Korrektheitsbeweise von Algorithmen

Manfred Hauswirth | Open Distributed Systems | Einführung in die Programmierung, WS 21/22

Grundlagen der Algorithmenanalyse

Inhalt

- Wie beschreibt man einen Algorithmus?
- Rechenmodell
- Laufzeitanalyse (Zeitkomplexität)
- Speicherplatzanalyse (Raumkomplexität)
- Wie beweist man die Korrektheit eines Algorithmus?

Was ist ein mathematischer Beweis?

Informelle Definition

- Ein Beweis ist eine Herleitung einer Aussage aus bereits bewiesenen Aussagen und/oder Grundannahmen (Axiomen).

Korrektheitsbeweise

Was muss ich eigentlich zeigen?

- Häufiges Problem: Was muss man in einem Korrektheitsbeweis beweisen?

Was wissen wir?

- Problembeschreibung definiert zulässige Eingaben und zugehörige (gewünschte) Ausgaben

Korrektheitsbeweise

Wann ist ein Algorithmus korrekt?

- Wir bezeichnen einen Algorithmus für eine vorgegebene Problembeschreibung als korrekt, wenn er für jede Eingabe die in der Problembeschreibung spezifizierte Ausgabe berechnet.
- Streng genommen kann man also nur von Korrektheit sprechen, wenn vorher das angenommene Verhalten des Algorithmus geeignet beschrieben wurde.

Beispiel: Sortieren

- Problem: Sortieren
- Eingabe: Folge von n Zahlen (a1,…,a_n)
- Ausgabe: Permutation (a'_1,…,a'_n) von (a_1,…, a_n), so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Korrektheitsbeweise

Was müssen wir zeigen?

- Für jede gültige Eingabe sortiert unser Algorithmus korrekt

Aber wie? (Auf welchen Annahmen können wir aufbauen?)

- Die Grundannahme in der Algorithmik ist, dass ein Pseudocodebefehl gemäß seiner Spezifikation ausgeführt wird
- Z.B.: Die Anweisung $x \leftarrow x + 1$ bewirkt, dass die Variable x um eins erhöht wird

Erster Korrektheitsbeweis: Rückgabe ist 10+n

Ein triviales Beispiel

EinfacherAlgorithmus(n)

1. $X \leftarrow 10$
2. $Y \leftarrow n$
3. $X \leftarrow X + Y$
4. ***return*** X

Behauptung

Der Algorithmus gibt den Wert 10+n zurück.

<u>Beweis:</u>

Erster Korrektheitsbeweis: Rückgabe ist 10+n

Ein triviales Beispiel

EinfacherAlgorithmus(n)

1. $X \leftarrow 10$
2. $Y \leftarrow n$
3. $X \leftarrow X + Y$
4. ***return*** X

Behauptung

Der Algorithmus gibt den Wert 10+n zurück.

<u>Beweis:</u>

Zu Beginn des Algorithmus sind alle Variablen bis auf den Parameter n undefiniert.

Erster Korrektheitsbeweis: Rückgabe ist 10+n

Ein triviales Beispiel

EinfacherAlgorithmus(n)

1. $\boxed{X \leftarrow 10}$
2. $Y \leftarrow n$
3. $X \leftarrow X + Y$
4. ***return*** X

Behauptung

Der Algorithmus gibt den Wert 10+n zurück.

<u>Beweis:</u>

Zu Beginn des Algorithmus sind alle Variablen bis auf den Parameter n undefiniert. Der Befehl in Zeile 1 weist X den Wert 10 zu.

Korrektheitsbeweise

Ein triviales Beispiel

EinfacherAlgorithmus(n)

1. $X \leftarrow 10$
2. $\boxed{Y \leftarrow n}$
3. $X \leftarrow X + Y$
4. **return** X

Behauptung

Der Algorithmus gibt den Wert 10+n zurück.

Beweis:

Zu Beginn des Algorithmus sind alle Variablen bis auf den Parameter n undefiniert. Der Befehl in Zeile 1 weist X den Wert 10 zu. Der Befehl in Zeile 2 weist Y den Wert n zu.

Erster Korrektheitsbeweis: Rückgabe ist 10+n

Ein triviales Beispiel

EinfacherAlgorithmus(n)

1. $X \leftarrow 10$
2. $Y \leftarrow n$
3. $\boxed{X \leftarrow X + Y}$
4. ***return*** X

Behauptung

Der Algorithmus gibt den Wert 10+n zurück.

Beweis:

Zu Beginn des Algorithmus sind alle Variablen bis auf den Parameter n undefiniert. Der Befehl in Zeile 1 weist X den Wert 10 zu. Der Befehl in Zeile 2 weist Y den Wert n zu. Der Befehl in Zeile 3 weist X den Wert X + Y zu. Da X vor der Zuweisung den Wert 10 enthielt und Y den Wert n, wird X auf 10+n gesetzt.

Erster Korrektheitsbeweis: Rückgabe ist 10+n

Ein triviales Beispiel

EinfacherAlgorithmus(n)

1. $X \leftarrow 10$
2. $Y \leftarrow n$
3. $X \leftarrow X + Y$
4. **return** X

Behauptung

Der Algorithmus gibt den Wert 10+n zurück.

Beweis:

Zu Beginn des Algorithmus sind alle Variablen bis auf den Parameter n undefiniert. Der Befehl in Zeile 1 weist X den Wert 10 zu. Der Befehl in Zeile 2 weist Y den Wert n zu. Der Befehl in Zeile 3 weist X den Wert X + Y zu. Da X vor der Zuweisung den Wert 10 enthielt und Y den Wert n, wird X auf 10+n gesetzt. Der Befehl in Zeile 4 gibt X zurück. Da X zu diesem Zeitpunkt den Wert 10+n hat, folgt die Korrektheit der Behauptung.

Erster Korrektheitsbeweis: Rückgabe ist 10+n

Ein triviales Beispiel

EinfacherAlgorithmus(n)

1. $X \leftarrow 10$
2. $Y \leftarrow n$
3. $X \leftarrow X + Y$
4. ***return*** X

Ein Korrektheitsbeweis vollzieht also das Programm Schritt für Schritt nach.

Behauptung

Der Algorithmus gibt den Wert 10+n zurück.

<u>Beweis:</u>

Zu Beginn des Algorithmus sind alle Variablen bis auf den Parameter n undefiniert. Der Befehl in Zeile 1 weist X den Wert 10 zu. Der Befehl in Zeile 2 weist Y den Wert n zu. Der Befehl in Zeile 3 weist X den Wert X + Y zu. Da X vor der Zuweisung den Wert 10 enthielt und Y den Wert n, wird X auf 10+n gesetzt. Der Befehl in Zeile 4 gibt X zurück. Da X zu diesem Zeitpunkt den Wert 10+n hat, folgt die Korrektheit der Behauptung.

Korrektheitsbeweis: Maximumssuche

Ein erstes nicht triviales Beispiel

Algorithmus Max-Search(Array A)

1. max $\leftarrow$ 1
2. **for** j $\leftarrow$ 2 **to** length(A) **do**
3.   **if** A[j] > A[max] **then** max $\leftarrow$ j
4. **return** max // ist der Index eines maximalen Elements

Aufgabe

Korrektheitsbeweis muss das Programm Schritt für Schritt nachvollziehen

Problem

Wir wissen nicht, wie viele Durchläufe die **for**-Schleife benötigt.

Das hängt von der Eingabelänge ab.

Korrektheitsbeweis: Maximumssuche

Ein erstes nicht triviales Beispiel

Algorithmus Max-Search(Array A)

1. max $\leftarrow$ 1
2. **for** j $\leftarrow$ 2 **to** length(A) **do**
3.   **if** A[j] > A[max] **then** max $\leftarrow$ j
4. **return** max // ist der Index eines maximalen Elements

Abhilfe

Wir benötigen eine Aussage, die den Zustand am Ende der Schleife nach einer beliebigen Anzahl Schleifendurchläufe angibt.

Zustand

Werte der Variablen des Programms

Korrektheitsbeweis: Schleifeninvariante

Ein erstes nicht triviales Beispiel

Algorithmus Max-Search(Array A)

1. max $\leftarrow$ 1
2. **for** j $\leftarrow$ 2 **to** length(A) **do**
3. **if** A[j] > A[max] **then** max $\leftarrow$ j
4. **return** max

Definition ***Schleifeninvariante***

Eine Schleifeninvariante ist eine Aussage A(i), die

- Zu Beginn des ersten Durchlaufs gilt (Initialisierung)
- Zu Beginn des i-ten Schleifendurchlaufs gilt (in Abhängigkeit von i)

Und eine Aussage über den Zustand nach Ablauf der Schleife erlaubt, den Austrittszustand zu bestimmen.

Korrektheitsbeweis: Schleifeninvariante

Ein erstes nicht triviales Beispiel

Algorithmus Max-Search(Array A)

1. max $\leftarrow$ 1
2. **for** j $\leftarrow$ 2 **to** length(A) **do**
3.   **if** A[j] > A[max] **then** max $\leftarrow$ j
4. **return** max

Schleifeninvariante – Austrittszustand

Eine Schleife wird beendet, wenn die Schleifenbedingung nicht mehr gilt. Dieser Zustand wird als Austrittszustand bezeichnet.

D.h. im Austrittszustand ist der Wert von j, mit dem die Schleife verlassen wird,

d.h. j = length(A) + 1

Korrektheitsbeweis: Schleifeninvariante

Ein erstes nicht triviales Beispiel

Algorithmus Max-Search(Array A)

1. max $\leftarrow$ 1
2. **for** j $\leftarrow$ 2 **to** length(A) **do**
3. **if** A[j] > A[max] **then** max $\leftarrow$ j
4. **return** max

*Schleifeninvariante – Konventionen für **for**-Schleifen*

Bei einer for-Schleife nehmen wir an:

- Laufvariablen werden am Ende des Schleifendurchlaufs erhöht.
- Bei der Initialisierung sind die Laufvariablen mit dem Startwert initialisiert.

Somit kann (und sollte) die Invariante von der Laufvariablen abhängen.

Korrektheitsbeweis: Maximumssuche

Ein erstes nicht triviales Beispiel

Algorithmus Max-Search(Array A)

1. max $\leftarrow$ 1
2. **for** j $\leftarrow$ 2 **to** length(A) **do**
3. **if** A[j] > A[max] **then** max $\leftarrow$ j
4. **return** max

Abkürzung:
A[1..j-1] entspricht A[1, …, j-1]

Lemma

Die **for**-Schleife im Algorithmus Max-Search erfüllt folgende Schleifeninvariante:

(Invariante) A[max] ist ein größtes Element aus A[1..j-1].

Korrektheitsbeweis: Maximumssuche

Lemma

Die **for**-Schleife im Algorithmus Max-Search erfüllt folgende Schleifeninvariante:

(Invariante) A[max] ist ein größtes Element aus A[1..j-1].

Korrektheitsbeweis: Maximumssuche

Lemma

Die **for**-Schleife im Algorithmus Max-Search erfüllt folgende Schleifeninvariante:

(Invariante) A[max] ist ein größtes Element aus A[1..j-1].

Beweis

Vor der Schleife setzt der Befehl in Zeile 1 max auf 1.

Wir zeigen in Abhängigkeit von der Laufvariable j, dass die Invariante erfüllt ist.

Korrektheitsbeweis: Maximumssuche

Lemma

Die **for**-Schleife im Algorithmus Max-Search erfüllt folgende Schleifeninvariante:

(Invariante) A[max] ist ein größtes Element aus A[1..j-1].

Beweis

Vor der Schleife setzt der Befehl in Zeile 1 max auf 1.

Wir zeigen in Abhängigkeit von der Laufvariable j, dass die Invariante erfüllt ist.

(Erster Schritt) Zur Initialisierung der Schleife ist max=1 und j=2.

A[1..1] enthält nur ein Element, nämlich A[1].

Da A[max] = A[1] ist, ist A[max] ein größtes Element aus A[1..1].

=> Die Invariante gilt zur Initialisierung.

Korrektheitsbeweis: Maximumssuche

Lemma

Die **for**-Schleife im Algorithmus Max-Search erfüllt folgende Schleifeninvariante:

(Invariante) A[max] ist ein größtes Element aus A[1..j-1].

Beweis (fortgesetzt)

(Voraussetzung) Sei die Invariante erfüllt für $j=j_0 < \text{length}(A)+1$.

Korrektheitsbeweis: Maximumssuche

Lemma

Die **for**-Schleife in Algorithmus Max-Search erfüllt folgende Schleifeninvariante:

(Invariante) A[max] ist ein größtes Element aus A[1..j-1].

Beweis (fortgesetzt)

(Voraussetzung) Sei die Invariante erfüllt für $j=j_0 < \text{length}(A)+1$.

Zu zeigen: Die Invariante ist erfüllt für j+1. („Induktionsschritt“: j ⇨ j+1)

Korrektheitsbeweis: Maximumssuche

Lemma

Die **for**-Schleife in Algorithmus Max-Search erfüllt folgende Schleifeninvariante:

(Invariante)   A[max] ist ein größtes Element aus A[1..j-1].

Beweis (fortgesetzt)

(Voraussetzung) Sei die Invariante erfüllt für $j=j_0 < \text{length}(A)+1$.

Zu zeigen: Die Invariante ist erfüllt für j+1.   („Induktionsschritt“: j ⇨ j+1)

Wir betrachten den Durchlauf der Schleife mit Laufvariable $j=j_0$.

Korrektheitsbeweis: Maximumssuche

Lemma

Die **for**-Schleife in Algorithmus Max-Search erfüllt folgende Schleifeninvariante:

(Invariante)   A[max] ist ein größtes Element aus A[1..j-1].

Beweis (fortgesetzt)

(Voraussetzung) Sei die Invariante erfüllt für $j=j_0 < \text{length}(A)+1$.

Zu zeigen: Die Invariante ist erfüllt für j+1.   („Induktionsschritt“: j ⇨ j+1)

Wir betrachten den Durchlauf der Schleife mit Laufvariable $j=j_0$.

Falls A[j] ≤ A[max] ist, so wird die **then**-Anweisung nicht ausgeführt.
Dann ist A[max] auch größtes Element aus A[1..j].
Am Ende der Schleife wird j um 1 erhöht.
Somit gilt die Aussage auch für j+1.

Korrektheitsbeweis: Maximumssuche

Lemma

Die **for**-Schleife in Algorithmus Max-Search erfüllt folgende Schleifeninvariante:

(Invariante) A[max] ist ein größtes Element aus A[1..j-1].

Beweis (fortgesetzt)

Falls A[j] > A[max] ist, so ist nach I.V.

A[j] größer als das größte Element aus A[1...j-1]
somit ist A[j] das größte Element aus A[1..j].

Korrektheitsbeweis: Maximumssuche

Lemma

Die **for**-Schleife in Algorithmus Max-Search erfüllt folgende Schleifeninvariante:

(Invariante) A[max] ist ein größtes Element aus A[1..j-1].

Beweis (fortgesetzt)

Falls A[j] > A[max] ist, so ist nach I.V.

A[j] größer als das größte Element aus A[1...j-1]

somit ist A[j] das größte Element aus A[1..j].

In der **then**-Anweisung wird max=j gesetzt.

Damit ist A[max] das größte Element aus A[1..j].

Korrektheitsbeweis: Maximumssuche

Lemma

Die **for**-Schleife in Algorithmus Max-Search erfüllt folgende Schleifeninvariante:

(Invariante) A[max] ist ein größtes Element aus A[1..j-1].

Beweis (fortgesetzt)

Falls A[j] > A[max] ist, so ist nach I.V.

A[j] größer als das größte Element aus A[1...j-1]
somit ist A[j] das größte Element aus A[1..j].
In der **then**-Anweisung wird max=j gesetzt.
Damit ist A[max] das größte Element aus A[1..j].
Am Ende der Schleife wird j um 1 erhöht.
Damit gilt die Aussage auch für j+1.

Korrektheitsbeweis: Maximumssuche

Lemma

Die **for**-Schleife in Algorithmus Max-Search erfüllt folgende Schleifeninvariante:

(Invariante) A[max] ist ein größtes Element aus A[1..j-1].

Beweis (fortgesetzt)

Das entspricht dem Prinzip der vollständigen Induktion: Ist die Invariante vor jedem Schleifendurchlauf und vor dem Schleifenaustritt erfüllt, gilt sie für jeden Schleifendurchlauf und insbesondere am Ende der Schleife!

Korrektheitsbeweis: Maximumssuche

Korrektheit von Max-Search

Für den Algorithmus Max-Search gilt folgende Korrektheitsaussage:

(Invariante) A[max] ist ein größtes Element aus A[1..j-1].

Beweis

Durch Ausnutzen der Schleifeninvariante

Am Ende der Schleife gilt:

A[max] ist ein größtes Element aus A[1..(length(A) +1) – 1].

d.h. A[max] ist ein größtes Element aus A[1..length(A)].

Korrektheitsbeweis: Maximumssuche

Ein erstes nicht triviales Beispiel

Algorithmus Max-Search(Array A)

1. max $\leftarrow$ 1
2. **for** j $\leftarrow$ 2 **to** length(A) **do**
3.   **if** A[j] > A[max] **then** max $\leftarrow$ j
4. **return** max

Invarianten bei der Programmierung

Invarianten sollten zur Kommentierung von Schleifen benutzt werden.

Diese kann man u.a. mit Hilfe von „Assertions“ zur Laufzeit überprüfen.

Ausflug: Vollständige Induktion

Vollständige Induktion

- **Zu zeigen**: p: $N_0 \rightarrow$ Boolean (Prädikat)
  $N_0 = \{0,1,..\}$
- **Induktionsanfang**: Zu beweisen:
  p(0) ist WAHR
- **Induktionsvoraussetzung**: Für alle $n \in N_0$, mit $n <= n_0$ gilt:
  p(n) ist WAHR
- **Induktionsschritt:** Zu beweisen ist:
  p(n) ist WAHR für $n <= n_0 \Rightarrow p(n+1)$ ist WAHR
- **Induktionsschluss**: Für alle $n \in N_0$ gilt
  p(n) ist WAHR.

Beispiel für vollständigen Induktion

Technische Universität Berlin

Zu zeigen: Für alle n in N_0 gilt

$$p(n):\ 1 + 2 + 3 + \ldots + (n - 1) + n = \frac{n(n+1)}{2}$$

Open Distributed Sytems
ODS

Beispiel für vollständigen Induktion

p(n): $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$

Induktionsanfang: $1 = \frac{1(1+1)}{2} = \frac{2}{2}$

Induktionsschritt:

$$1 + 2 + 3 + \ldots + n + (n + 1) = \frac{n(n+1)}{2} + (n + 1)$$

$$= (n + 1)(\frac{n}{2} + 1)$$

$$= \frac{(n+1)(n+2)}{2}$$

Beispiel für vollständigen Induktion

Induktionsschluss: Für alle n in N_0

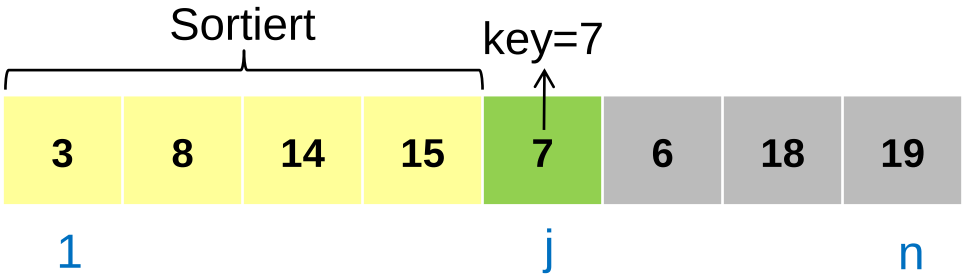
$$1 + 2 + 3 + \ldots + (n - 1) + n = \frac{n(n+1)}{2}$$

Korrektheit Insertionsort

Insertion Sort

```
InsertionSort(Array A)
1.  for j ← 2 to length(A) do
2.      key ← A[j]
3.      i ← j-1
4.      while i>0 and A[i]>key do
5.          A[i+1] ← A[i]
6.          i ← i-1
7.      A[i+1] ← key
```

- Eingabegröße n
- length(A) = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind, eine Stelle nach rechts
- Speichere key in Lücke



Korrektheitsbeweis: Insertion Sort

```
InsertionSort(Array A)
for j ← 2 to length(A) 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
```

- Eingabegröße n
- length(A) = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind, eine Stelle nach rechts
- Speichere key in Lücke

Lemma

- Die for-Schleife im Algorithmus Insertion Sort erfüllt folgende Schleifeninvariante:

(Invariante) A[1..j-1] ist die sortierte Permutation von A[1..j-1].
A[1..j-1] ist „sortiert“

Invarianten InsertionSort

InsertionSort(Array A)

1. **for** j ← 2 **to** length(A) **do**
2. key ← A[j]
3. i ← j-1
4. **while** i>0 **and** A[i]>key **do**
5. A[i+1] ← A[i]
6. i ← i-1
7. A[i+1] ← key

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“
- Austritt: A[1..length(A)] ist sortiert

Invarianten InsertionSort

```
InsertionSort(Array A)
for j ← 2 to length(A) 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
```

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“
- Austritt: A[1..length(A)] ist sortiert

Lemma

- Die while-Schleife in InsertionSort erfüllt folgende Schleifeninvariante:

(Invariante) A[1..j-1] \ A[i+1] ist „sortiert“ und A[i..j-1] > key.

Invarianten InsertionSort

InsertionSort(Array A)

1. **for** j ← 2 **to** length(A) **do**
2. key ← A[j]
3. i ← j-1
4. **while** i>0 **and** A[i]>key **do**
5. A[i+1] ← A[i]
6. i ← i-1
7. A[i+1] ← key

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“
- Inv: A[1..j-1] \ A[i+1] ist „sortiert“
  A[i..j-1] > key
- Austritt: A[1..length(A)] ist sortiert

Invarianten InsertionSort

InsertionSort(Array A)

1. **for** j $\leftarrow$ 2 **to** length(A) **do**
2. key $\leftarrow$ A[j]
3. i $\leftarrow$ j-1
4. **while** i>0 **and** A[i]>key **do**
5. A[i+1] $\leftarrow$ A[i]
6. i $\leftarrow$ i-1
7. A[i+1] $\leftarrow$ key

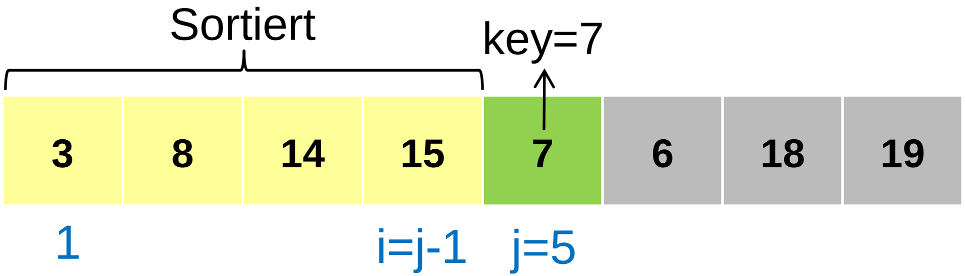
- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“
- Initialisierung: i=j-1, A[1..j-1] ist
- sortiert und A[i] > key
- Inv: A[1..j-1] \ A[i+1] ist „sortiert“
  A[i..j-1] > key
- Austritt: A[1..length(A)] ist sortiert

Invarianten InsertionSort

```
InsertionSort(Array A)
for j ← 2 to length(A) 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
```

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“

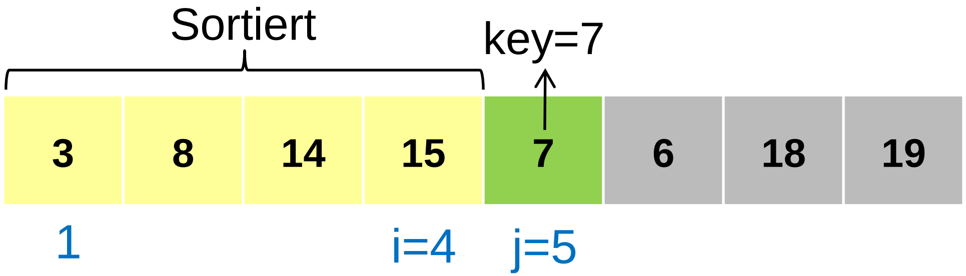
- Initialisierung: i=j-1, A[1..j-1] ist sortiert und A[i] > key
- Inv: A[1..j-1] \ A[i+1] ist „sortiert“ und A[i..j-1] > key



Invarianten InsertionSort

```
InsertionSort(Array A)
for j ← 2 to length(A) 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
```

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“
- Initialisierung: i=j-1, A[1..j-1] ist sortiert und A[i] > key
- Inv: A[1..j-1] \ A[i+1] ist „sortiert“ und A[i..j-1] > key



Invarianten InsertionSort

Technische Universität Berlin

```
InsertionSort(Array A)
for j ← 2 to length(A) 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
```

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“
- Initialisierung: i=j-1, A[1..j-1] ist sortiert und A[i] > key
- Inv: A[1..j-1] \ A[i+1] ist „sortiert“ und A[i..j-1] > key

key=7

3	**8**	**14**	**15**	**15**	**6**	**18**	**19**
1			i=4	j=5			

Open Distributed Sytems
ODS

Invarianten InsertionSort


```
InsertionSort(Array A)
for j ← 2 to length(A) 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
```

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“
- Initialisierung: i=j-1, A[1..j-1] ist sortiert und A[i] > key
- Inv: A[1..j-1] \ A[i+1] ist „sortiert“ und A[i..j-1] > key

key=7

3	**8**	**14**	**15**	**15**	**6**	**18**	**19**
1		i=3		j=5			

Invarianten InsertionSort

```
InsertionSort(Array A)
for j ← 2 to length(A) 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
```

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“
- Initialisierung: i=j-1, A[1..j-1] ist sortiert und A[i] > key
- Inv: A[1..j-1] \ A[i+1] ist „sortiert“ und A[i..j-1] > key

key=7

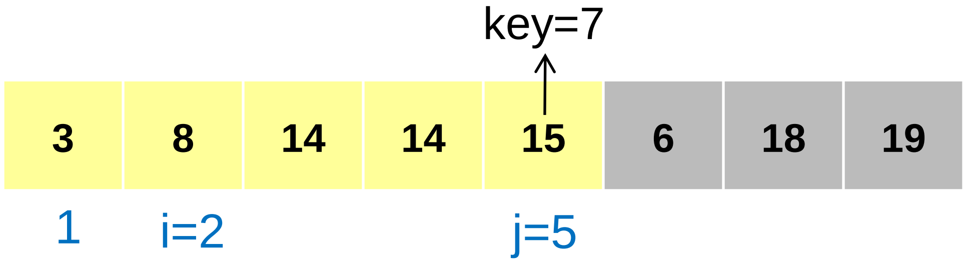
3	**8**	**14**	**14**	**15**	**6**	**18**	**19**
1		i=3		j=5			

Invarianten InsertionSort

```
InsertionSort(Array A)
for j ← 2 to length(A) 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
```

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“

- Initialisierung: i=j-1, A[1..j-1] ist sortiert und A[i] > key
- Inv: A[1..j-1] \ A[i+1] ist „sortiert“ und A[i..j-1] > key



Invarianten InsertionSort

InsertionSort(Array A)

```
for j ← 2 to length(A) 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
```

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“
- Initialisierung: i=j-1, A[1..j-1] ist sortiert und A[i] > key
- Inv: A[1..j-1] \ A[i+1] ist „sortiert“ und A[i..j-1] > key

key=7

3	**8**	**8**	**14**	**15**	**6**	**18**	**19**
1	i=2			j=5			

Invarianten InsertionSort

```
InsertionSort(Array A)
for j ← 2 to length(A) 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
```

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“
- Initialisierung: i=j-1, A[1..j-1] ist sortiert und A[i] > key
- Inv: A[1..j-1] \ A[i+1] ist „sortiert“ und A[i..j-1] > key

key=7

3	8	8	14	15	6	18	19

i=1 j=5

Invarianten InsertionSort

```
InsertionSort(Array A)
1.  for j ← 2 to length(A) do
2.      key ← A[j]
3.      i ← j-1
4.      while i>0 and A[i]>key do
5.          A[i+1] ← A[i]
6.          i ← i-1
7.      A[i+1] ← key
```

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert"

- Initialisierung: i=j-1, A[1..j-1] ist sortiert und A[i] > key
- Inv: A[1..j-1] \ A[i+1] ist „sortiert" und A[i..j-1] > key

key=7

3	**8**	**8**	**14**	**15**	**6**	**18**	**19**

i=1 j=5

Invarianten InsertionSort

```
InsertionSort(Array A)
for j ← 2 to length(A) 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
```

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“
- Initialisierung: i=j-1, A[1..j-1] ist sortiert und A[i] > key
- Inv: A[1..j-1] \ A[i+1] ist „sortiert“ und A[i..j-1] > key
- Austritt: A[1..j] ist sortiert

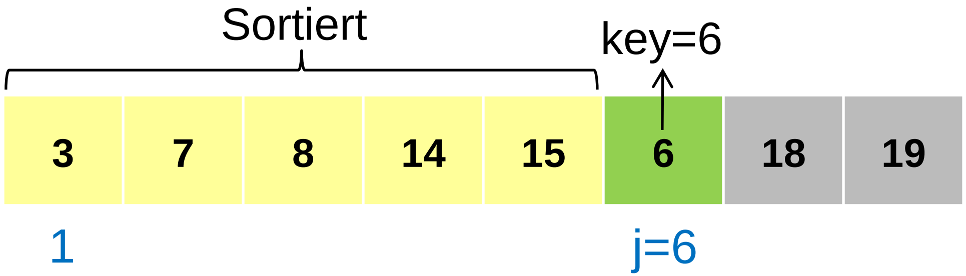
Sortiert

3	**7**	**8**	**14**	**15**	**6**	**18**	**19**
i=1				j=5			

Invarianten InsertionSort

```
InsertionSort(Array A)
for j ← 2 to length(A) 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
```

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“
- Initialisierung: i=j-1, A[1..j-1] ist sortiert und A[i] > key
- Inv: A[1..j-1] \ A[i+1] ist „sortiert“ und A[i..j-1] > key



Invarianten InsertionSort

InsertionSort(Array A)

1. **for** j ← 2 **to** length(A) **do**
2.   key ← A[j]
3.   i ← j-1
4.   **while** i>0 **and** A[i]>key **do**
5.   A[i+1] ← A[i]
6.   i ← i-1
7.   A[i+1] ← key

- Initialisierung: j=2, A[1..1] ist sortiert
- Invariante: A[1..j-1] ist „sortiert“
- Initialisierung: i=j-1, A[1..j-1] ist
- sortiert und A[i] > key
- Inv: A[1..j-1] \ A[i+1] ist „sortiert“,
  A[i..j-1] > key
- Austritt: A[1..j] \ A[i+1] ist „sortiert“, A[i] ≤ key < A[i+2]
- (wenn i+1=j, dann gilt die letzte Ungl. nicht unbedingt)
- oder i=0 und key < A[2]
- Austritt: j=length(A) ⇒ A[1..length(A)] ist „sortiert“

Korrektheitsbeweis: Insertion Sort

```
InsertionSort(Array A)
for j ← 2 to length(A) 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
```

- Eingabegröße n
- length(A) = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind, eine Stelle nach rechts
- Speichere key in Lücke

Korrektheit von InsertionSort

- Für den Algorithmus Insertion Sort gilt folgende Korrektheitsaussage:

Nach dem Aufruf von InsertionSort(A[1..length(A)]) ist A „sortiert“

Beweis

- Durch Anwendung der äußeren Schleifeninvariante.