Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Lehrstuhl für
INFORMATIK I
Algorithmen & Komplexität

Institut für Informatik

Algorithmen und Datenstrukturen

Wintersemester 2022/23

1. Vorlesung

Kapitel 1: Sortieren

Prof. Dr. Alexander Wolff

Lehrstuhl für Informatik I

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 $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

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 $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Das Problem

Eingabe

Gegeben: eine Folge $A = \boxed{\langle a_1, a_2, \ldots, a_n \rangle}$ von n Zahlen

Umordnung

Gesucht: eine *Permutation* $\boxed{\langle a'_1, a'_2, \ldots, a'_n \rangle}$ von A,
so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Ausgabe

Das Problem

Eingabe

Gegeben: eine Folge $A = \boxed{\langle a_1, a_2, \ldots, a_n \rangle}$ von n Zahlen

Umordnung

Gesucht: eine *Permutation* $\boxed{\langle a'_1, a'_2, \ldots, a'_n \rangle}$ von A,
so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Ausgabe

Das Problem

Eingabe

Gegeben: eine Folge $A = \boxed{\langle a_1, a_2, \ldots, a_n \rangle}$ von n Zahlen

Algorithmus

Umordnung

Gesucht: eine *Permutation* $\boxed{\langle a'_1, a'_2, \ldots, a'_n \rangle}$ von A,
so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Ausgabe

Das Problem

Eingabe

Gegeben: eine Folge $A = \boxed{\langle a_1, a_2, \ldots, a_n \rangle}$ von n Zahlen

Algorithmus

Umordnung

Gesucht: eine *Permutation* $\boxed{\langle a'_1, a'_2, \ldots, a'_n \rangle}$ von A,
so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Ausgabe

Beachte: Computerinterne Zahlendarstellung hier unwichtig!

Das Problem

Eingabe

Gegeben: eine Folge $A = \boxed{\langle a_1, a_2, \ldots, a_n \rangle}$ von n Zahlen

Algorithmus

Umordnung

Gesucht: eine *Permutation* $\boxed{\langle a'_1, a'_2, \ldots, a'_n \rangle}$ von A,
so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Ausgabe

Beachte: Computerinterne Zahlendarstellung hier unwichtig!

Wichtig:

Das Problem

Eingabe

Gegeben: eine Folge $A = \boxed{\langle a_1, a_2, \ldots, a_n \rangle}$ von n Zahlen

$\downarrow$ *Algorithmus*

Umordnung

Gesucht: eine *Permutation* $\boxed{\langle a'_1, a'_2, \ldots, a'_n \rangle}$ von A,
so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

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.

Das Problem

Eingabe

Gegeben: eine Folge $A = \boxed{\langle a_1, a_2, \ldots, a_n \rangle}$ von n Zahlen

$\downarrow$ *Algorithmus*

Umordnung

Gesucht: eine *Permutation* $\boxed{\langle a'_1, a'_2, \ldots, a'_n \rangle}$ von A,
so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

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: | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | $\longrightarrow$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

Das Problem

Eingabe

Gegeben: eine Folge $A = \boxed{\langle a_1, a_2, \ldots, a_n \rangle}$ von n Zahlen

$\Big\downarrow$ *Algorithmus*

Umordnung

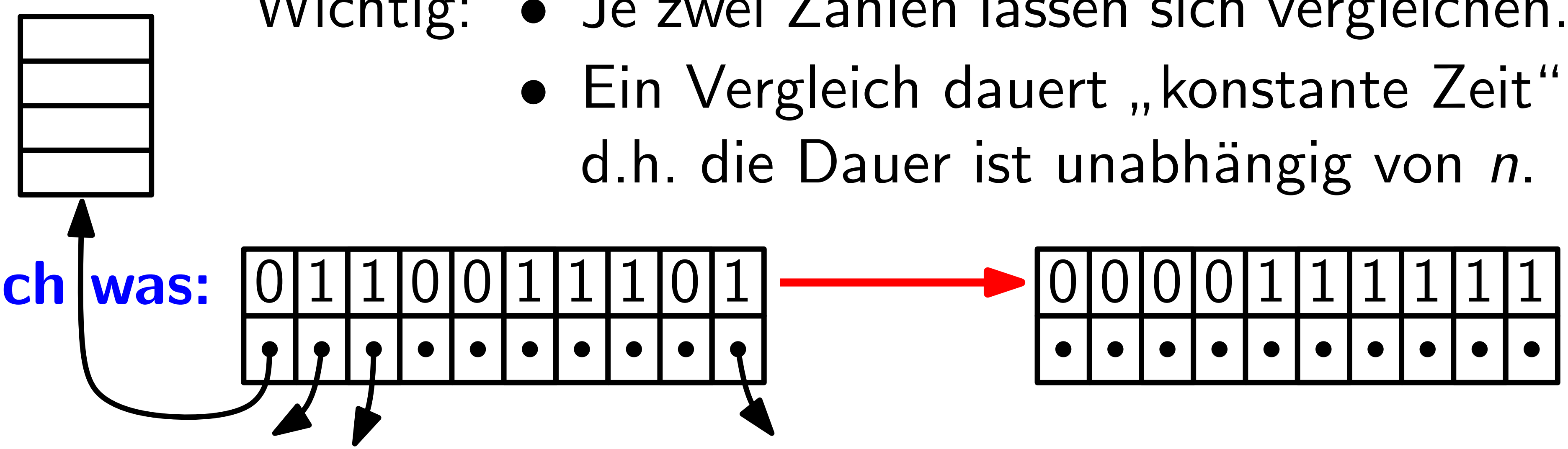
Gesucht: eine *Permutation* $\boxed{\langle a'_1, a'_2, \ldots, a'_n \rangle}$ von A,
so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

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

Eingabe

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

Algorithmus

Umordnung

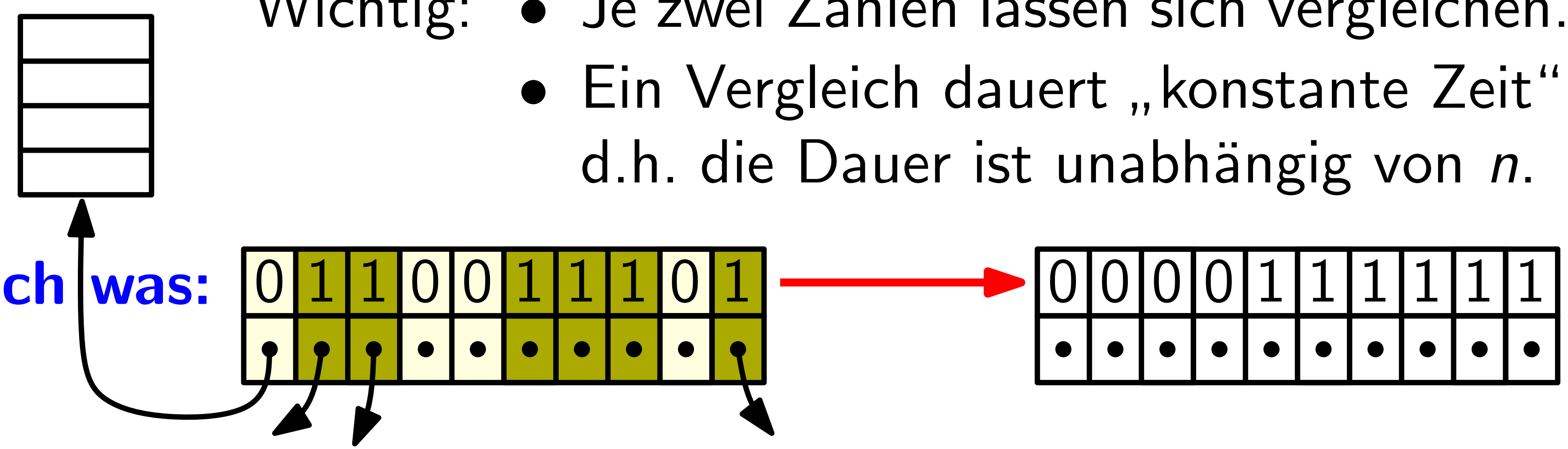
Gesucht: eine *Permutation* $\langle a'_1, a'_2, \ldots, a'_n \rangle$ von A,
so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

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

Eingabe

Gegeben: eine Folge $A = \boxed{\langle a_1, a_2, \ldots, a_n \rangle}$ von n Zahlen

Algorithmus

Umordnung

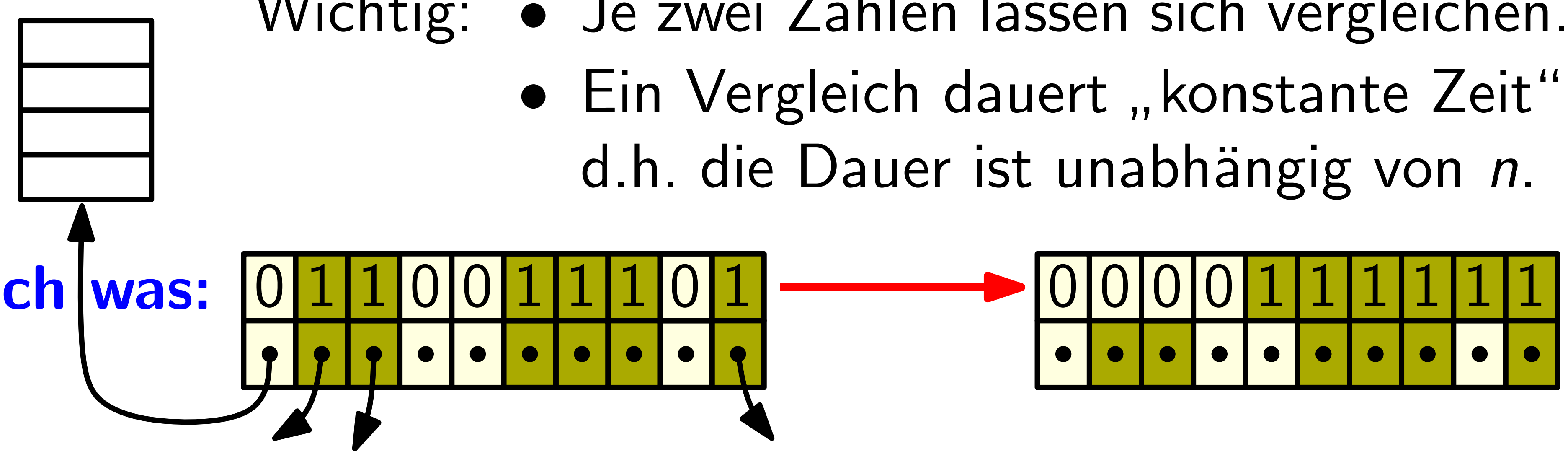
Gesucht: eine *Permutation* $\boxed{\langle a'_1, a'_2, \ldots, a'_n \rangle}$ von A,
so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

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

Eingabe

Gegeben: eine Folge $A = \boxed{\langle a_1, a_2, \ldots, a_n \rangle}$ von n Zahlen

Umordnung

Algorithmus

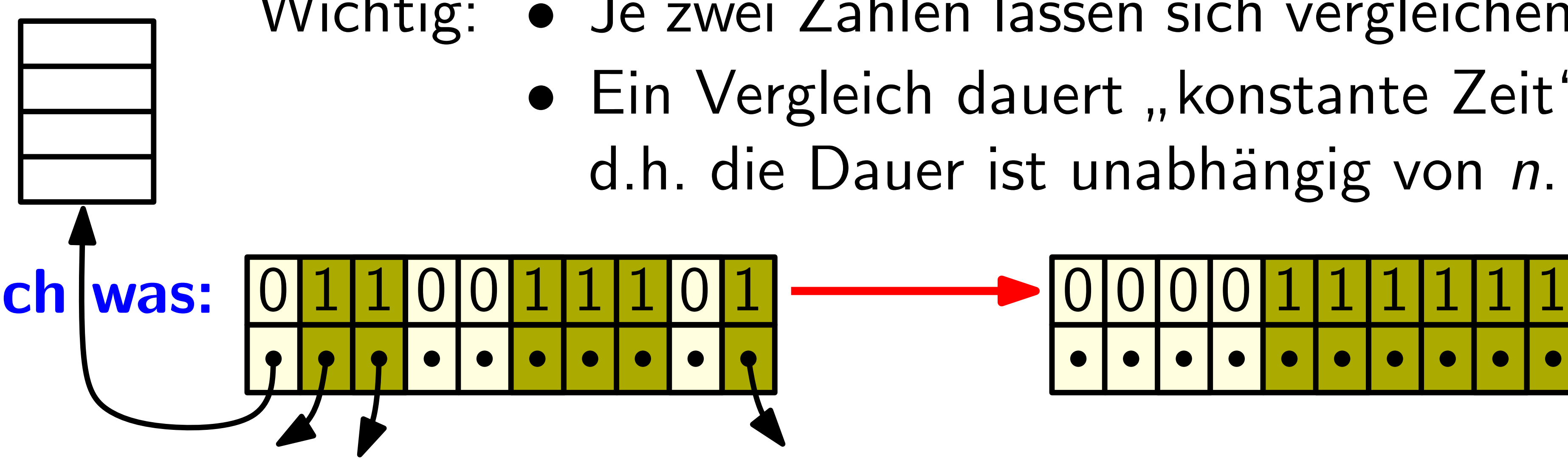
Gesucht: eine *Permutation* $\boxed{\langle a'_1, a'_2, \ldots, a'_n \rangle}$ von A,
so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

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:



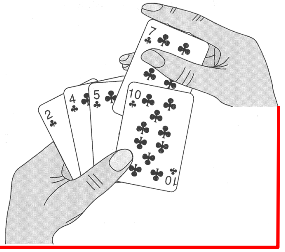
Frage an alle Erstis

Frage an alle Erstis

Wie sortieren Sie?

Eine Lösung

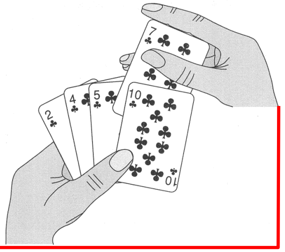
InsertionSort



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

Eine Lösung

InsertionSort

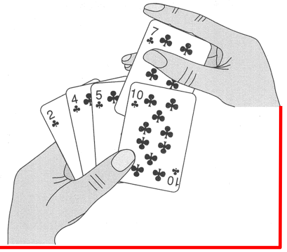


- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.

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

Eine Lösung

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]

Eine Lösung

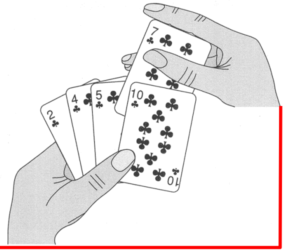
InsertionSort

- 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]

Eine Lösung

InsertionSort

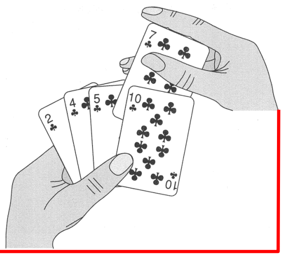


- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand nimmt Karten nacheinander auf und steckt sie (von rechts kommend) an die richtige Position zwischen die Karten in der linken Hand.
- Linke Hand hält immer eine sortierte Reihenfolge aufrecht.

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

Eine Lösung

InsertionSort



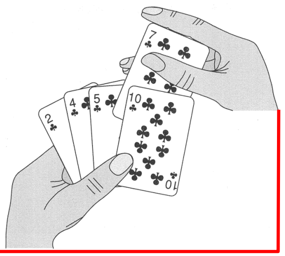
- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand nimmt Karten nacheinander auf und steckt sie (von rechts kommend) an die richtige Position zwischen die Karten in der linken Hand.

  inkrementeller Alg.
- Linke Hand hält immer eine sortierte Reihenfolge aufrecht.

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

Eine Lösung

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.

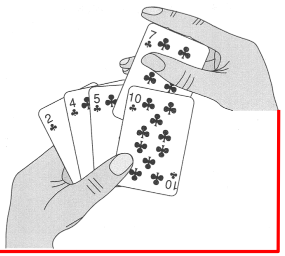
inkrementeller Alg.

- Linke Hand hält immer eine sortierte Reihenfolge aufrecht.

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

Eine Lösung

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.

inkrementeller Alg.

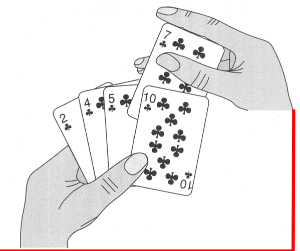
- Linke Hand hält immer eine sortierte Reihenfolge aufrecht.

Invariante!

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

Eine Lösung

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.

inkrementeller Alg.

- Linke Hand hält immer eine sortierte Reihenfolge aufrecht.

Invariante!

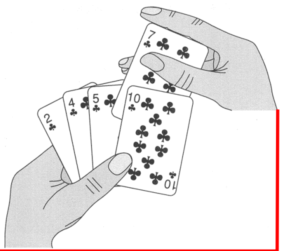
$\Downarrow$

Korrektheit

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

Eine Lösung

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.

inkrementeller Alg.

- 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]

Ein inkrementeller Algorithmus

// In Pseudocode

Ein inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of . . . A)

Ein inkrementeller Algorithmus

// In Pseudocode

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

Ein inkrementeller Algorithmus

// In Pseudocode

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

Ein inkrementeller Algorithmus

// In Pseudocode

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

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

Ein inkrementeller Algorithmus

// In Pseudocode

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

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

Variable

Ein inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of ... A)

Ein inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of ... A)

berechne Lösung für $A[1]$

Ein inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of ... A)

    berechne Lösung für $A[1]$ // Initialisierung

Ein inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of . . . A)

berechne Lösung für $A[1]$ // Initialisierung

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

Ein inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of . . . A)

  berechne Lösung für $A[1]$ // Initialisierung

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

Ein inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of . . . A)

  berechne Lösung für $A[1]$ // Initialisierung

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

Zuweisungoperator

Ein 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 $j \leftarrow 2$

Ein 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 $j = 2$

Ein inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of ... A)

berechne Lösung für $A[1]$ // Initialisierung

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

Anzahl der Elemente des Feldes A

Zuweisungoperator – in manchen Sprachen $j := 2$

– in manchen Büchern $j \leftarrow 2$

– in Java $j = 2$

Ein inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of $\ldots$ A)

  berechne Lösung für $A[1]$ // Initialisierung

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

    // Schleifenkörper; wird $(A.length - 1)$-mal durchlaufen

Ein inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of $\ldots$ 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 $A[1..j-1]$

Ein 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 $A[1], A[2], \ldots, A[j]$

Ein 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 $A[1], A[2], \ldots, A[j]$

Ein 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 $A[1], A[2], \ldots, A[j]$

Ein 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

Ein 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

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 $A[1..j-1]$

return Lösung

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 $A[1..j-1]$

... *kommt noch* ...

return Lösung

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 $A[1..j-1]$

... *kommt noch* ...

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

Ein inkrementeller Algorithmus

InsertionSort

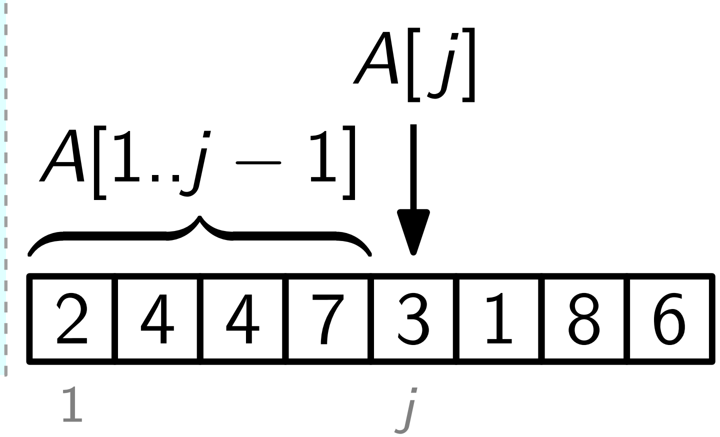
~~IncrementalAlg~~(array of … int A) // Schreiben wir künftig so: int[] A

~~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



Ein inkrementeller Algorithmus

InsertionSort

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

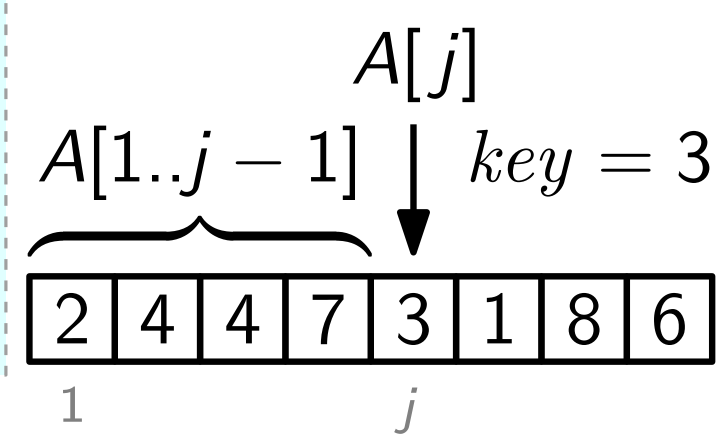
  ~~berechne Lösung für $A[1]$~~ // nix zu tun: $A[1..1]$ ist sortiert

  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

    $key = A[j]$

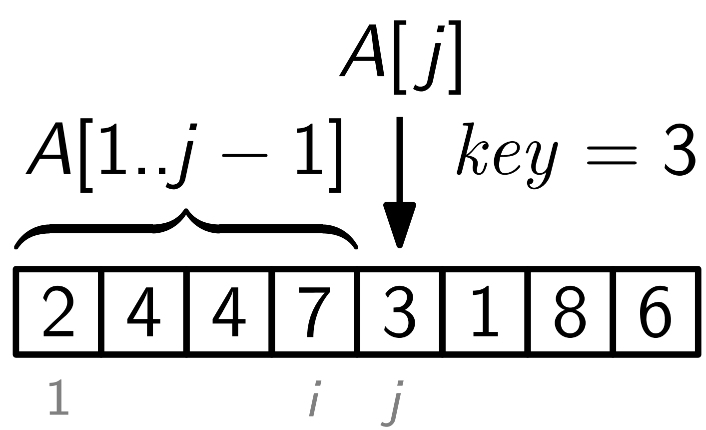


Ein inkrementeller Algorithmus

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
        $key = A[j]$
        $i = j - 1$



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**

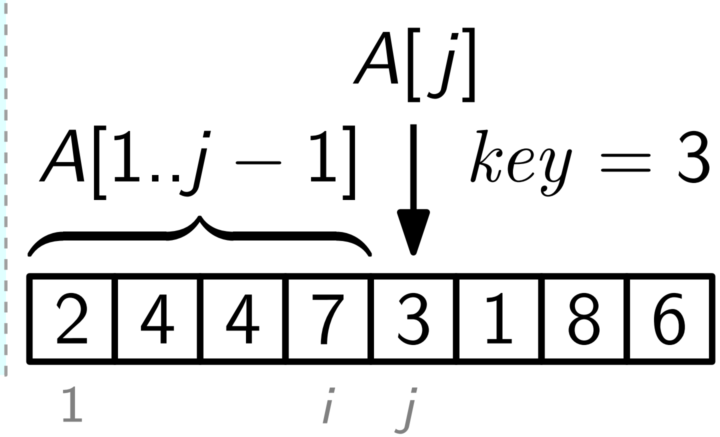
// berechne Lösung für $A[1..j]$ mithilfe der für $A[1..j-1]$

// *hier:* füge $A[j]$ in die sortierte Folge $A[1..j-1]$ ein

$key = A[j]$

$i = j - 1$

while $i > 0$ **and** $A[i] > key$ **do**



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 $A[1..j-1]$

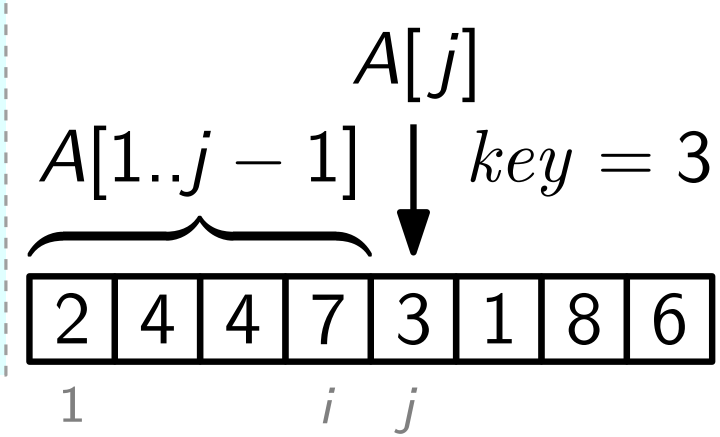
// *hier:* füge $A[j]$ in die sortierte Folge $A[1..j-1]$ ein

$key = A[j]$

$i = j - 1$

while $i > 0$ **and** $A[i] > key$ **do**

Wie verschieben wir die Einträge größer key nach rechts?



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

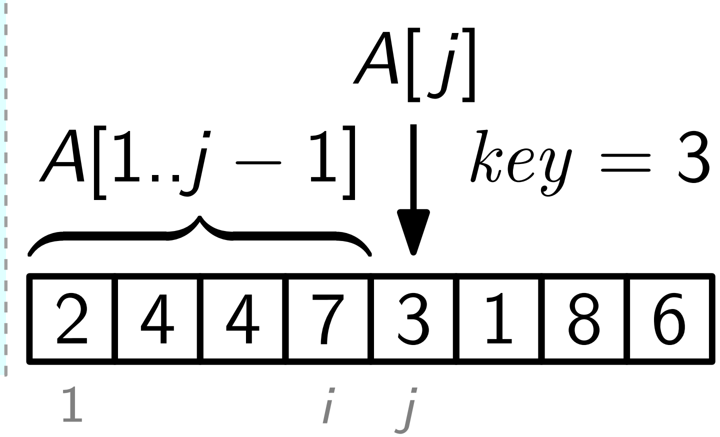
    $key = A[j]$

    $i = j - 1$

    while $i > 0$ **and** $A[i] > key$ **do**

      $A[i+1] = A[i]$

      $i = i - 1$



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

$key = A[j]$

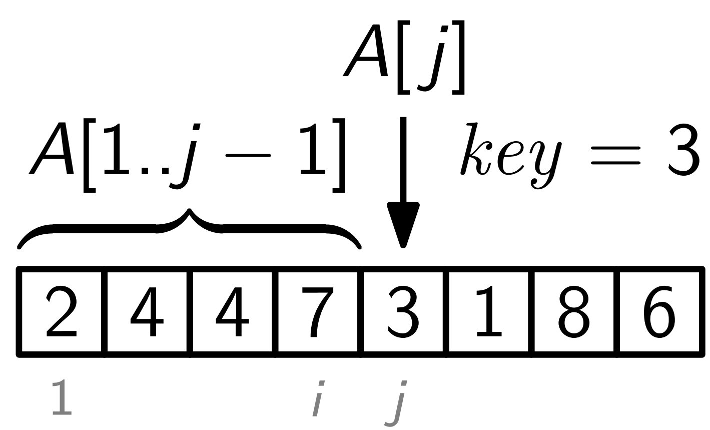
$i = j - 1$

while $i > 0$ **and** $A[i] > key$ **do**

$A[i+1] = A[i]$

$i = i - 1$

$A[i+1] = key$



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 $A[1..j-1]$

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 $A[1..j-1]$ 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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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:
$A[1..j-1] = A[1..1]$ 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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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**
    - $A[i+1] = A[i]$
    - $i = i - 1$
  - $A[i+1] = key$

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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
```

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

Schleifeninvariante

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 Inv.

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

Schleifeninvariante

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 Inv. $\Rightarrow$ 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
```

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

Schleifeninvariante

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 Inv. $\Rightarrow$ 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
```

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

Schleifeninvariante

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

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

Noch ein Beispiel: Fakultät berechnen

Factorial(int k)

if $k < 0$ **then** **error**$(\ldots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**

return f

Zur Erinnerung: k *Fakultät* $:= k! :=$

Noch ein Beispiel: Fakultät berechnen

Factorial(int k)

if $k < 0$ **then error**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**

return f

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

Noch ein Beispiel: Fakultät berechnen

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

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**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
return f

Korrekt?

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

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 $j > k$. 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**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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 $0! = 1$, $1! = 1$, $2! = 2$, $3! = 6$, ...

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**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
return f

Schleifeninvariante:

Noch ein Beispiel: Fakultät berechnen

Factorial(int k)

if $k < 0$ **then error**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
return f

Schleifeninvariante:
$f = (j-1)!$

Noch ein Beispiel: Fakultät berechnen

Factorial(int k)

if $k < 0$ **then error**$(\ldots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
return f

Schleifeninvariante:
$f = (j-1)!$

1.) Initialisierung

Noch ein Beispiel: Fakultät berechnen

Factorial(int k)

if $k < 0$ **then** **error**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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**$(\ldots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
return f

Schleifeninvariante:
$f = (j-1)!$

1.) Initialisierung ✓ *2.) Aufrechterhaltung*

Noch ein Beispiel: Fakultät berechnen

Factorial(int k)

if $k < 0$ **then error**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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 INV, d.h. $f = (j-1)!$

Noch ein Beispiel: Fakultät berechnen

Factorial(int k)

if $k < 0$ **then error**$(\dots)$
$f = 1$
$j = 2$
while $j \le k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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 INV, 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**$(\ldots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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 INV, 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**$(\ldots)$
  $f = 1$
  $j = 2$
  while $j \leq k$ **do**
    $f = f \cdot j$
    $j = j + 1$
  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 INV, 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**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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 INV, d.h. $f = (j-1)!$
Dann wird f mit j multipliziert $\Rightarrow$ $f = j!$
Dann wird j um 1 erhöht $\Rightarrow$ $f = (j-1)!$

Noch ein Beispiel: Fakultät berechnen

Factorial(int k)

if $k < 0$ **then error**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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 INV, d.h. $f = (j-1)!$
Dann wird f mit j multipliziert $\Rightarrow \quad f = j!$
Dann wird j um 1 erhöht $\Rightarrow \quad f = (j-1)! \quad \Rightarrow$ INV

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 INV, d.h. $f = (j-1)!$
Dann wird f mit j multipliziert $\Rightarrow \quad f = j!$
Dann wird j um 1 erhöht $\Rightarrow \quad f = (j-1)! \quad \Rightarrow$ INV

Noch ein Beispiel: Fakultät berechnen

Factorial(int k)

if $k < 0$ **then error**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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**$(\ldots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
    $f = f \cdot j$
    $j = j + 1$
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**$(\ldots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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**$(\ldots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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**$(\dots)$
  $f = 1$
  $j = 2$
  while $j \leq k$ **do**
    $f = f \cdot j$
    $j = j + 1$
  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**$(\ldots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
    $f = f \cdot j$
    $j = j + 1$
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 INV liefert

Noch ein Beispiel: Fakultät berechnen

Factorial(int k)

if $k < 0$ **then error**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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 INV 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* ✓

Noch ein Beispiel: Fakultät berechnen

Factorial(int k)

if $k < 0$ **then error**$(\dots)$
$f = 1$
$j = 2$
while $j \leq k$ **do**
  $f = f \cdot j$
  $j = j + 1$
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!*

Zählen Sie Vergleiche für verschiedene Eingaben.

Selbstkontrolle

- *Programmieren Sie InsertionSort in Java!*

  Zählen Sie Vergleiche für verschiedene Eingaben.

- *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!*

Zählen Sie Vergleiche für verschiedene Eingaben.

- *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!*

  Zählen Sie Vergleiche für verschiedene Eingaben.

- *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!*

Zählen Sie Vergleiche für verschiedene Eingaben.

- *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`
  - `chat.uni-wuerzburg.de/invite/TZFubc`