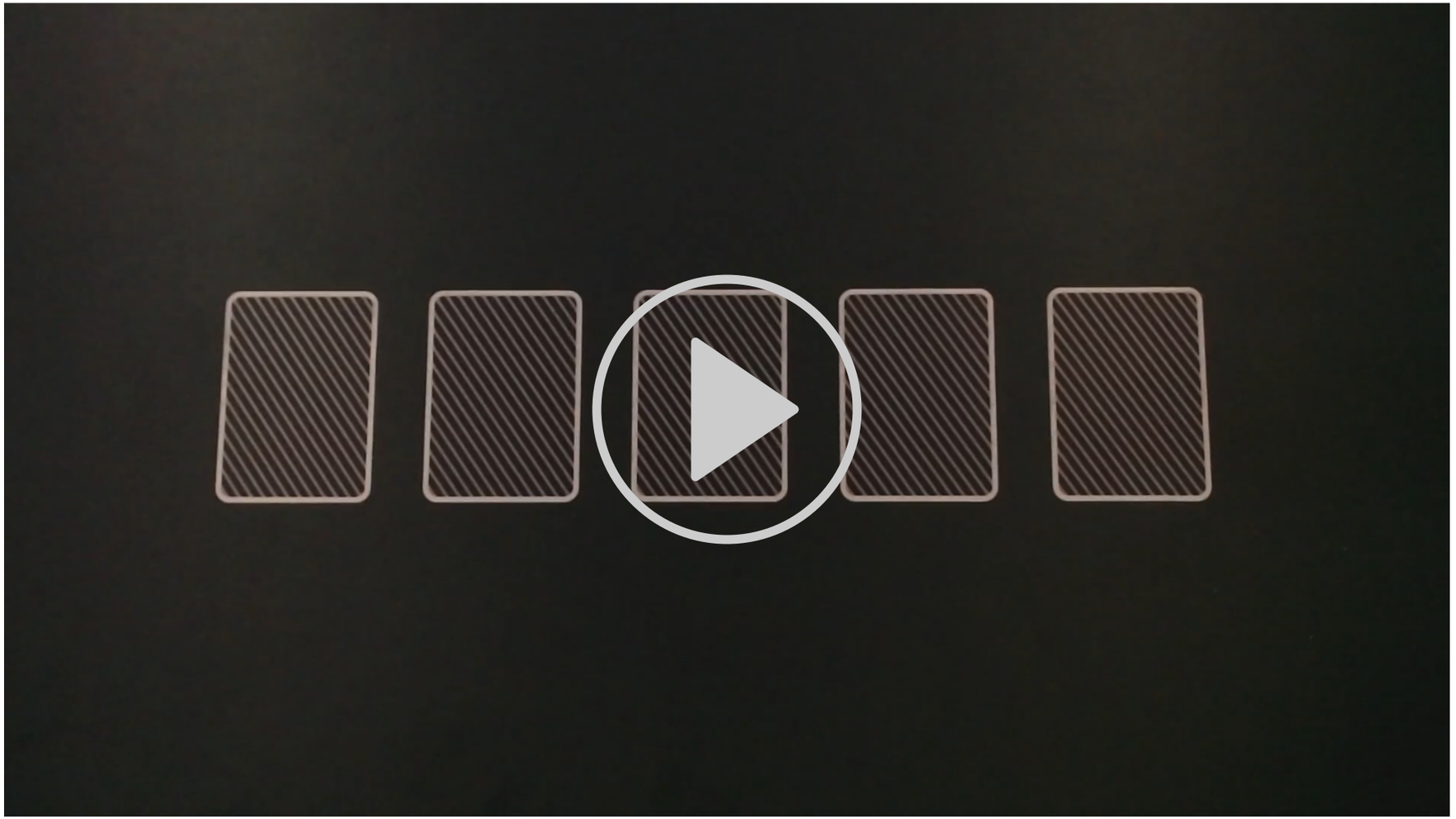


Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen



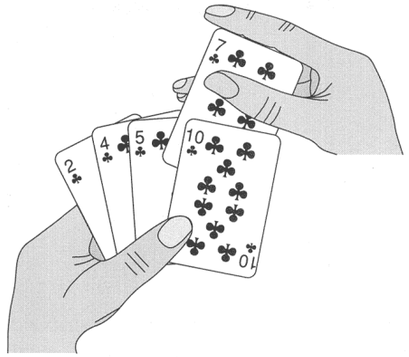
InsertionSort



Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen

InsertionSort

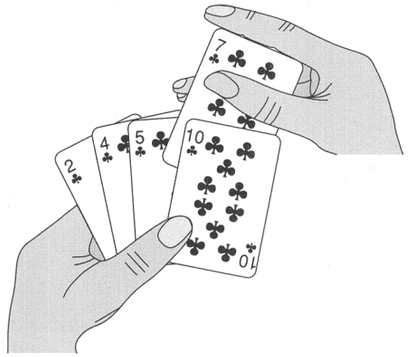


- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen

InsertionSort

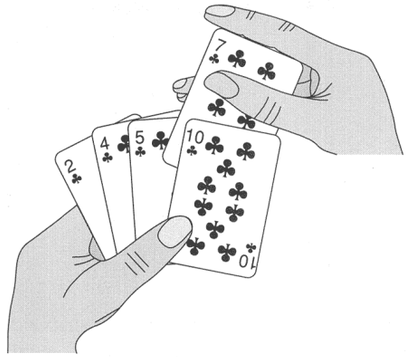


- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand nimmt Karten nacheinander auf und steckt sie (von rechts kommend) an die richtige Position zwischen die Karten in der linken Hand.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen

InsertionSort

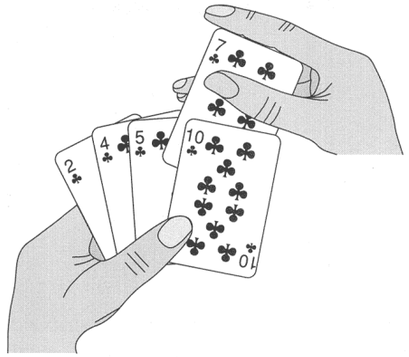


- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand nimmt Karten nacheinander auf und steckt sie (von rechts kommend) an die richtige Position zwischen die Karten in der linken Hand.
- Linke Hand hält **immer eine sortierte Reihenfolge** aufrecht.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

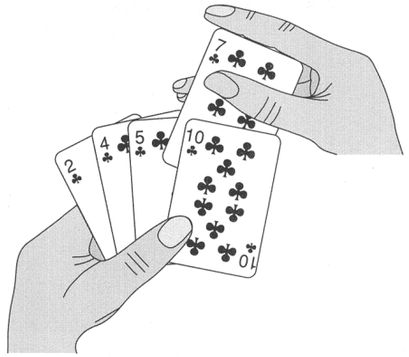
Mögliche Lösungen

SelectionSort

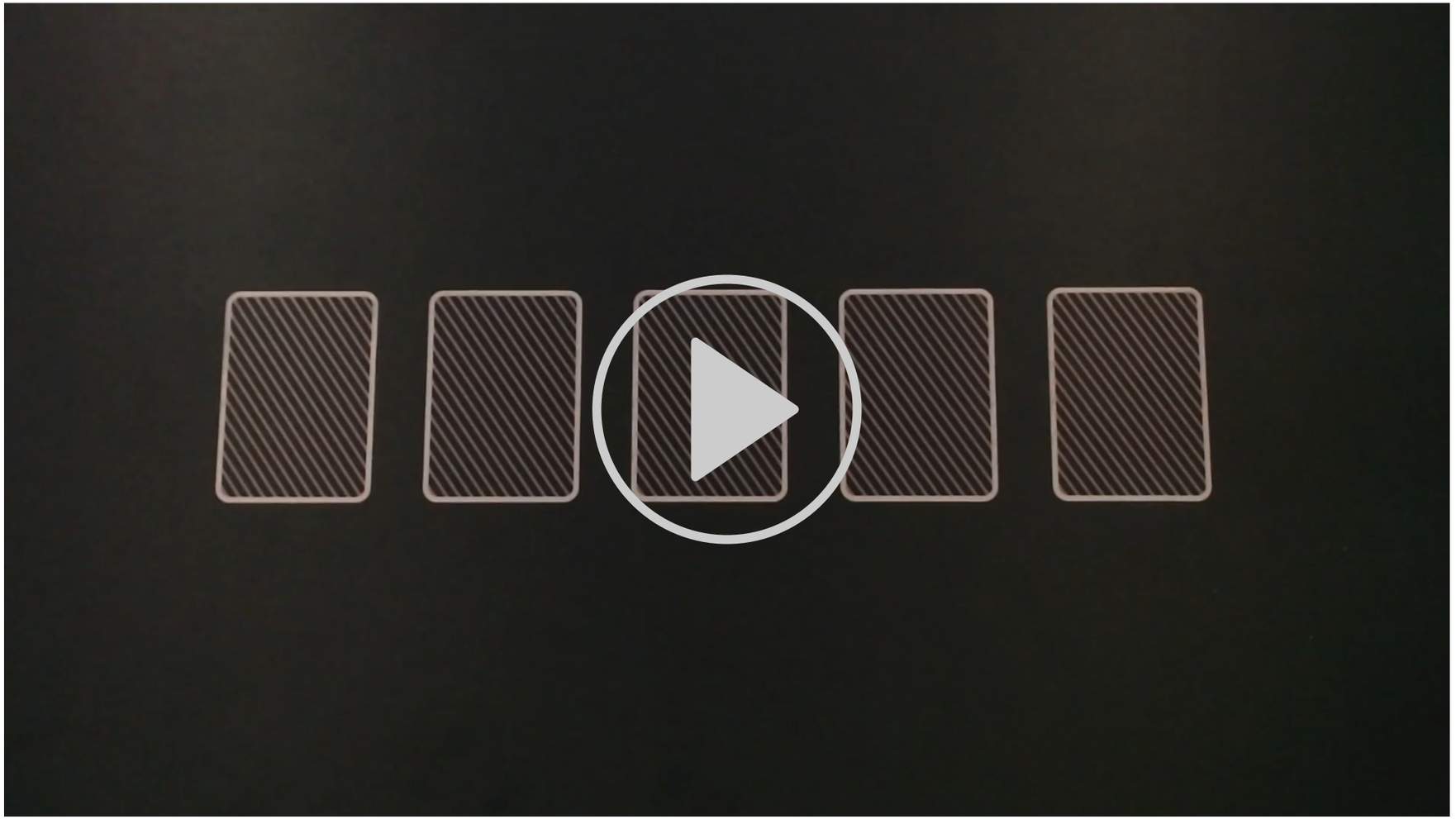


Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen



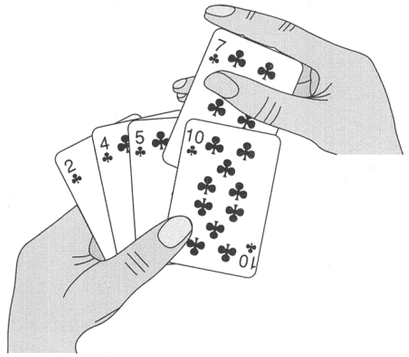
SelectionSort



Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen

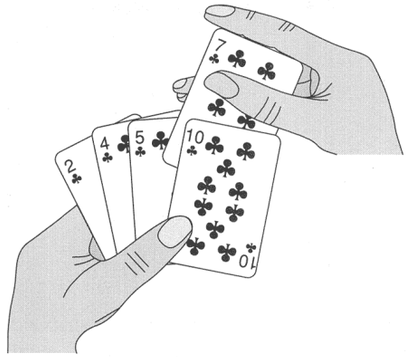
SelectionSort



- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen

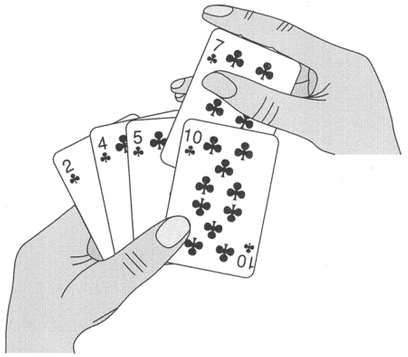


SelectionSort

- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand sucht kleinste Karte und nimmt sie in die linke Hand.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen



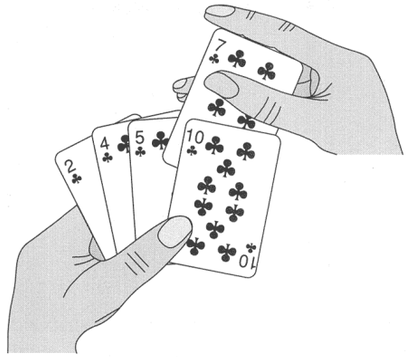
SelectionSort

- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand sucht kleinste Karte und nimmt sie in die linke Hand.
- Linke Hand hält **immer eine sortierte Reihenfolge** aufrecht.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

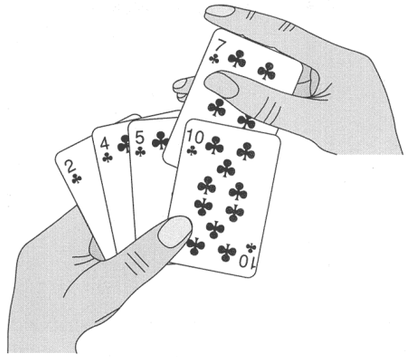
Mögliche Lösungen

BubbleSort

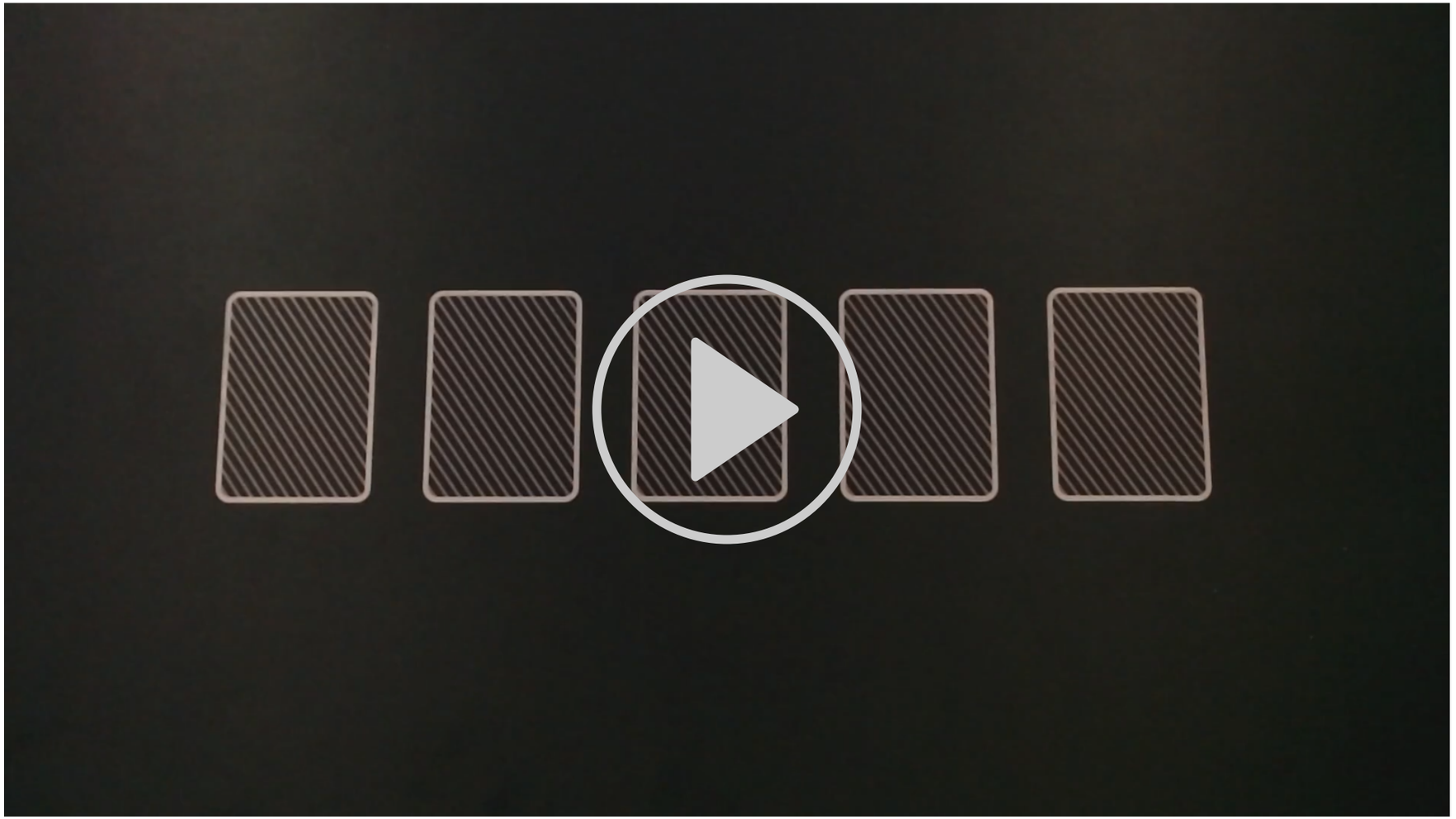


Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen



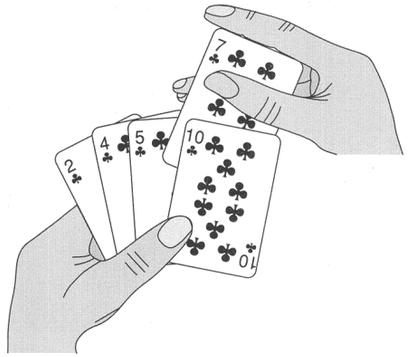
BubbleSort



Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen

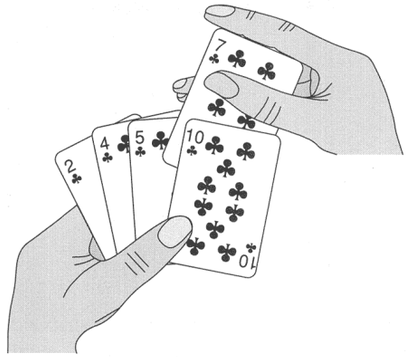
BubbleSort



- Linke Hand hält **alle Karten**.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen

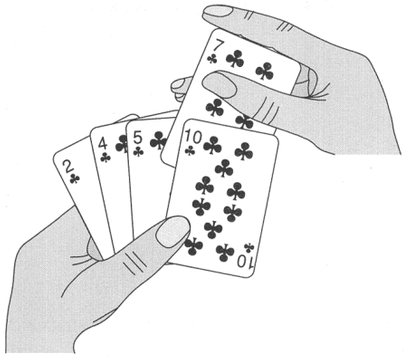


BubbleSort

- Linke Hand hält **alle Karten**.
- Vergleiche erste mit zweiter Karte. Wenn erste Karte größer, tausche die Karten.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen



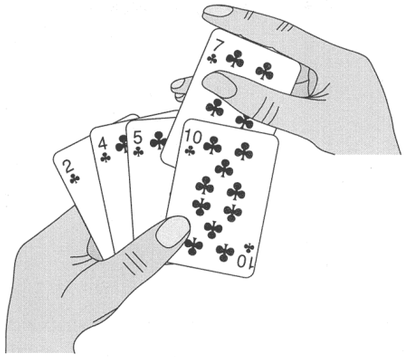
BubbleSort

- Linke Hand hält **alle Karten**.
- Vergleiche erste mit zweiter Karte. Wenn erste Karte größer, tausche die Karten.
- Vergleiche nun zweite mit dritter Karte, usw...

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen

BubbleSort

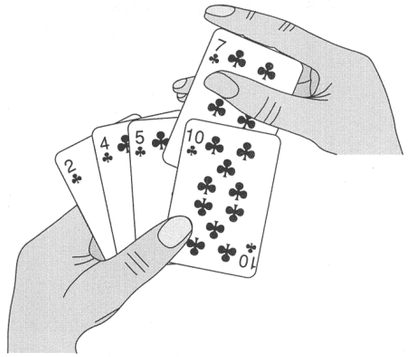


- Linke Hand hält **alle Karten**.
- Vergleiche erste mit zweiter Karte. Wenn erste Karte größer, tausche die Karten.
- Vergleiche nun zweite mit dritter Karte, usw...
- Beginne wieder von vorne und wiederhole, **bis Karten in linker Hand sortiert**.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

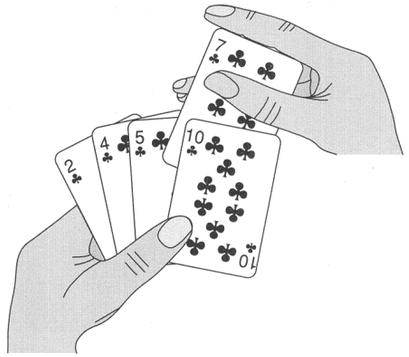
Mögliche Lösungen

BogoSort

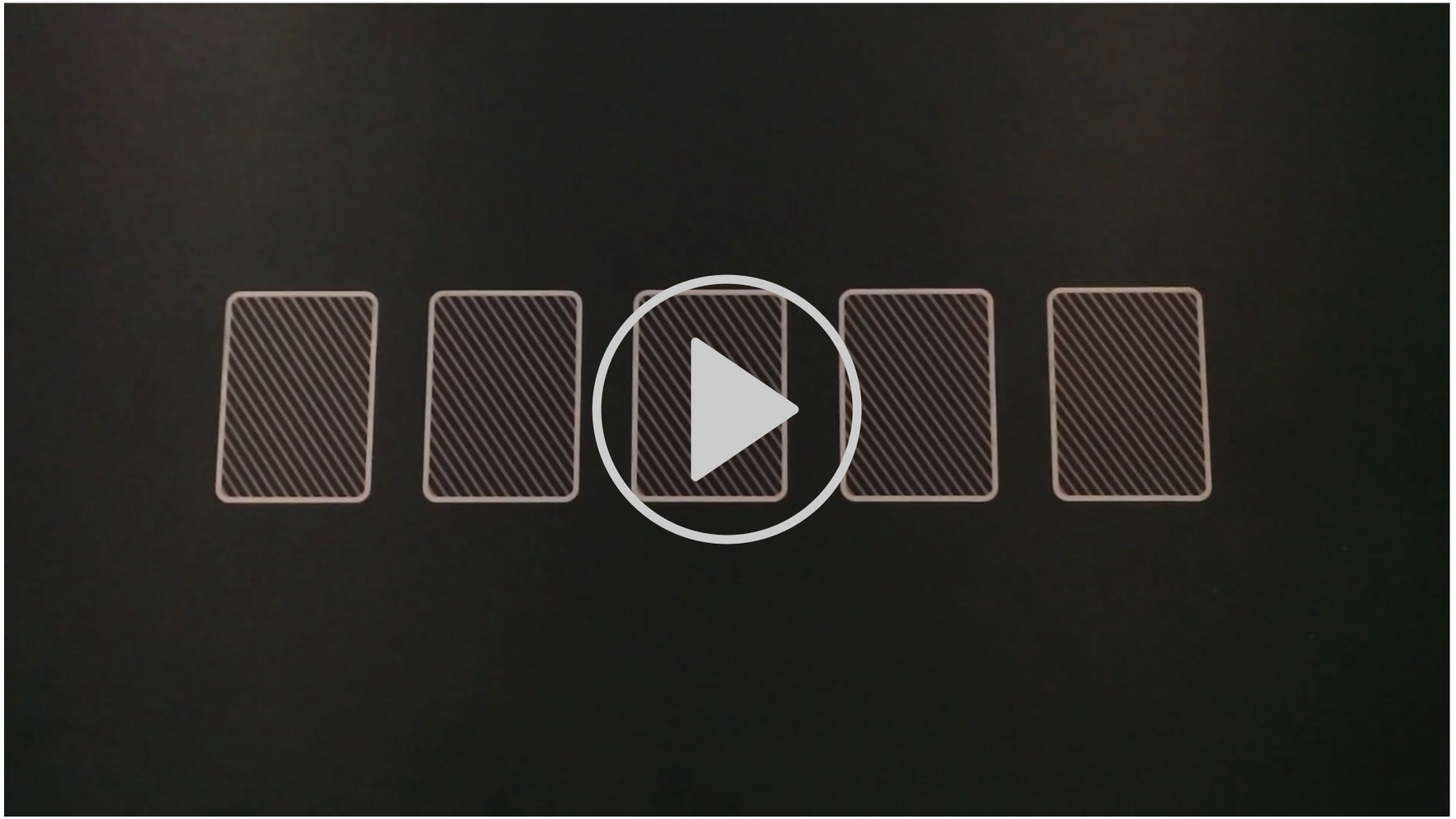


Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen



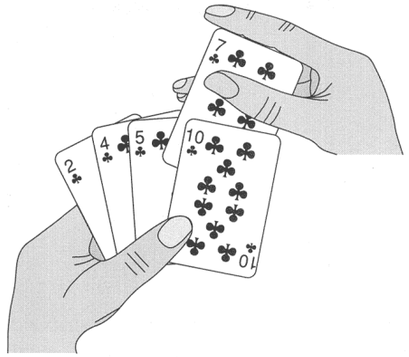
BogoSort



Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen

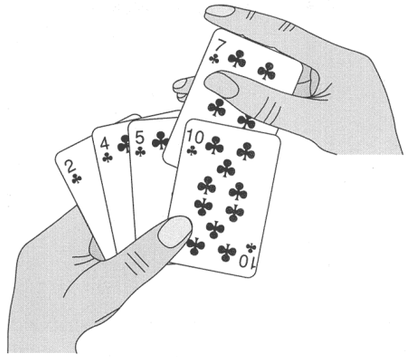
BogoSort



- Linke Hand hält **alle Karten**.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen



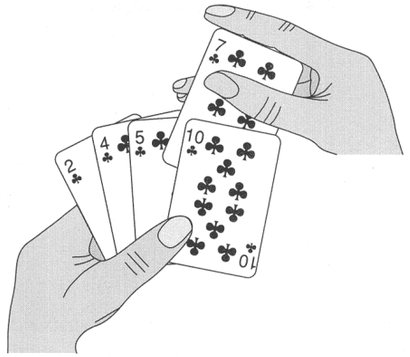
BogoSort

- Linke Hand hält **alle Karten**.
- Wirf Karten in die Luft und sammle sie in zufälliger Reihenfolge auf.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen

BogoSort

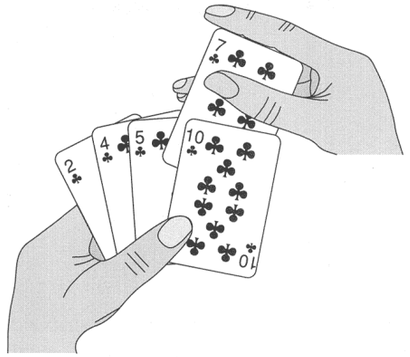


- Linke Hand hält **alle Karten**.
- Wirf Karten in die Luft und sammle sie in zufälliger Reihenfolge auf.
- Wenn Karten nicht sortiert, beginne wieder von vorne.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen

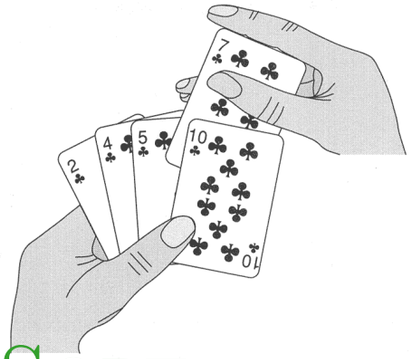
BogoSort



- Linke Hand hält **alle Karten**.
- Wirf Karten in die Luft und sammle sie in zufälliger Reihenfolge auf.
- Wenn Karten nicht sortiert, beginne wieder von vorne.
- Beginne wieder von vorne und wiederhole, **bis Karten in linker Hand sortiert**.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen



InsertionSort

- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand nimmt Karten nacheinander auf und steckt sie (von rechts kommend) an die richtige Position zwischen die Karten in der linken Hand.
- Linke Hand hält **immer eine sortierte Reihenfolge** aufrecht.

SelectionSort

- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand such kleinste Karte und nimmt sie in die linke Hand
- Linke Hand hält **immer eine sortierte Reihenfolge** aufrecht.

BubbleSort

- Linke Hand hält **alle Karten**
- Vergleiche erste mit zweiter Karte. Wenn erste Karte größer, tausche die Karten.
- Vergleiche nun zweite mit dritter Karte, usw...
- Beginne wieder von vorne und wiederhole, **bis Karten in linker Hand sortiert**.

BogoSort

- Linke Hand hält **alle Karten**
- Wirf Karten in die Luft und sammle sie in zufälliger Reihenfolge auf.
- Wenn Karten nicht sortiert, beginne wieder von vorne.
- Beginne wieder von vorne und wiederhole, **bis Karten in linker Hand sortiert**.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Mögliche Lösungen

InsertionSort

- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand nimmt Karten nacheinander auf und steckt sie (von rechts kommend) an die richtige Position zwischen die Karten in der linken Hand.
- Linke Hand hält **immer eine sortierte Reihenfolge** aufrecht.

SelectionSort

- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand such kleinste Karte und nimmt sie in die linke Hand
- Linke Hand hält **immer eine sortierte Reihenfolge** aufrecht.

BubbleSort

- Linke Hand hält **alle Karten**
- Vergleiche erste mit zweiter Karte. Wenn erste Karte größer, tausche die Karten.
- Vergleiche nun zweite mit dritter Karte, usw...
- Beginne wieder von vorne und wiederhole, **bis Karten in linker Hand sortiert**.

BogoSort

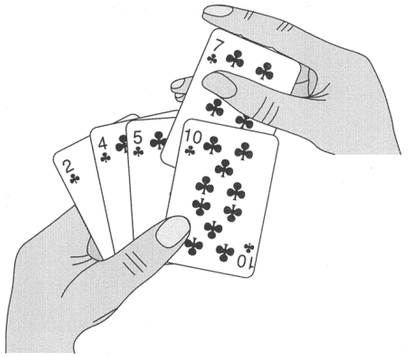
- Linke Hand hält **alle Karten**
- Wirf Karten in die Luft und sammle sie in zufälliger Reihenfolge auf.
- Wenn Karten nicht sortiert, beginne wieder von vorne.
- Beginne wieder von vorne und wiederhole, **bis Karten in linker Hand sortiert**.

Inkrementelle Algorithmen

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Inkrementeller Algorithmus

InsertionSort

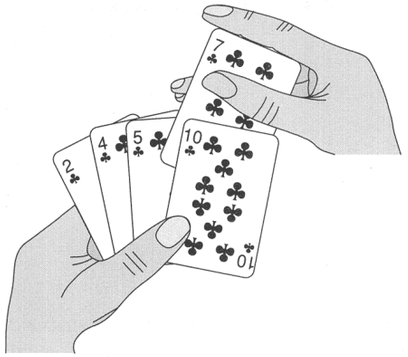


- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand nimmt Karten nacheinander auf und steckt sie (von rechts kommend) an die richtige Position zwischen die Karten in der linken Hand.
- Linke Hand hält **immer eine sortierte Reihenfolge** aufrecht.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Inkrementeller Algorithmus

InsertionSort

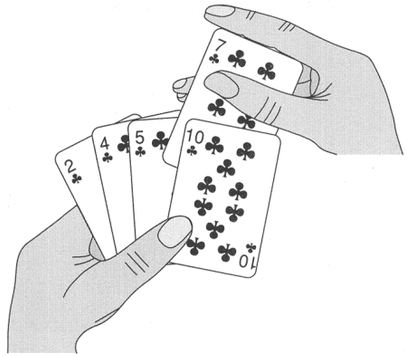


- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand nimmt Karten nacheinander auf und steckt sie (von rechts kommend) an die richtige Position zwischen die Karten in der linken Hand.
- Linke Hand hält **immer eine sortierte Reihenfolge** aufrecht.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Inkrementeller Algorithmus

InsertionSort

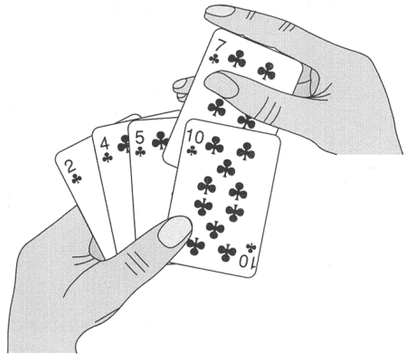


- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand nimmt Karten nacheinander auf und steckt sie (von rechts kommend) an die richtige Position zwischen die Karten in der linken Hand.
- Linke Hand hält **immer eine sortierte Reihenfolge** aufrecht.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Inkrementeller Algorithmus

InsertionSort

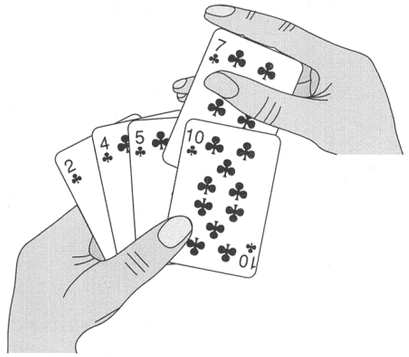


- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand nimmt Karten nacheinander auf und steckt sie (von rechts kommend) an die richtige Position zwischen die Karten in der linken Hand.
- Linke Hand hält **immer eine sortierte Reihenfolge** aufrecht.

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Inkrementeller Algorithmus

InsertionSort



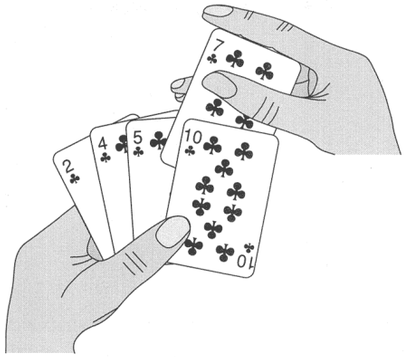
- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand nimmt Karten nacheinander auf und steckt sie (von rechts kommend) an die richtige Position zwischen die Karten in der linken Hand.
- Linke Hand hält **immer eine sortierte Reihenfolge** aufrecht.

Invariante

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Inkrementeller Algorithmus

InsertionSort



- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand nimmt Karten nacheinander auf und steckt sie (von rechts kommend) an die richtige Position zwischen die Karten in der linken Hand.
- Linke Hand hält **immer eine sortierte Reihenfolge** aufrecht.

Invariante

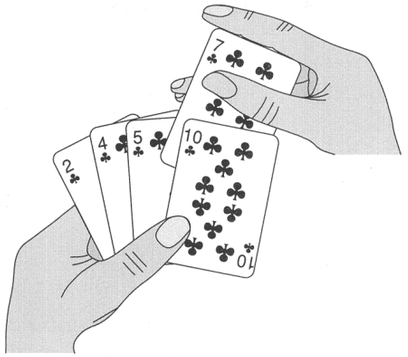
⇓

Korrektheit

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Inkrementeller Algorithmus

InsertionSort



- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand nimmt Karten nacheinander auf und steckt sie (von rechts kommend) an die richtige Position zwischen die Karten in der linken Hand.
- Linke Hand hält **immer eine sortierte Reihenfolge** aufrecht.

Invariante

$\Downarrow$

Korrektheit: am Ende sind alle Karten in der linken Hand – und zwar **sortiert!**

Zeichnung aus: „Introduction to Algorithms“ [Cormen et al., MIT Press, 2. Aufl., 2001]

Inkrementteller Algorithmus

// In Pseudocode

Inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of ... A)

Inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of ... A)

Name des Alg.

Inkrementeller Algorithmus

// In Pseudocode

$\underbrace{\textsc{IncrementalAlg}}_{\text{Name des Alg.}}(\underbrace{\text{array of } \ldots\ A}_{\text{Eingabe}})$

Inkrementeller Algorithmus

// In Pseudocode

Typ der Eingabe (hier ein Feld von ...)

IncrementalAlg(array of ... A)

Name des Alg. Eingabe

Inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of ... A)

Typ der Eingabe (hier ein Feld von ...)

Variable

Name des Alg.

Eingabe

Inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of ... A)

Inkrementeller Algorithmus

// In Pseudocode

```
IncrementalAlg( array of ... A )
    berechne Lösung für A[1]
```

Inkrementeller Algorithmus

// In Pseudocode

```
IncrementalAlg( array of ... A )
    berechne Lösung für A[1]          // Initialisierung
```

Inkrementeller Algorithmus

// In Pseudocode

```
IncrementalAlg( array of ... A )
    berechne Lösung für A[1]                    // Initialisierung
    for j = 2 to A.length do
        |
        |
        |_
```

Inkrementeller Algorithmus

// In Pseudocode

```
IncrementalAlg( array of ... A )
    berechne Lösung für A[1]                    // Initialisierung
    for j = 2 to A.length do                    // Schleifenkopf
        |
        |
        |_
```

Inkrementeller Algorithmus

// In Pseudocode

```
INCREMENTALALG( array of ... A )
    berechne Lösung für A[1]          // Initialisierung
    for j = 2 to A.length do          // Schleifenkopf
```

Zuweisungoperator

Inkrementeller Algorithmus

// In Pseudocode

```
IncrementalAlg( array of ... A )
    berechne Lösung für A[1]          // Initialisierung
    for j = 2 to A.length do          // Schleifenkopf
```

Zuweisungoperator

- in manchen Sprachen $j := 2$

Inkrementeller Algorithmus

// In Pseudocode

```
IncrementalAlg( array of ... A )
    berechne Lösung für A[1]                    // Initialisierung
    for j = 2 to A.length do                    // Schleifenkopf
        |
        |
```

Zuweisungoperator

- in manchen Sprachen $j := 2$
- in manchen Büchern $j \leftarrow 2$

Inkrementeller Algorithmus

// In Pseudocode

```
IncrementalAlg( array of ... A )
    berechne Lösung für A[1]                    // Initialisierung
    for j = 2 to A.length do                    // Schleifenkopf
```

Zuweisungoperator

- in manchen Sprachen $j := 2$
- in manchen Büchern $j \leftarrow 2$
- in Java $j = 2$

Inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of ... A)

berechne Lösung für Anzahl der Elemente des Feldes A

for $j = 2$ **to** $A.length$ **do** // Schleifenkopf

Zuweisungoperator

- in manchen Sprachen $j := 2$
- in manchen Büchern $j \leftarrow 2$
- in Java $j = 2$

Inkrementeller Algorithmus

// In Pseudocode

```
IncrementalAlg( array of ... A )
    berechne Lösung für A[1]                    // Initialisierung
    for j = 2 to A.length do                    // Schleifenkopf
        // Schleifenkörper; wird (A.length − 1)-mal durchlaufen
```

Inkrementeller Algorithmus

// In Pseudocode

```
IncrementalAlg( array of ... A )
    berechne Lösung für A[1]                    // Initialisierung
    for j = 2 to A.length do                    // Schleifenkopf
        // Schleifenkörper; wird (A.length − 1)-mal durchlaufen
        berechne Lösung für A[1 ... j] mithilfe der für A[1 ... j − 1]
```

Inkrementeller Algorithmus

// In Pseudocode

```
IncrementalAlg( array of ... A )
    berechne Lösung für A[1]                    // Initialisierung
    for j = 2 to A.length do                    // Schleifenkopf
        // Schleifenkörper; wird (A.length − 1)-mal durchlaufen
        berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]
```

Teilarray von A mit den Elementen $A[1], A[2], \ldots, A[j]$

Inkrementeller Algorithmus

// In Pseudocode

```
IncrementalAlg( array of ... A )
    berechne Lösung für A[1]                    // Initialisierung
    for j = 2 to A.length do                    // Schleifenkopf
        // Schleifenkörper; wird (A.length − 1)-mal durchlaufen
        berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]

    return Lösung
```

Teilarray von A mit den Elementen $A[1], A[2], \ldots, A[j]$

Inkrementeller Algorithmus

// In Pseudocode

```
IncrementalAlg( array of ... A )
    berechne Lösung für A[1]                    // Initialisierung
    for j = 2 to A.length do                    // Schleifenkopf
        // Schleifenkörper; wird (A.length − 1)-mal durchlaufen
        berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]
    return Lösung                               // Ergebnisrückgabe
```

Teilarray von A mit den Elementen $A[1], A[2], \ldots, A[j]$

Inkrementeller Algorithmus

```
IncrementalAlg( array of ... A )
    berechne Lösung für A[1]
    for j = 2 to A.length do
        |
        |
        |  berechne Lösung für A[1 ... j] mithilfe der für A[1 ... j − 1]
    return Lösung
```

Inkrementeller Algorithmus

InsertionSort

~~IncrementalAlg~~(array of $\ldots$ A)

berechne Lösung für $A[1]$

for $j = 2$ **to** $A.length$ **do**

berechne Lösung für $A[1 \ldots j]$ mithilfe der für $A[1 \ldots j-1]$

return Lösung

Inkrementeller Algorithmus

InsertionSort

~~IncrementalAlg~~(array of int A)

berechne Lösung für $A[1]$

for $j = 2$ **to** $A.length$ **do**

berechne Lösung für $A[1 \ldots j]$ mithilfe der für $A[1 \ldots j-1]$

return Lösung

Inkrementeller Algorithmus

InsertionSort

```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]
    for j = 2 to A.length do
        berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]
    return Lösung
```

Inkrementeller Algorithmus

InsertionSort

~~IncrementalAlg~~(array of int A) // Schreiben wir künftig so: int[ ] A

    ~~berechne Lösung für $A[1]$~~ // nix zu tun: $A[1 \ldots 1]$ ist sortiert

    for $j = 2$ **to** $A.length$ **do**

        berechne Lösung für $A[1 \ldots j]$ mithilfe der für $A[1 \ldots j-1]$

    return Lösung

Inkrementeller Algorithmus

InsertionSort

```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]
        ...kommt noch...

    return Lösung
```

Inkrementeller Algorithmus

InsertionSort

```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]
        ...kommt noch...

                                      // nicht nötig – das aufrufende Programm
    return Lösung                     // hat Zugriff auf das sortierte Feld A
```

Inkrementeller Algorithmus

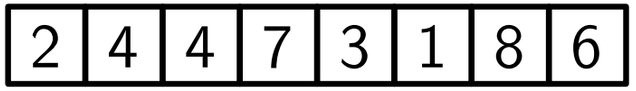
InsertionSort

```
IncrementalAlg( array of int  A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]      // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]
        // hier: füge A[j] in die sortierte Folge A[1...j − 1] ein
```

Inkrementeller Algorithmus

InsertionSort

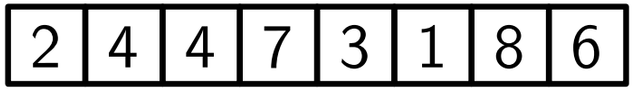
```
IncrementalAlg( array of int  A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]      // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]
        // hier: füge A[j] in die sortierte Folge A[1...j − 1] ein
```



Inkrementeller Algorithmus

InsertionSort

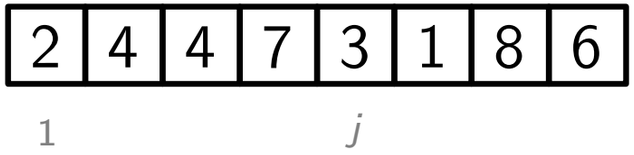
```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]      // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]
        // hier: füge A[j] in die sortierte Folge A[1...j − 1] ein
```



Inkrementeller Algorithmus

InsertionSort

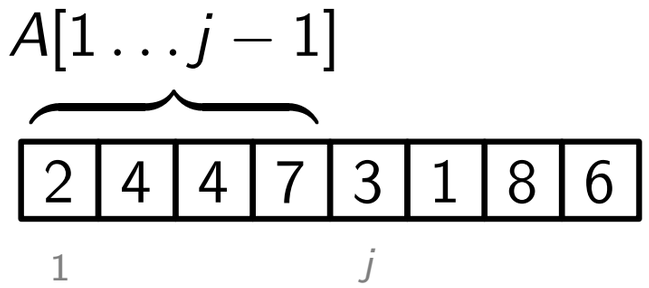
```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]
        // hier: füge A[j] in die sortierte Folge A[1...j − 1] ein
```



Inkrementeller Algorithmus

InsertionSort

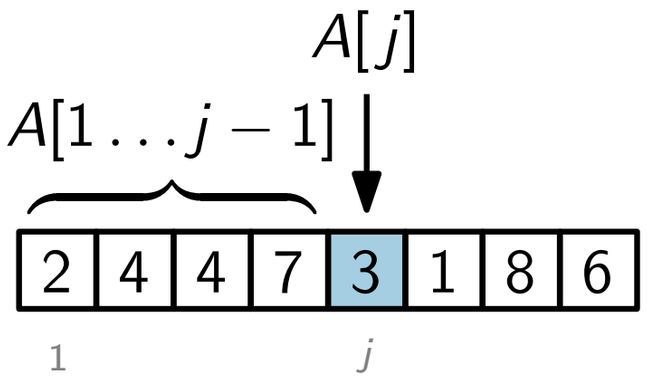
```
IncrementalAlg( array of int  A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]
        // hier: füge A[j] in die sortierte Folge A[1...j − 1] ein
```



Inkrementeller Algorithmus

InsertionSort

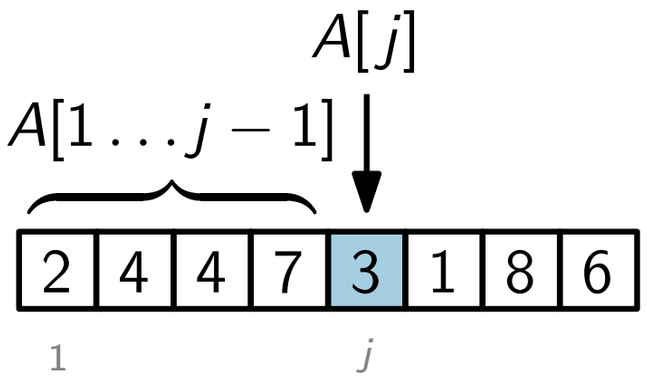
```
IncrementalAlg( array of int  A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]      // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]
        // hier: füge A[j] in die sortierte Folge A[1...j − 1] ein
```



Inkrementeller Algorithmus

InsertionSort

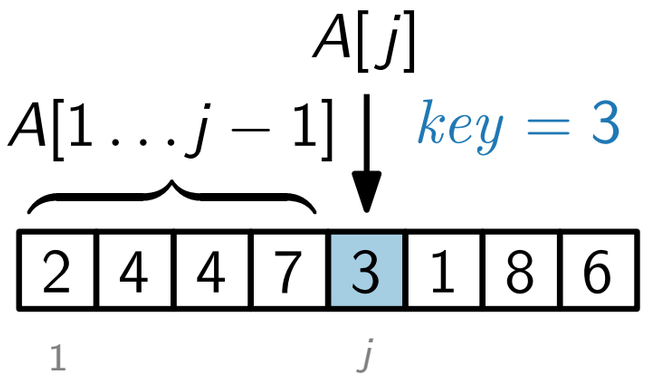
```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]      // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j-1]
        // hier: füge A[j] in die sortierte Folge A[1...j-1] ein
        key = A[j]
```



Inkrementeller Algorithmus

InsertionSort

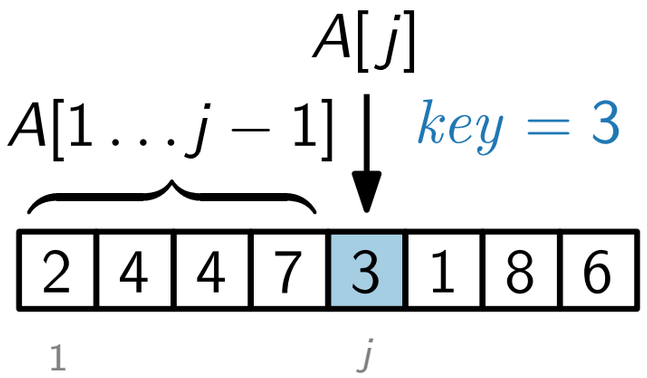
```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]
        // hier: füge A[j] in die sortierte Folge A[1...j − 1] ein
        key = A[j]
```



Inkrementeller Algorithmus

InsertionSort

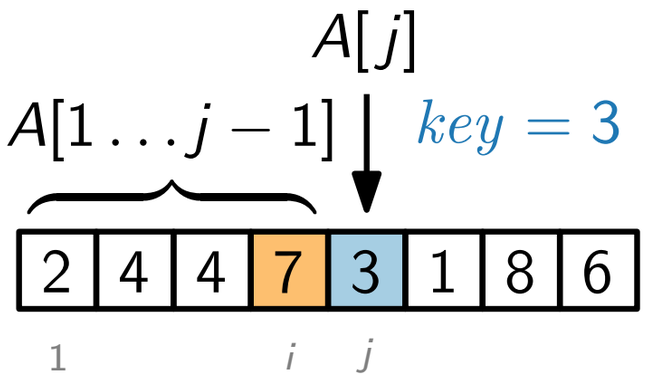
```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]
        // hier: füge A[j] in die sortierte Folge A[1...j − 1] ein
        key = A[j]
        i = j − 1
```



Inkrementeller Algorithmus

InsertionSort

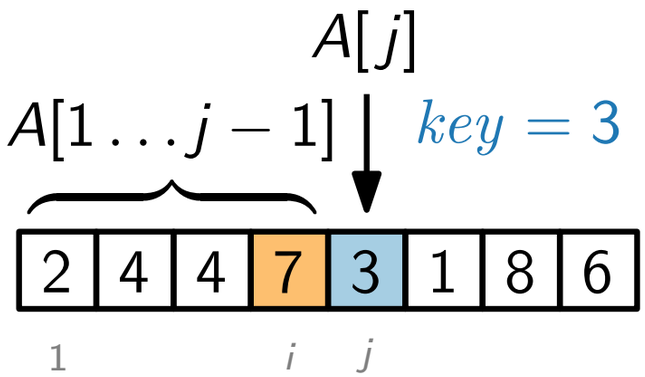
```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]      // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j-1]
        // hier: füge A[j] in die sortierte Folge A[1...j-1] ein
        key = A[j]
        i = j - 1
```



Inkrementeller Algorithmus

InsertionSort

```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j-1]
        // hier: füge A[j] in die sortierte Folge A[1...j-1] ein
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key do
            A[i+1] = A[i]
```

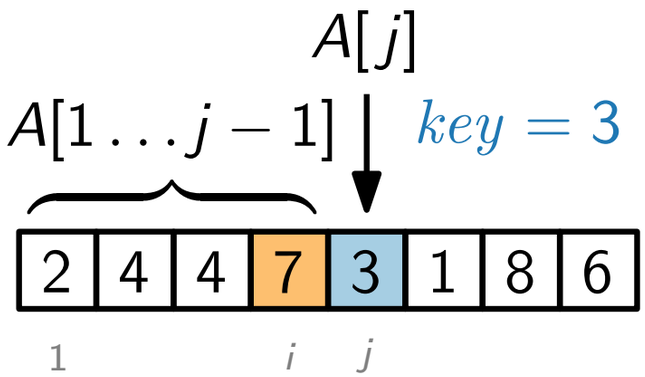


Inkrementeller Algorithmus

InsertionSort

```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j − 1]
        // hier: füge A[j] in die sortierte Folge A[1...j − 1] ein
        key = A[j]
        i = j − 1
        while i > 0 and A[i] > key do
```

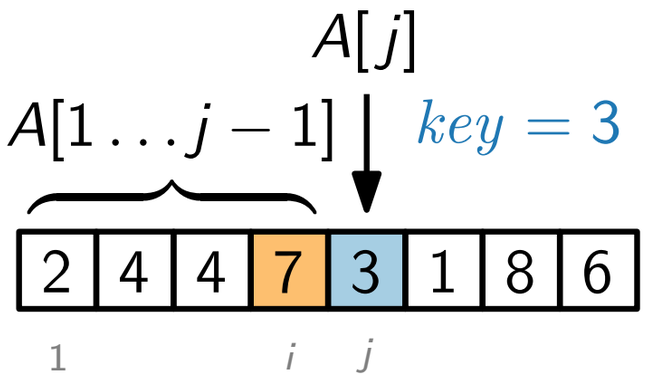
Wie verschieben wir die Einträge größer key nach rechts?



Inkrementeller Algorithmus

InsertionSort

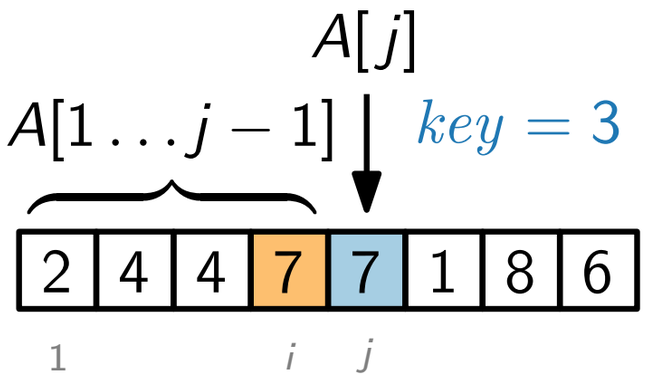
```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j-1]
        // hier: füge A[j] in die sortierte Folge A[1...j-1] ein
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key do
            A[i+1] = A[i]
```



Inkrementeller Algorithmus

InsertionSort

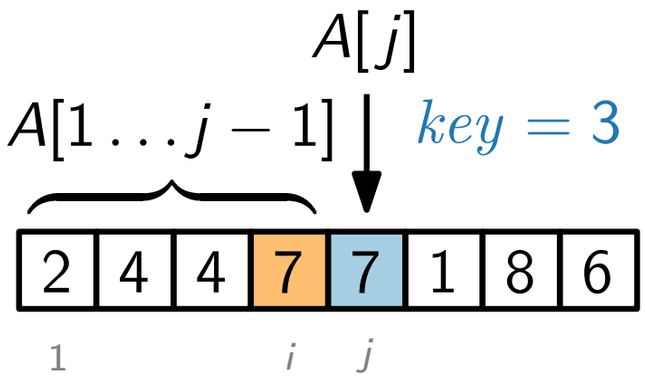
```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j-1]
        // hier: füge A[j] in die sortierte Folge A[1...j-1] ein
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key do
            A[i+1] = A[i]
```



Inkrementeller Algorithmus

InsertionSort

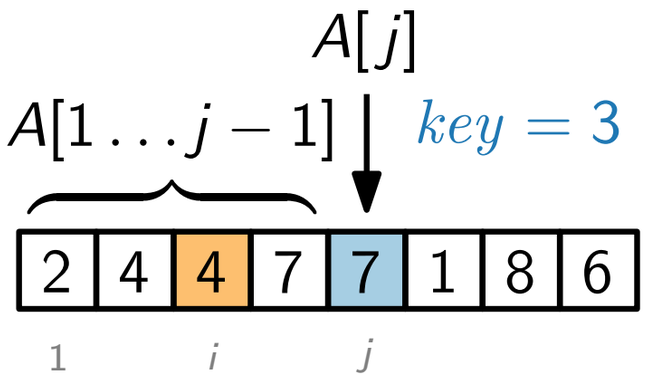
```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j-1]
        // hier: füge A[j] in die sortierte Folge A[1...j-1] ein
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key do
            A[i+1] = A[i]
            i = i - 1
```



Inkrementeller Algorithmus

INSERTIONSORT

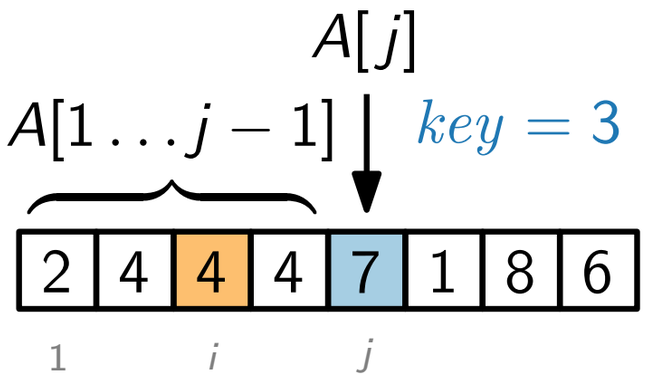
```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j-1]
        // hier: füge A[j] in die sortierte Folge A[1...j-1] ein
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key do
            A[i+1] = A[i]
            i = i - 1
```



Inkrementeller Algorithmus

InsertionSort

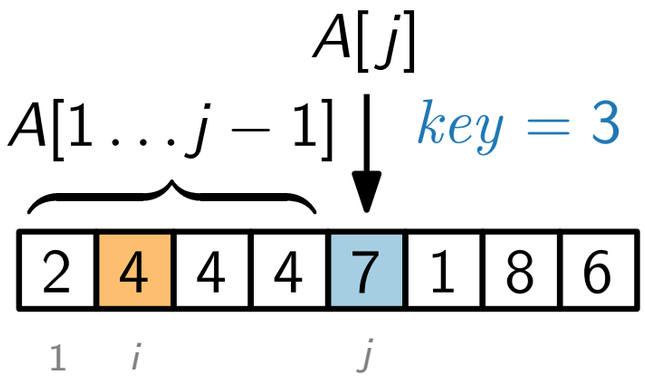
```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j-1]
        // hier: füge A[j] in die sortierte Folge A[1...j-1] ein
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key do
            A[i+1] = A[i]
            i = i - 1
```



Inkrementeller Algorithmus

InsertionSort

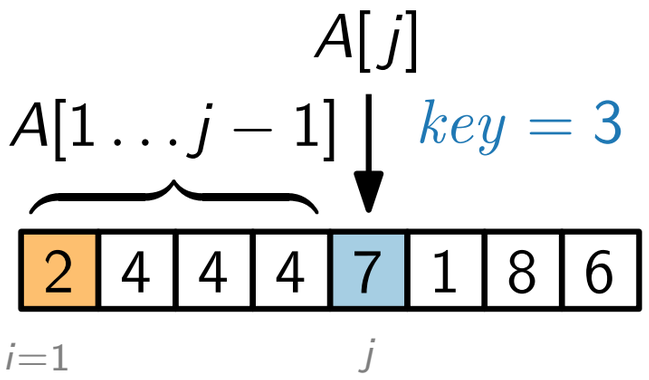
```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]      // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j-1]
        // hier: füge A[j] in die sortierte Folge A[1...j-1] ein
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key do
            A[i+1] = A[i]
            i = i - 1
```



Inkrementeller Algorithmus

InsertionSort

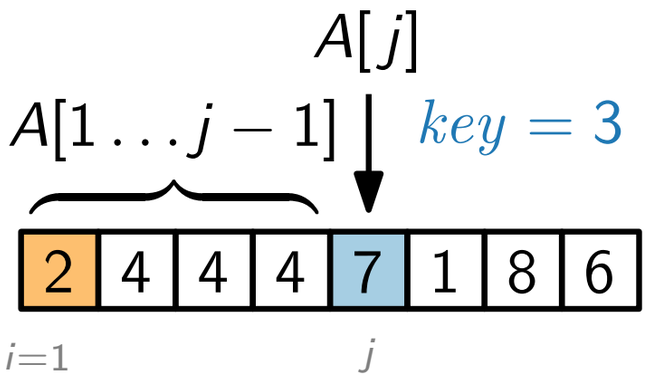
```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j-1]
        // hier: füge A[j] in die sortierte Folge A[1...j-1] ein
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key do
            A[i+1] = A[i]
            i = i - 1
```



Inkrementeller Algorithmus

InsertionSort

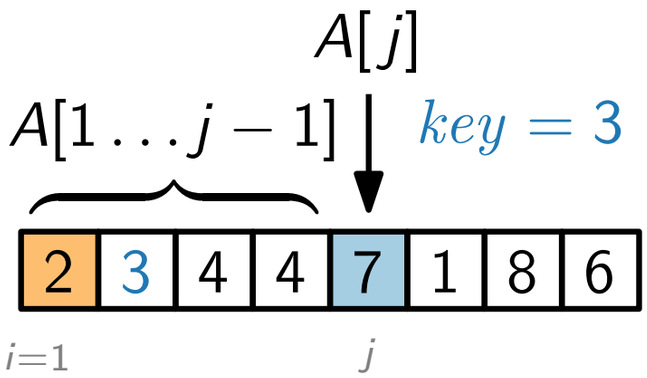
```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j-1]
        // hier: füge A[j] in die sortierte Folge A[1...j-1] ein
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key do
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key
```



Inkrementeller Algorithmus

InsertionSort

```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j-1]
        // hier: füge A[j] in die sortierte Folge A[1...j-1] ein
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key do
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key
```

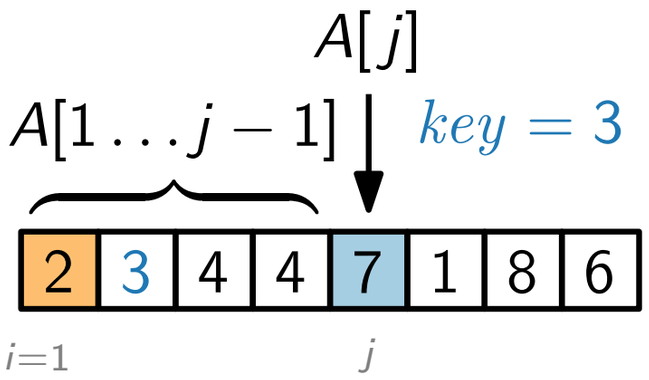


Inkrementeller Algorithmus

InsertionSort

```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
    berechne Lösung für A[1]          // nix zu tun: A[1...1] ist sortiert
    for j = 2 to A.length do
        // berechne Lösung für A[1...j] mithilfe der für A[1...j-1]
        // hier: füge A[j] in die sortierte Folge A[1...j-1] ein
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key do
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key
```

Demo.
`https://algo.uni-trier.de/demos/sort.html`



Fertig?

Fertig?

Nicht ganz. . .

Wir interessieren uns heute (und im Rest dieser Vorlesung) für folgende zentrale Fragen:

Fertig?

Nicht ganz. . .

Wir interessieren uns heute (und im Rest dieser Vorlesung) für folgende zentrale Fragen:

- Ist der Algorithmus korrekt?

Fertig?

Nicht ganz...

Wir interessieren uns heute (und im Rest dieser Vorlesung) für folgende zentrale Fragen:

- Ist der Algorithmus korrekt?
- Welche Laufzeit hat der Algorithmus?

Fertig?

Nicht ganz...

Wir interessieren uns heute (und im Rest dieser Vorlesung) für folgende zentrale Fragen:

- Ist der Algorithmus korrekt?
- Welche Laufzeit hat der Algorithmus?
- Wie viel Speicherplatz benötigt der Algorithmus?

Fertig?

Nicht ganz...

Wir interessieren uns heute (und im Rest dieser Vorlesung) für folgende zentrale Fragen:

- Ist der Algorithmus korrekt?
- Welche Laufzeit hat der Algorithmus?
- Wie viel Speicherplatz benötigt der Algorithmus?

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

Idee der **Schleifeninvariante:**

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Idee der **Schleifeninvariante:**

Wo?

Was?

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

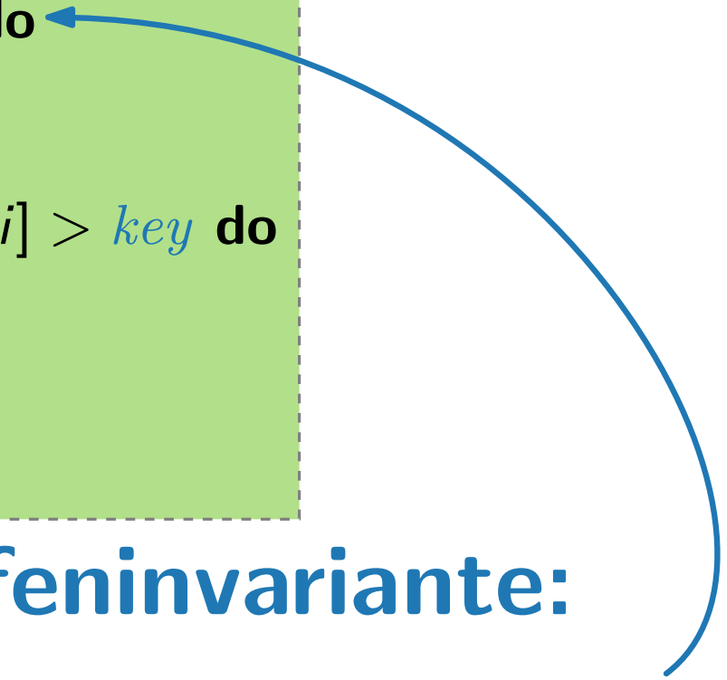
Idee der **Schleifeninvariante:**

Wo? am Beginn jeder Iteration der **for**-Schleife. . .

Was?

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```



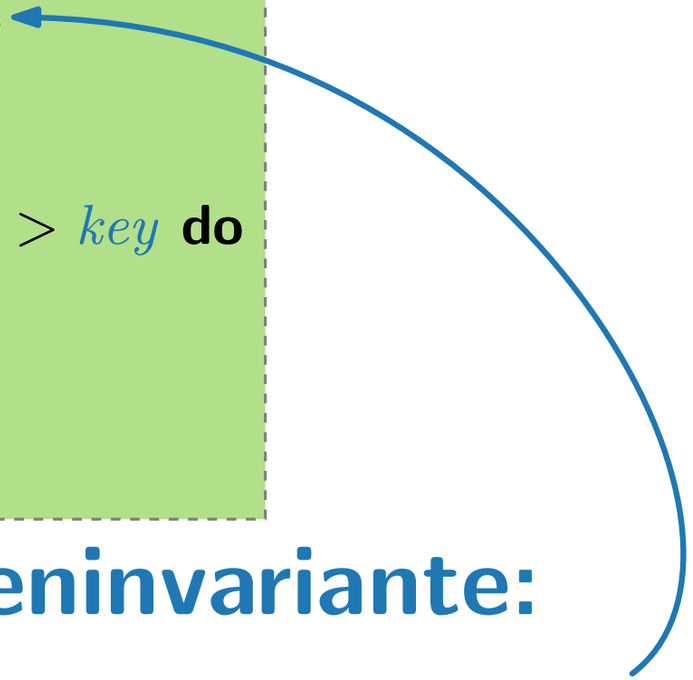
Idee der **Schleifeninvariante:**

Wo? am Beginn jeder Iteration der **for**-Schleife. . .

Was?

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```



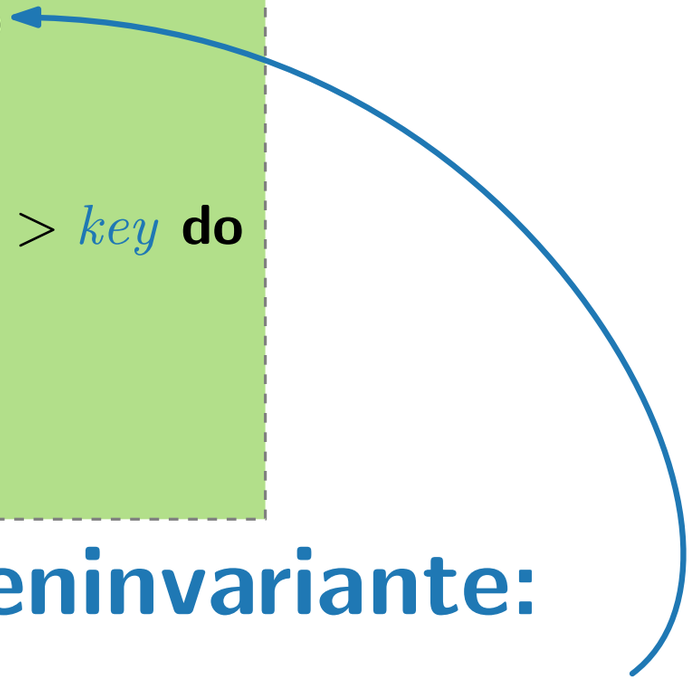
Idee der **Schleifeninvariante:**

Wo? am Beginn jeder Iteration der **for**-Schleife. . .

Was?

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```



Idee der **Schleifeninvariante:**

Wo? am Beginn jeder Iteration der **for**-Schleife. . .

Was? **WANTED:** Bedingung, die
a) an dieser Stelle immer erfüllt ist und
b) bei Abbruch der Schleife Korrektheit liefert

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Hier enthält $A[1 \ldots j-1]$

Idee der **Schleifeninvariante:**

Wo? am Beginn jeder Iteration der **for**-Schleife...

Was? **WANTED:** Bedingung, die
a) an dieser Stelle immer erfüllt ist und
b) bei Abbruch der Schleife Korrektheit liefert

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Idee der **Schleifeninvariante:**

Wo? am Beginn jeder Iteration der **for**-Schleife...

Was? **WANTED:** Bedingung, die
a) an dieser Stelle immer erfüllt ist und
b) bei Abbruch der Schleife Korrektheit liefert

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Idee der **Schleifeninvariante:**

Wo? am Beginn jeder Iteration der **for**-Schleife...

Was? **WANTED:** Bedingung, die
a) an dieser Stelle immer erfüllt ist und
b) bei Abbruch der Schleife Korrektheit liefert

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung

Zeige: Invariante ist beim 1. Schleifendurchlauf erfüllt.

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung

Zeige: Invariante ist beim 1. Schleifendurchlauf erfüllt.

Hier:

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung

Zeige: Invariante ist beim 1. Schleifendurchlauf erfüllt.

Hier: klar, denn für $j = 2$ gilt:
$A[1 \ldots j-1] = A[1 \ldots 1]$ ist unverändert und sortiert.

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung**

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung**

Zeige: Wenn die Invariante vor dem j. Schleifendurchlauf erfüllt ist, dann auch vor dem $j+1$.

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung**

Zeige: Wenn die Invariante vor dem j. Schleifendurchlauf erfüllt ist, dann auch vor dem $j+1$.

Hier:

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung**

Zeige: Wenn die Invariante vor dem j. Schleifendurchlauf erfüllt ist, dann auch vor dem $j+1$.

Hier: Eigentlich: Invariante für **while**-Schleife aufstellen und beweisen!

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung**

Zeige: Wenn die Invariante vor dem j. Schleifendurchlauf erfüllt ist, dann auch vor dem $j+1$.

Hier:

Beob. Elemente werden so lange nach rechts geschoben wie nötig. key wird korrekt eingefügt.

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓

Zeige: Wenn die Invariante vor dem j. Schleifendurchlauf erfüllt ist, dann auch vor dem $j+1$.

Hier:

Beob. Elemente werden so lange nach rechts geschoben wie nötig. key wird korrekt eingefügt.

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Zeige: Zusammengenommen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Zeige: Zusammengenommen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier:

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Zeige: Zusammengenommen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier: Verletzte Schleifenbedingung ist $j > A.length$.

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Zeige: Zusammengenommen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier: Verletzte Schleifenbedingung ist $j > A.length$.
D.h. $j = A.length + 1$.

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Zeige: Zusammengenommen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier: Verletzte Schleifenbedingung ist $j > A.length$.
D.h. $j = A.length + 1$. Einsetzen in Invariante

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Zeige: Zusammengenommen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier: Verletzte Schleifenbedingung ist $j > A.length$.
D.h. $j = A.length + 1$. Einsetzen in Invariante $\Rightarrow$ Algorithmus korrekt!

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung** ✓

Zeige: Zusammengenommen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier: Verletzte Schleifenbedingung ist $j > A.length$.
D.h. $j = A.length + 1$. Einsetzen in Invariante $\Rightarrow$ Algorithmus korrekt!

Korrektheit *beweisen*

```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Schleifeninvariante

Hier enthält $A[1 \ldots j-1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung **2.) Aufrechterhaltung** **3.) Terminierung**

Noch ein Beispiel: Fakultät berechnen

Zur Erinnerung: $k\ \mathit{Fakultät} = k! =$

Noch ein Beispiel: Fakultät berechnen

Zur Erinnerung: k *Fakultät* $= k! = 1 \cdot 2 \cdot \ldots \cdot (k-1) \cdot k$

Noch ein Beispiel: Fakultät berechnen

Zur Erinnerung: k *Fakultät* $= k! = 1 \cdot 2 \cdot \ldots \cdot (k-1) \cdot k$,

wobei $0! = 1,\ 1! = 1,\ 2! = 2,\ 3! = 6, \ldots$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |
  |
  return f
```

Was fehlt hier?
Füllen Sie die Lücke!

Zur Erinnerung: k *Fakultät* $= k! = 1 \cdot 2 \cdot \ldots \cdot (k-1) \cdot k$,
wobei $0! = 1,\ 1! = 1,\ 2! = 2,\ 3! = 6, \ldots$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Zur Erinnerung: k *Fakultät* $= k! = 1 \cdot 2 \cdot \ldots \cdot (k-1) \cdot k$,
wobei $0! = 1,\ 1! = 1,\ 2! = 2,\ 3! = 6,\ \ldots$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Korrekt?

Zur Erinnerung: k *Fakultät* $= k! = 1 \cdot 2 \cdot \ldots \cdot (k-1) \cdot k$,
wobei $0! = 1,\ 1! = 1,\ 2! = 2,\ 3! = 6, \ldots$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Korrekt?

Was passiert, wenn die Schleife gar nicht betreten wird?

Zur Erinnerung: k *Fakultät* $= k! = 1 \cdot 2 \cdot \ldots \cdot (k-1) \cdot k$,
wobei $0! = 1,\ 1! = 1,\ 2! = 2,\ 3! = 6, \ldots$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Korrekt?

Was passiert, wenn die Schleife gar nicht betreten wird?

Dann ist $j > k$.

Zur Erinnerung: k *Fakultät* $= k! = 1 \cdot 2 \cdot \ldots \cdot (k-1) \cdot k$,
wobei $0! = 1,\ 1! = 1,\ 2! = 2,\ 3! = 6, \ldots$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Korrekt?

Was passiert, wenn die Schleife gar nicht betreten wird?

Dann ist $j > k$. Da $j = 2 \Rightarrow$

Zur Erinnerung: k *Fakultät* $= k! = 1 \cdot 2 \cdot \ldots \cdot (k-1) \cdot k$,
wobei $0! = 1,\ 1! = 1,\ 2! = 2,\ 3! = 6, \ldots$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Korrekt?

Was passiert, wenn die Schleife gar nicht betreten wird?

Dann ist $j > k$. Da $j = 2 \Rightarrow$

$k = 0$ oder $k = 1$. Also $k! =$

Zur Erinnerung: k *Fakultät* $= k! = 1 \cdot 2 \cdot \ldots \cdot (k-1) \cdot k$,

wobei $0! = 1,\ 1! = 1,\ 2! = 2,\ 3! = 6, \ldots$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Korrekt?

Was passiert, wenn die Schleife gar nicht betreten wird?

Dann ist $j > k$. Da $j = 2 \Rightarrow$
$k = 0$ oder $k = 1$. Also $k! = 1$.

Zur Erinnerung: k *Fakultät* $= k! = 1 \cdot 2 \cdot \ldots \cdot (k-1) \cdot k$,
wobei $0! = 1,\ 1! = 1,\ 2! = 2,\ 3! = 6, \ldots$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Korrekt?

Was passiert, wenn die Schleife gar nicht betreten wird?

Dann ist $j > k$. Da $j = 2 \Rightarrow$

$k = 0$ oder $k = 1$. Also $k! = 1$.

Rückgabewert ist $f = 1$.

Zur Erinnerung: k *Fakultät* $= k! = 1 \cdot 2 \cdot \ldots \cdot (k-1) \cdot k$,

wobei $0! = 1,\ 1! = 1,\ 2! = 2,\ 3! = 6, \ldots$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Korrekt?

Was passiert, wenn die Schleife gar nicht betreten wird?

Dann ist $j > k$. Da $j = 2 \Rightarrow$

$k = 0$ oder $k = 1$. Also $k! = 1$.

Rückgabewert ist $f = 1$. $\Rightarrow$ Korrekt.

Zur Erinnerung: k *Fakultät* $= k! = 1 \cdot 2 \cdot \ldots \cdot (k-1) \cdot k$,

wobei $0! = 1$, $1! = 1$, $2! = 2$, $3! = 6$, $\ldots$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung

Zeige: Invariante ist beim 1. Schleifendurchlauf erfüllt.

Hier:

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung

Zeige: Invariante ist beim 1. Schleifendurchlauf erfüllt.

Hier: klar, denn für $j = 2$ gilt:

$f =$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung

Zeige: Invariante ist beim 1. Schleifendurchlauf erfüllt.

Hier: klar, denn für $j = 2$ gilt:
$f = (2-1)! = 1! = 1$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓

Zeige: Invariante ist beim 1. Schleifendurchlauf erfüllt.

Hier: klar, denn für $j = 2$ gilt:
$f = (2-1)! = 1! = 1$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung**

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung**

Zeige: Wenn die Invariante vor dem j. Schleifendurchlauf erfüllt ist, dann auch vor dem $j+1$.

Hier:

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung**

Zeige: Wenn die Invariante vor dem j. Schleifendurchlauf erfüllt ist, dann auch vor dem $j+1$.

Hier: Vor dem j. Durchlauf gilt **Invariante**, d.h. $f = (j-1)!$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung**

Zeige: Wenn die Invariante vor dem j. Schleifendurchlauf erfüllt ist, dann auch vor dem $j+1$.

Hier: Vor dem j. Durchlauf gilt **Invariante**, d.h. $f = (j-1)!$
Dann wird f mit j multipliziert $\Rightarrow$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung**

Zeige: Wenn die Invariante vor dem j. Schleifendurchlauf erfüllt ist, dann auch vor dem $j+1$.

Hier: Vor dem j. Durchlauf gilt **Invariante**, d.h. $f = (j-1)!$
Dann wird f mit j multipliziert $\Rightarrow$ $f = j!$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung**

Zeige: Wenn die Invariante vor dem j. Schleifendurchlauf erfüllt ist, dann auch vor dem $j+1$.

Hier: Vor dem j. Durchlauf gilt **Invariante**, d.h. $f = (j-1)!$

Dann wird f mit j multipliziert $\Rightarrow$ $f = j!$

Dann wird j um 1 erhöht $\Rightarrow$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung**

Zeige: Wenn die Invariante vor dem j. Schleifendurchlauf erfüllt ist, dann auch vor dem $j+1$.

Hier: Vor dem j. Durchlauf gilt **Invariante**, d.h. $f = (j-1)!$

Dann wird f mit j multipliziert $\Rightarrow \quad f = j!$

Dann wird j um 1 erhöht $\Rightarrow \quad f = (j-1)!$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung**

Zeige: Wenn die Invariante vor dem j. Schleifendurchlauf erfüllt ist, dann auch vor dem $j+1$.

Hier: Vor dem j. Durchlauf gilt **Invariante**, d.h. $f = (j-1)!$

Dann wird f mit j multipliziert $\Rightarrow$ $f = j!$

Dann wird j um 1 erhöht $\Rightarrow$ $f = (j-1)!$ $\Rightarrow$ **Invariante**

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓

Zeige: Wenn die Invariante vor dem j. Schleifendurchlauf erfüllt ist, dann auch vor dem $j+1$.

Hier: Vor dem j. Durchlauf gilt **Invariante**, d.h. $f = (j-1)!$
Dann wird f mit j multipliziert $\Rightarrow$ $f = j!$
Dann wird j um 1 erhöht $\Rightarrow$ $f = (j-1)!$ $\Rightarrow$ **Invariante**

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Zeige: Algo terminiert.
Zusammen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier:

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Zeige: Algo terminiert.
Zusammen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier: Algo terminiert, da j in jedem Durchlauf erhöht wird.

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Zeige: Algo terminiert.
Zusammen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier: Algo terminiert, da j in jedem Durchlauf erhöht wird.
Verletzte Schleifenbedingung: $j > k$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Zeige: Algo terminiert.
Zusammen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier: Algo terminiert, da j in jedem Durchlauf erhöht wird.
Verletzte Schleifenbedingung: $j > k$, also $j = k + 1$.

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Zeige: Algo terminiert.
Zusammen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier: Algo terminiert, da j in jedem Durchlauf erhöht wird.
Verletzte Schleifenbedingung: $j > k$, also $j = k + 1$.
Einsetzen von „$j = k + 1$“ in **Invariante** liefert

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Zeige: Algo terminiert.
Zusammen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier: Algo terminiert, da j in jedem Durchlauf erhöht wird.
Verletzte Schleifenbedingung: $j > k$, also $j = k + 1$.
Einsetzen von „$j = k + 1$“ in **Invariante** liefert $f = k!$

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung**

Zeige: Algo terminiert.
Zusammen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier: Algo terminiert, da j in jedem Durchlauf erhöht wird.
Verletzte Schleifenbedingung: $j > k$, also $j = k + 1$.
Einsetzen von „$j = k + 1$“ in **Invariante** liefert $f = k!$ $\Rightarrow$ Algorithmus korrekt!

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung** ✓

Noch ein Beispiel: Fakultät berechnen

```
FACTORIAL(int k)
  if k < 0 then error(...)
  f = 1
  j = 2
  while j ≤ k do
  |   f = f · j
  |   j = j + 1
  return f
```

Schleifeninvariante:

$f = (j-1)!$

1.) Initialisierung ✓ **2.) Aufrechterhaltung** ✓ **3.) Terminierung** ✓

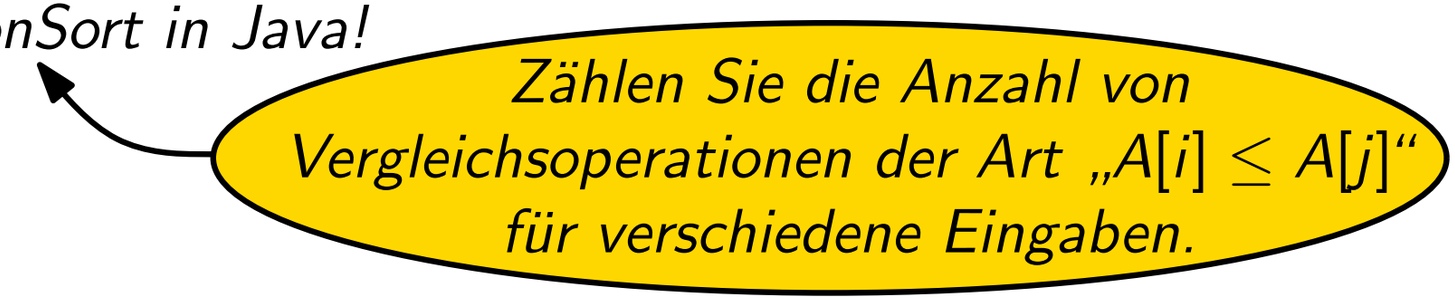
Der Algorithmus FACTORIAL(int) terminiert und liefert das korrekte Ergebnis.

Selbstkontrolle

- *Programmieren Sie InsertionSort in Java!*

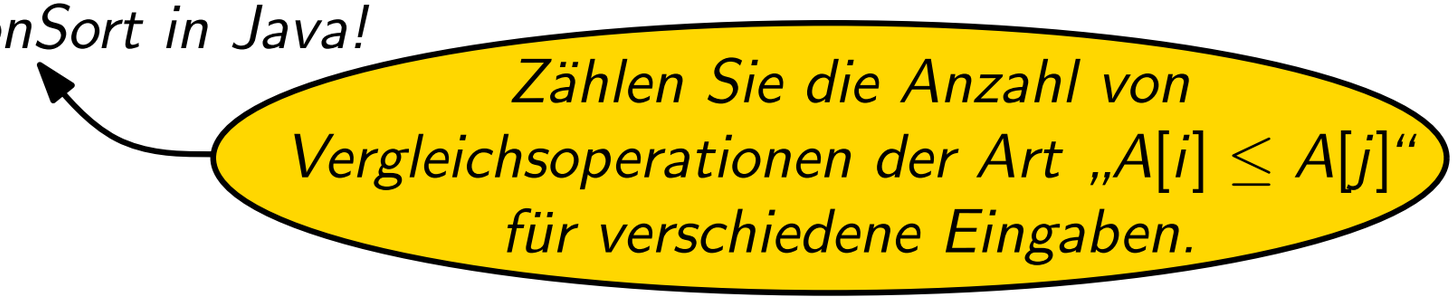
Selbstkontrolle

- *Programmieren Sie InsertionSort in Java!*



Selbstkontrolle

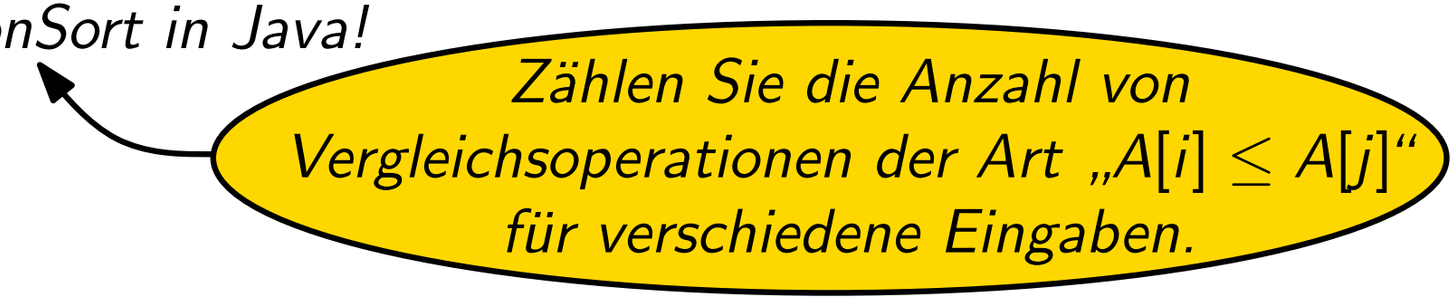
- *Programmieren Sie InsertionSort in Java!*



- *Lesen Sie Kapitel 1 und Anhang A des Buchs von Cormen et al. durch und machen Sie dazu so viel Übungsaufgaben wie möglich!*

Selbstkontrolle

- *Programmieren Sie InsertionSort in Java!*

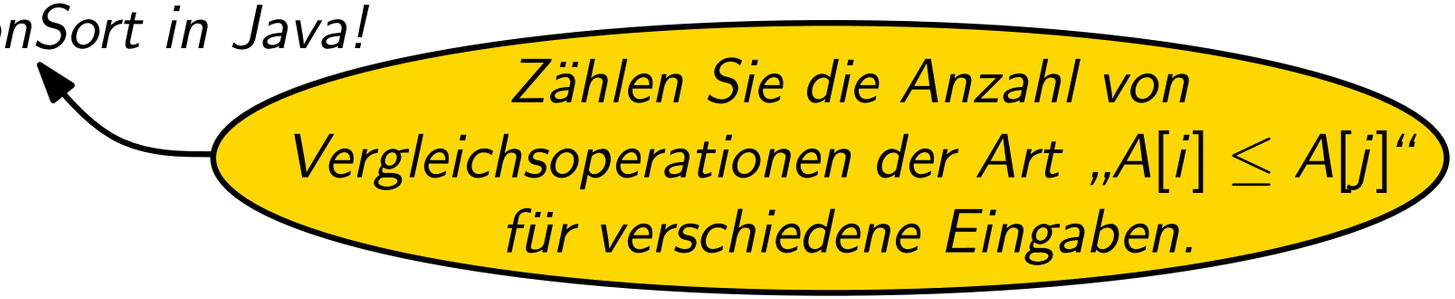


- *Lesen Sie Kapitel 1 und Anhang A des Buchs von Cormen et al. durch und machen Sie dazu so viel Übungsaufgaben wie möglich!*

- *Bringen Sie Fragen in die Übung mit!*

Selbstkontrolle

- *Programmieren Sie InsertionSort in Java!*



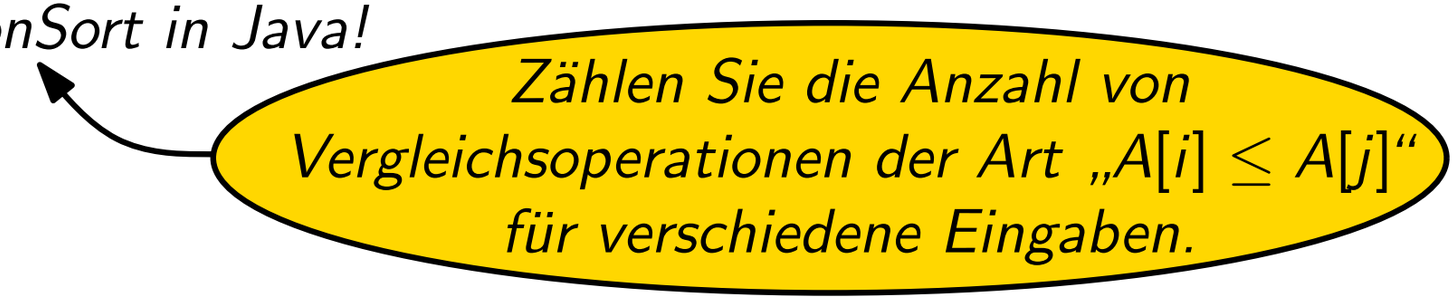
- *Lesen Sie Kapitel 1 und Anhang A des Buchs von Cormen et al. durch und machen Sie dazu so viel Übungsaufgaben wie möglich!*

- *Bringen Sie Fragen in die Übung mit!*

- *Bleiben Sie von Anfang an am Ball!*

Selbstkontrolle

- *Programmieren Sie InsertionSort in Java!*



- *Lesen Sie Kapitel 1 und Anhang A des Buchs von Cormen et al. durch und machen Sie dazu so viel Übungsaufgaben wie möglich!*

- *Bringen Sie Fragen in die Übung mit!*

- *Bleiben Sie von Anfang an am Ball!*

- *Schreiben Sie sich in die Vorlesung ein:*
  - `wuecampus2.uni-wuerzburg.de`
  - `wuestudy.zv.uni-wuerzburg.de`