technische universität
dortmund

LS 2 /
Informatik

Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)

Organisatorisches

Vorlesung DAP2

- Dienstag 12-14 c.t.
- Donnerstag 14-16 c.t.

Zu meiner Person

- Christian Sohler
- Fachgebiet: Komplexitätstheorie und effiziente Algorithmen
- Lehrstuhl 2, Informatik
- Raum 302

Organisatorisches

Übungen

- Mittwoch: 10-12 (2),12-14 (2), 14-16 (2), 16-18 (3)
- Donnerstag: 10-12 (3), 12-14 (2), 16-18 (2)
- Freitag: 12-14 (2), 14-16 (2)

- Zum Teil mehrere parallele Gruppen (Anzahl in Klammern)
- Anmeldung über AsSESS
- Anmeldung ab heute 14 Uhr
- Anmeldeschluss: Donnerstag 24 Uhr
- Änderungen der Übungsgruppe: Bis Montag 10 Uhr

Organisatorisches

Übungen

- Übungsblatt erscheint Mittwochs und enthält Präsenzübungen und Heimübungen
- Die erste Übung ist Präsenzübungen
- Zu Hause soll nur der Heimübungsteil bearbeitet werden
- Abgabe Heimübung: Freitag 12 Uhr Briefkästen in der OH 20
- Zulassung zur Klausur (Studienleistung Übung; Teil 1):
  - 50% der Heimübungspunkte
- Max. 3 Personen pro Übungsblatt
- Die regelmäßige Teilnahme an den Übungen wird im Hinblick auf die Tests und die Klausur dringend empfohlen

Organisatorisches

Übungen Praktikum (außer ETIT und IKT)

- Für Studierende des Bachelorstudiengangs Informatik verpflichtend
- Bachelor Elektrotechnik/Informationstechnik und Informations- und Kommunikationstechnik hat eigenes Praktikum
- Das Praktikum wird in **Java** durchgeführt
- Termine:
- Montag 10-12, 12-14
- Dienstag 10-12
- Mittwoch 10-12 (2),12-14 (3), 14-16 (2), 16-18 (2)
- Donnerstag 8-10 (3), 10-12 (2), 12-14

- Anmeldung über AsSESS (ab heute 14 Uhr; Anmeldeschluss Do. 24 Uhr; Änderungen bis Montag 10 Uhr)

Organisatorisches

Übungen Praktikum Bachelor ETIT und IKT

- Eigenes Praktikum
- Für Bachelor ETIT ist das Praktikum Wahlpflicht (Modulhandbuch Nr. ETIT-107)
- Für Bachelor IKT ist das Praktikum verpflichtend (Modulhandbuch Nr. IF-002, Element 3)
- Das Praktikum wird in **C/C++** unter Visual Studio durchgeführt
- Bis 15.4. müssen die Teilnehmer einen funktionierenden Retina-Account haben
- Bis 15.4., 12 Uhr müssen sich die ETIT und IKT Studierenden auf der Seite http://www.kt.e-technik.tu-dortmund.de/cms/de/lehre/praktikum/DAP2-Praktikum/index.html eingetragen haben (erreichbar über Homepage des Lehrstuhls KT, Prof. Kays)
- Dort gibt es eigene Übungsaufgaben
- Termine: Montags ab 15.4. 14 Uhr

Organisatorisches

Übungen Praktikum

- Heimübungen, Präsenzübungen
- Zulassung zur Klausur (Studienleistung Praktikum):
- 50% der Punkte bei den Präsenzaufgaben
- 50% der Punkte bei den Heimaufgaben
- Anwesenheitpflicht (bei Feiertagen -> andere Übung; wenn aus zwingenden Gründen verhindert -> mail an Übungsleiter)

Sonstiges

- Schüler -> Amer Krivosija
- Poolräume können außerhalb der Veranstaltungszeiten immer genutzt werden
- Weitere Informationen auf der Praktikumswebseite (erreichbar von der Vorlesungswebseite)

Organisatorisches

Lernraumbetreuung Raum (334)

Übungen

- Mo 10-12, 16-18
- Di 8-10
- Mi 10-12, 12-14, 14-16
- Do 8-10, 16-18

- Weitere Angebote werden zeitnah bekanntgegeben

Organisatorisches

Tests

- 1. Test: 4. Juni
- 2. Test: 20. Juni
- Einer der beiden Test muss mit 50% der Punkte bestanden werden (Studienleistung Übung; Teil 2)

Organisatorisches

Bei Fragen

- Meine Sprechzeiten: Montag 11-12 Uhr oder einfach nach der Vorlesung

Organisatorische Fragen zur Vorlesung an

- Amer Krivosija(amer.krivosija@tu-dortmund.de)

Organisatorische Fragen zum Praktikum an

- Marcel Preuß(preuss@ls6.cs.tu-dortmund.de)

- Außerdem INPUD Forum

Organisatorisches

Klausurtermine

- 29.7. 7:30 – 11:30 Uhr
- 24.9. 12-16 Uhr
- (Klausurlänge ist 180 Minuten)

Weitere Infos

- Vorlesungsseite http://ls2-www.cs.tu-dortmund.de/lehre/sommer2013/dap2/
- Oder von der Startseite des LS 2 -> Teaching -> DAP2

Einige Hinweise/Regeln

Klausur

- Eine Korrelation mit den Übungsaufgaben ist zu erwarten

Laptops

- Sind in der Vorlesung nicht zugelassen

Literatur

Skripte

- Kein Vorlesungsskript

Bücher und verwendete Literatur

- Cormen, Leisserson, Rivest: Introduction to Algorithms, MIT Press
- Kleinberg, Tardos: Algorithm Design, Addison Wesley

WWW

- Kurs „Introduction to Algorithms“ am MIT. Online Material (Folien, Video und Audio Files!)
- http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/CourseHome/

Lernziele

- *Bewertung von Algorithmen und Datenstrukturen*
  - Laufzeitanalyse
  - Speicherbedarf
  - Korrektheitsbeweise
- *Kenntnis grundlegender Algorithmen und Datenstrukturen*
  - Sortieren
  - Wörterbücher
  - Graphalgorithmen
- *Kenntnis grundlegender Entwurfsmethoden*
  - Teile und Herrsche
  - gierige Algorithmen
  - dynamische Programmierung

Lernziele

- *Unterschiede zu DAP1*
- DAP 1 behandelt Algorithmik aus der Perspektive der Softwaretechnik
- DAP 2 legt die theoretischen Grundlagen zur Algorithmenanalyse

Motivation

Beispiele für algorithmische Probleme, die z.T. mit Hilfe komplexer mathematischer Methoden gelöst werden

- Internetsuchmaschinen
- Berechnung von Bahnverbindungen
- Optimierung von Unternehmensabläufen
- Datenkompression
- Computer Spiele
- Datenanalyse

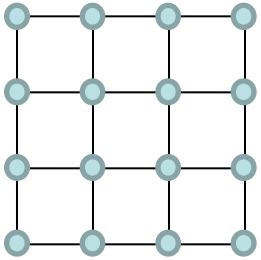
Alle diese Bereiche sind (immer noch) Stoff aktueller Forschung im Bereich Datenstrukturen und Algorithmen

Motivation

Problembeschreibung

- Ein m x n- Gitter heißt c-färbbar, wenn man seine Knoten mit c Farben so färben kann, dass kein am Gitter orientiertes achsenparalleles Rechteck alle Eckknoten in derselben Farbe hat
- Aufgabe: Finde eine 4-Färbung für ein 17x17 Gitter (289$ Problem)

- Beispiel:
(4x4 Gitter)

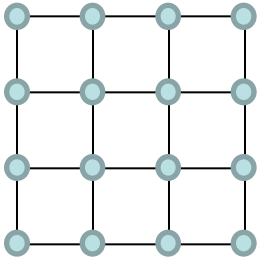


- http://blog.computationalcomplexity.org/2009/11/17x17-challenge-worth-28900-this-is-not.html

Motivation

Problembeschreibung

- Ein m x n- Gitter heißt c-färbbar, wenn man seine Knoten mit c Farben so färben kann, dass kein am Gitter orientiertes achsenparalleles Rechteck alle Eckknoten in derselben Farbe hat
- Aufgabe: Finde eine 4-Färbung für ein 17x17 Gitter **Gelöst!**
- Beispiel:
  (4x4 Gitter)



- http://blog.computationalcomplexity.org/2009/11/17x17-challenge-worth-28900-this-is-not.html

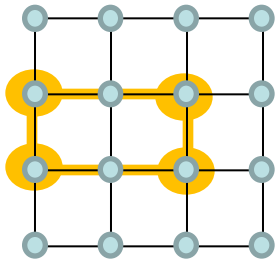
Motivation

Problembeschreibung

- Ein m x n- Gitter heißt c-färbbar, wenn man seine Knoten mit c Farben so färben kann, dass kein am Gitter orientiertes achsenparalleles Rechteck alle Eckknoten in derselben Farbe hat
- Aufgabe: Finde eine 4-Färbung für ein 17x17 Gitter

Gelöst!

- Beispiel:
  (4x4 Gitter)
- Die vier unterlegten Knoten dürfen z.B. nicht alle dieselbe Farbe haben



- http://blog.computationalcomplexity.org/2009/11/17x17-challenge-worth-28900-this-is-not.html

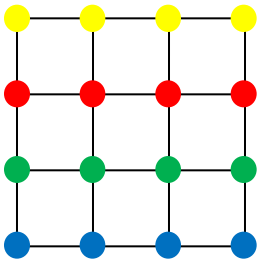
Motivation

Problembeschreibung

- Ein m x n- Gitter heißt c-färbbar, wenn man seine Knoten mit c Farben so färben kann, dass kein am Gitter orientiertes achsenparalleles Rechteck alle Eckknoten in derselben Farbe hat
- Aufgabe: Finde eine 4-Färbung für ein 17x17 Gitter

Gelöst!

- Beispiel:
  (4x4 Gitter)



4x4 Gitter ist 4-färbbar!
Geht es besser?

- http://blog.computationalcomplexity.org/2009/11/17x17-challenge-worth-28900-this-is-not.html

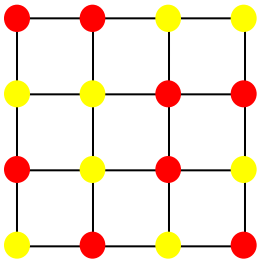
Motivation

Problembeschreibung

- Ein m x n- Gitter heißt c-färbbar, wenn man seine Knoten mit c Farben so färben kann, dass kein am Gitter orientiertes achsenparalleles Rechteck alle Eckknoten in derselben Farbe hat
- Aufgabe: Finde eine 4-Färbung für ein 17x17 Gitter

Gelöst!

- Beispiel:
  (4x4 Gitter)



Ja! 4x4 Gitter ist 2-färbbar!

- http://blog.computationalcomplexity.org/2009/11/17x17-challenge-worth-28900-this-is-not.html

Motivation

17x17 Problem

- Es ist z.Z. nicht möglich, das 17x17 Problem mit einem Rechner zu lösen
- Warum ist dieses Problem so schwer zu lösen?
- Es gibt sehr viele Färbungen!

Fragen/Aufgaben

- Können wir die Laufzeit eines Algorithmus vorhersagen?
- Können wir bessere Algorithmen finden?

Algorithmenentwurf

Anforderungen

- Korrektheit
- Effizienz (Laufzeit, Speicherplatz)

Entwurf umfasst

1. Beschreibung des Algorithmus/der Datenstruktur
2. Korrektheitsbeweis
3. Analyse von Laufzeit und Speicherplatz

Algorithmenentwurf

Warum mathematische Korrektheitsbeweise?

- Fehler können fatale Auswirkungen haben (Steuerungssoftware in Flugzeugen, Autos, AKWs)
- Fehler können selten auftreten („Austesten“ funktioniert nicht)

Der teuerste algorithmische Fehler?

- Pentium bug (>400 Mio $)
- Enormer Image Schaden
- Trat relativ selten auf

Algorithmenentwurf

Warum Laufzeit/Speicherplatz optimieren?

- Riesige Datenmengen durch Vernetzung (Internet)
- Datenmengen wachsen schneller als Rechenleistung und Speicher
- Physikalische Grenzen
- Schlechte Algorithmen versagen häufig bereits bei kleinen und mittleren Eingabegrößen

1. Teil der Vorlesung – Grundlagen der Algorithmenanalyse

Inhalt

- Wie beschreibt man einen Algorithmus?
- Rechenmodell
- Laufzeitanalyse
- Wie beweist man die Korrektheit eines Algorithmus?

Pseudocode

- Beschreibungssprache ähnlich wie C, Java, Pascal, etc…
- Hauptunterschied: Wir benutzen immer die klarste und präziseste Beschreibung
- Manchmal kann auch ein vollständiger Satz die beste Beschreibung sein
- Wir ignorieren Software Engineering Aspekte wie
  - Modularität
  - Fehlerbehandlung

Sortieren

- Problem: Sortieren
- Eingabe: Folge von n Zahlen $(a_1, \ldots, a_n)$
- Ausgabe: Permutation $(a_1', \ldots, a_n')$ von $(a_1, \ldots, a_n)$, so dass $a_1' \leq a_2' \leq \ldots \leq a_n'$

Beispiel:

- Eingabe: 15, 7, 3, 18, 8, 4
- Ausgabe: 3, 4, 7, 8, 15, 18

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

Pseudocode

- Schleifen (**for**, **while**, **repeat**)

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

Pseudocode

- Schleifen (**for**, **while**, **repeat**)

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

Pseudocode

- Zuweisungen durch ←

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

Pseudocode

- Variablen (z.B. i, j, key) sind lokal definiert

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

Pseudocode

- Keine Typdeklaration, wenn Typ klar aus dem Kontext

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

Pseudocode

- Zugriff auf Feldelemente mit [.]

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

Pseudocode

- Verbunddaten sind typischerweise als Objekte organisiert
- Ein Objekt besteht aus Attributen instantiiert durch Attributwerte
- Beispiel: Feld wird als Objekt mit Attribut Länge betrachtet

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

Pseudocode

- Beispiel: Objekt ist Graph G mit Knotenmenge V
- Auf den Attributwert V von Graph G wird mit V[G] zugegriffen

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

Pseudocode

- Objekte werden als Zeiger referenziert, d.h. für alle Attribute f eines Objektes x bewirkt y ← x das gilt: f[y] = f[x].

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

Pseudocode

- Blockstruktur durch Einrücken

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

Pseudocode

- Bedingte Verzweigungen (**if then else**)

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

Pseudocode

- Prozeduren „call-by-value“ ; jede aufgerufene Prozedur erhält neue Kopie der übergebenen Variable
- Die lokalen Änderungen sind nicht global sichtbar
- Bei Objekten wird nur der Zeiger kopiert (lokale Änderungen am Objekt global sichtbar)

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

Pseudocode

- Rückgabe von Parametern durch **return**

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
```

Beschreibung des Algorithmus in Pseudocode (kein C, Java, etc.)

Pseudocode

- Kommentare durch ➢

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
```

Idee InsertionSort

- *Die ersten j-1 Elemente sind sortiert (zu Beginn j=2)*
- *Innerhalb eines Schleifendurchlaufs wird das j-te Element in die sortierte Folge eingefügt*
- *Am Ende ist die gesamte Folge sortiert*

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
```

Beispiel

8	15	3	14	7	6	18	19

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
```

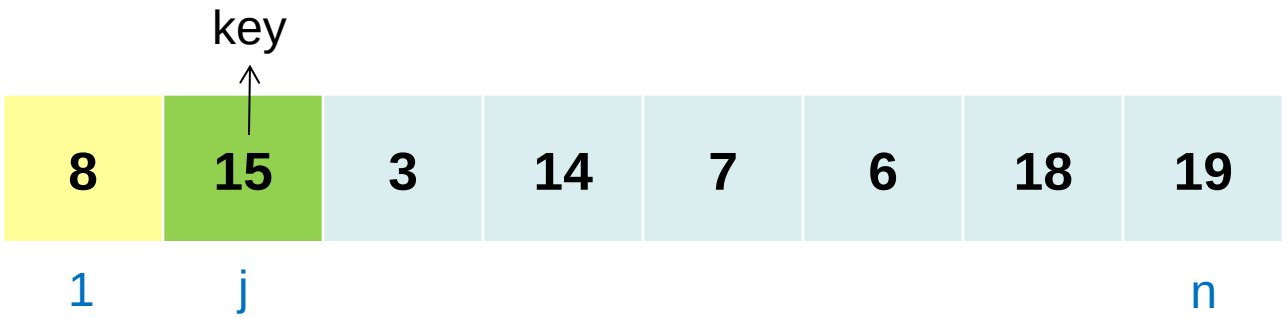
- Eingabegröße n
- length[A] = n

8	15	3	14	7	6	18	19
1	j						n

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
```

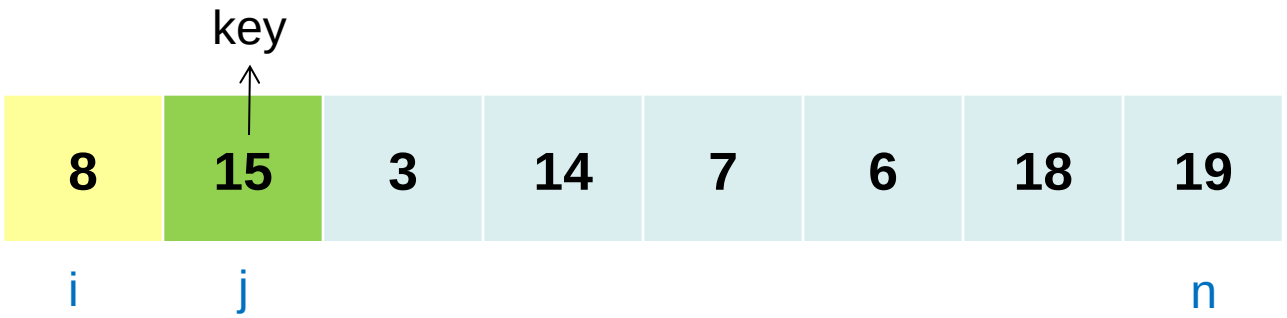
- Eingabegröße n
- length[A] = n



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
```

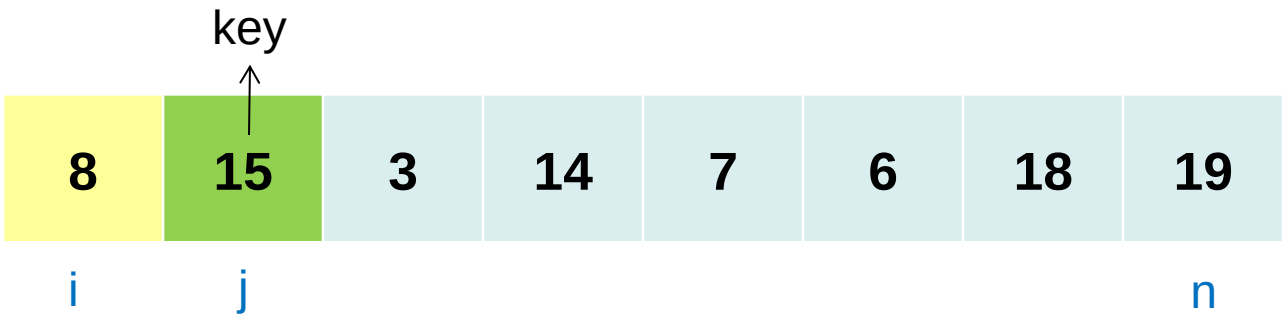
- Eingabegröße n
- length[A] = n



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
```

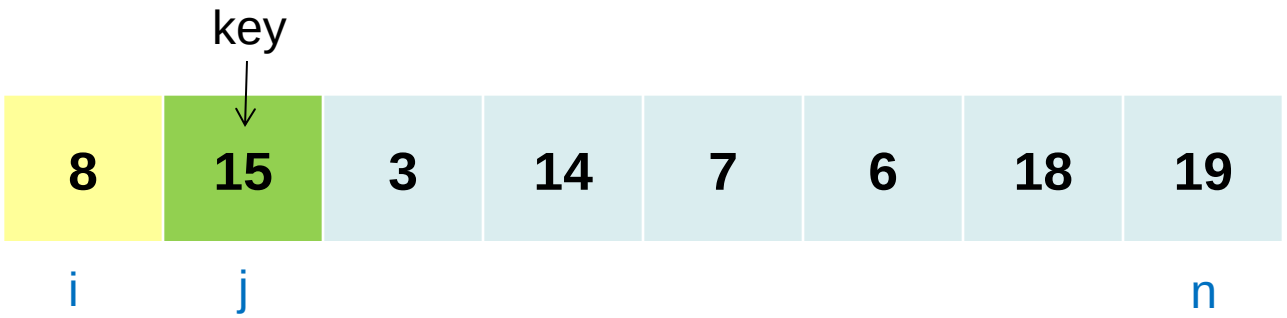
- Eingabegröße n
- length[A] = n



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



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
```

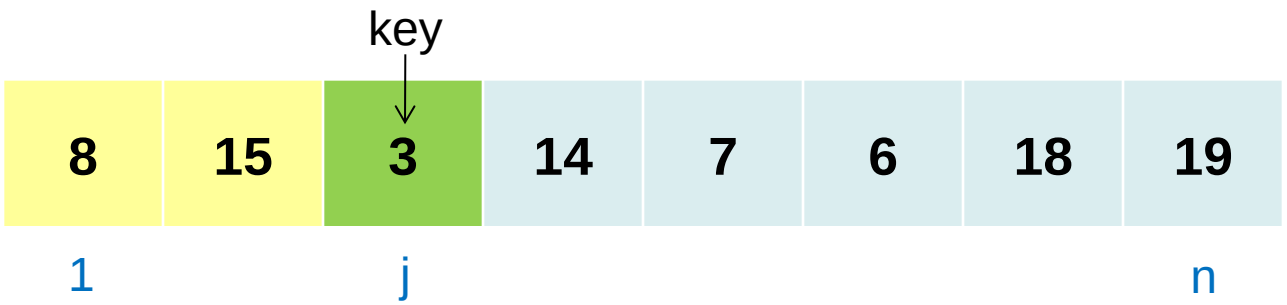
- Eingabegröße n
- length[A] = n

8	15	3	14	7	6	18	19
1		j					n

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



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

key=3

8	15	3	14	7	6	18	19
1	i	j					n

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

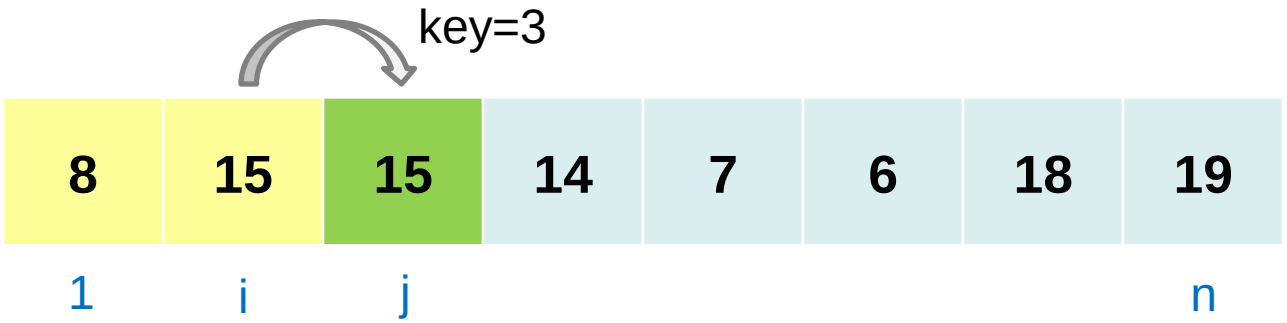
key=3

8	15	3	14	7	6	18	19
1	i	j					n

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



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

key=3

8	15	15	14	7	6	18	19
i		j					n

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

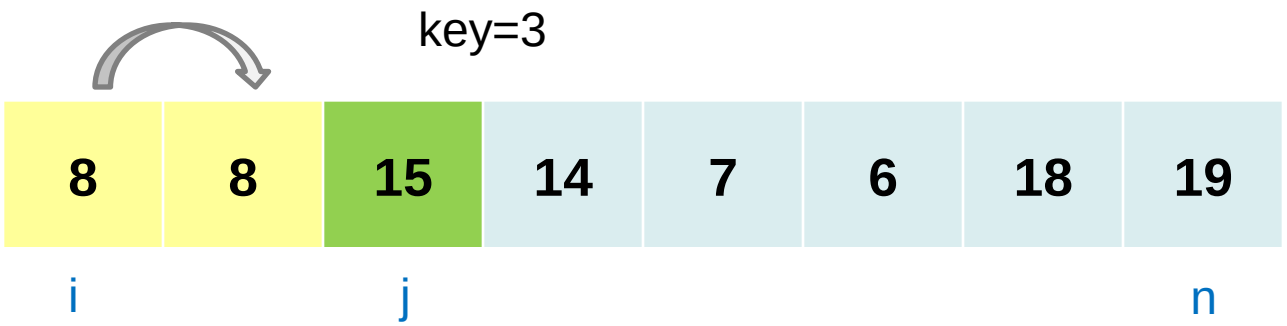
key=3

8	15	15	14	7	6	18	19
i		j					n

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



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

key=3

8	8	15	14	7	6	18	19
i	1		j				n

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

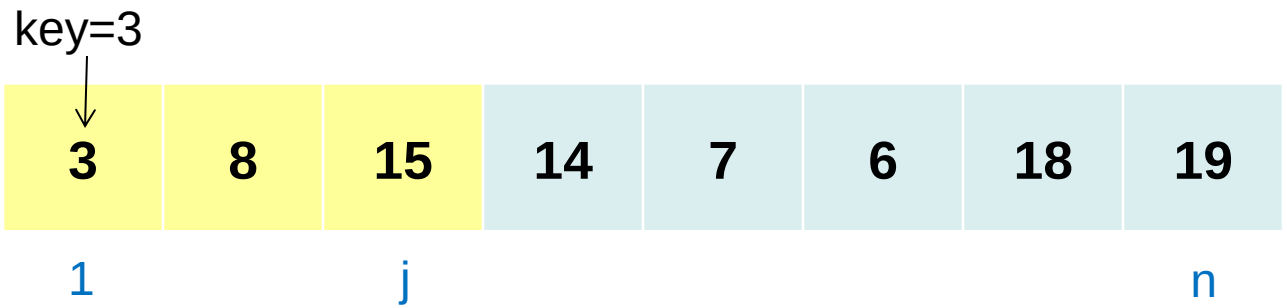
key=3

8	8	15	14	7	6	18	19
i	1		j				n

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
```

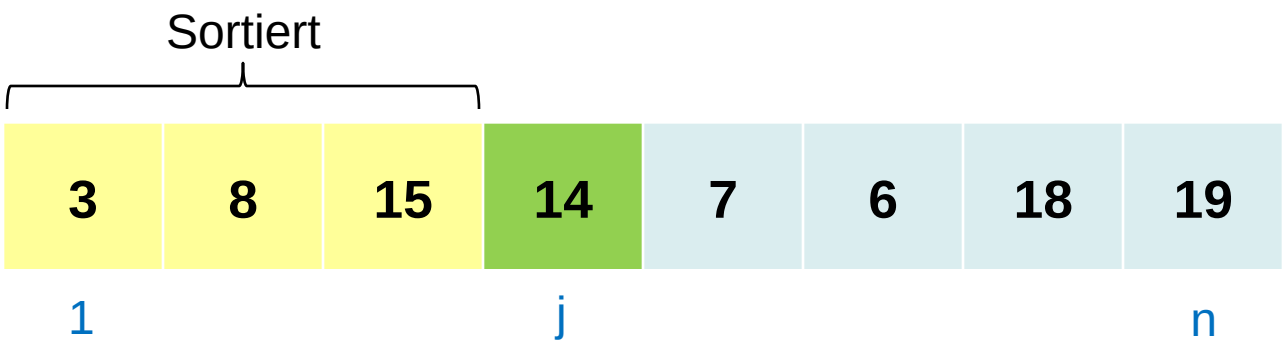
- Eingabegröße n
- length[A] = n



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
```

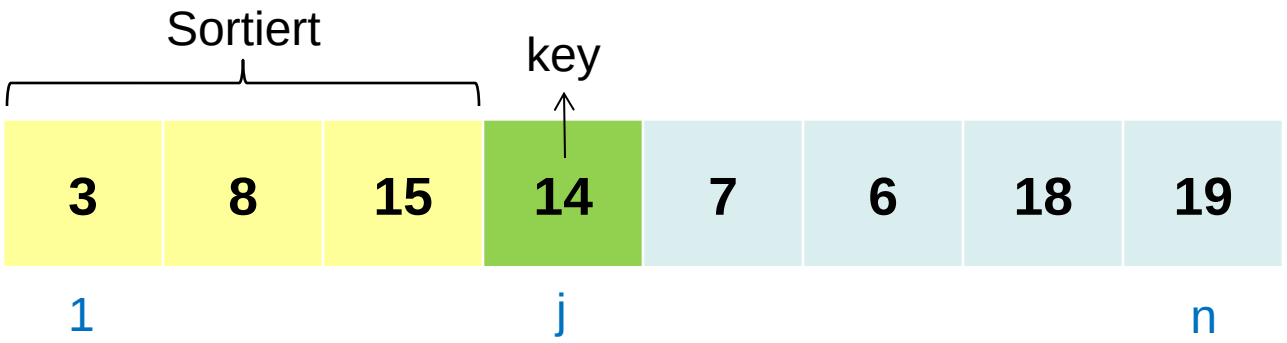
- Eingabegröße n
- length[A] = n



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
```

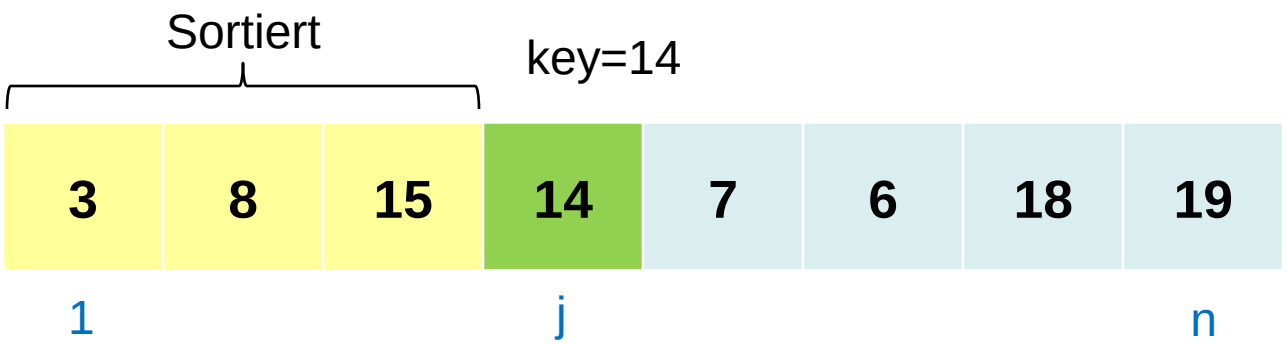
- Eingabegröße n
- length[A] = n



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

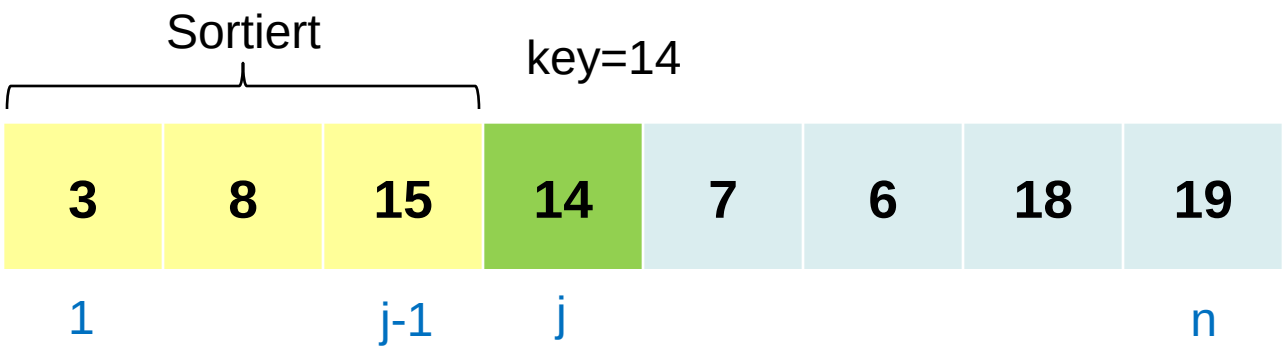


Insertion Sort

```
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

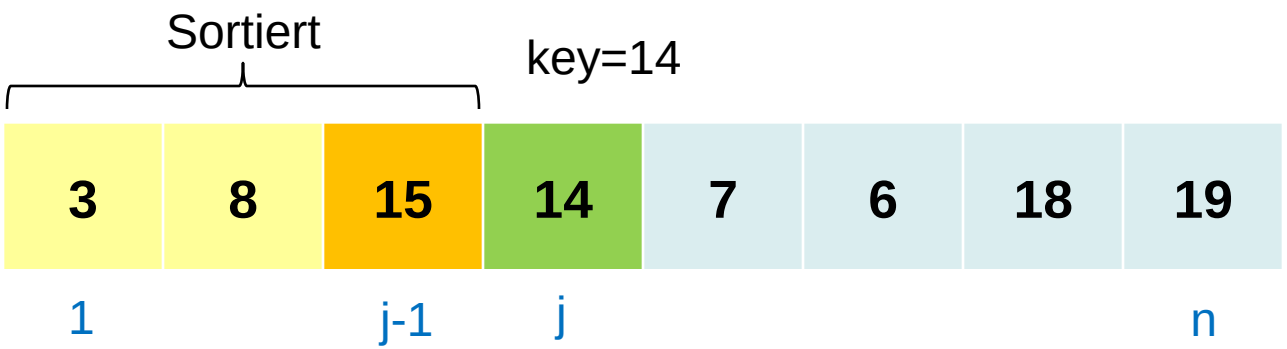


Insertion Sort

```
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



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

key=14

3	8	15	15	7	6	18	19
1	i	j-1	j				n

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

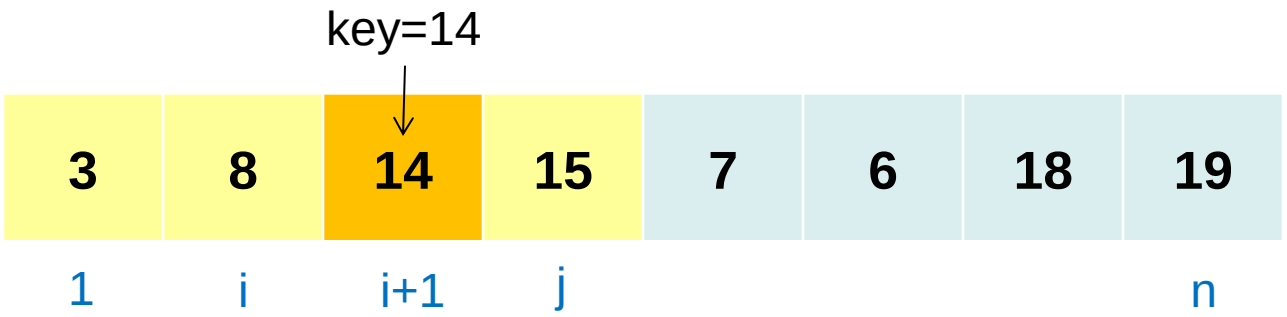
key=14

3	8	15	15	7	6	18	19
1	i	i+1	j				n

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
```

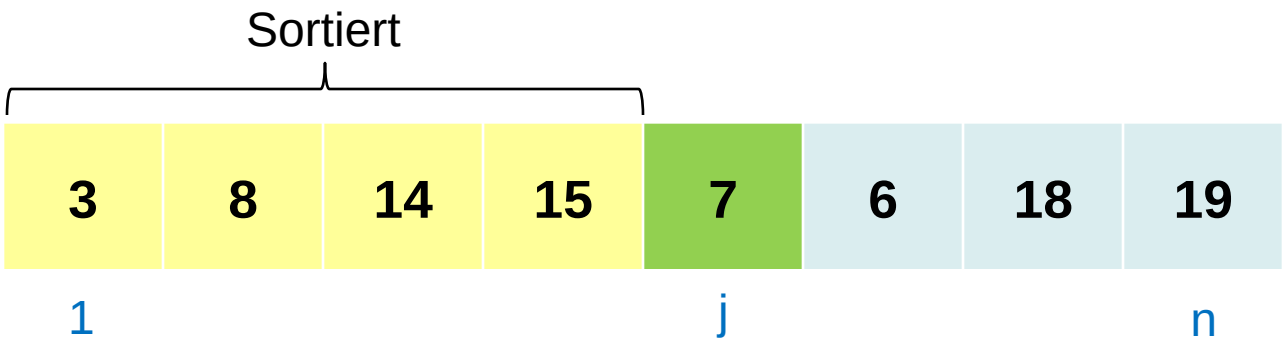
- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts
- Speichere key in Lücke



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
```

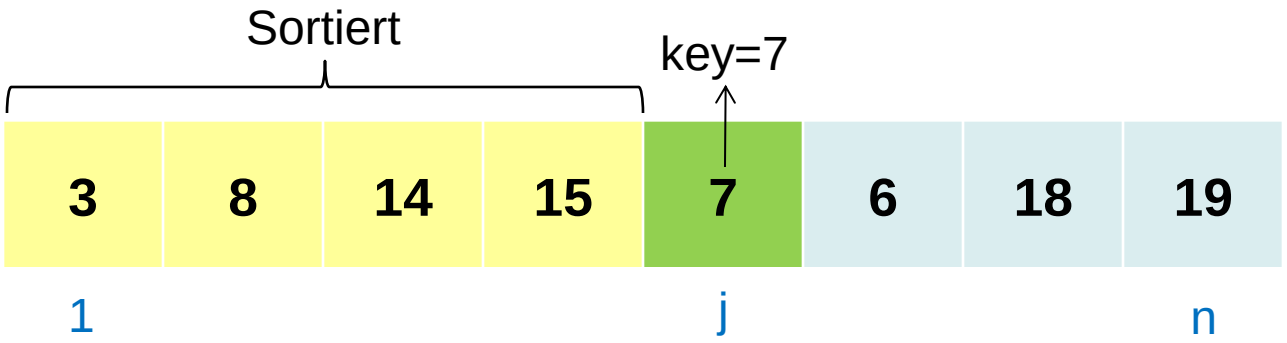
- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts
- Speichere key in Lücke



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
```

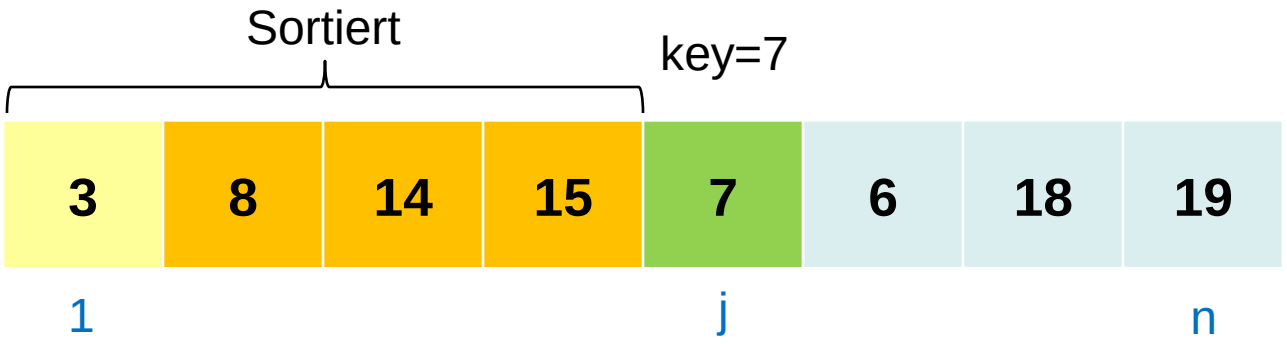
- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus A[1…j-1], die größer als key sind eine Stelle nach rechts
- Speichere key in Lücke



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
```

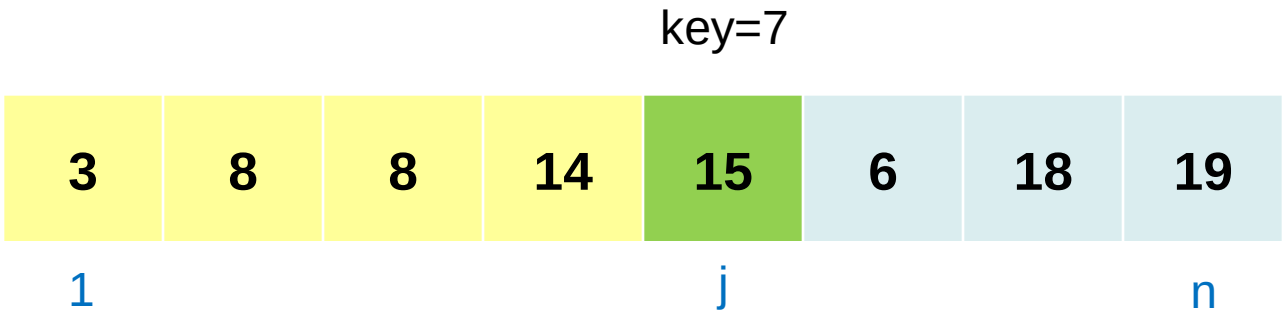
- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts
- Speichere key in Lücke



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
```

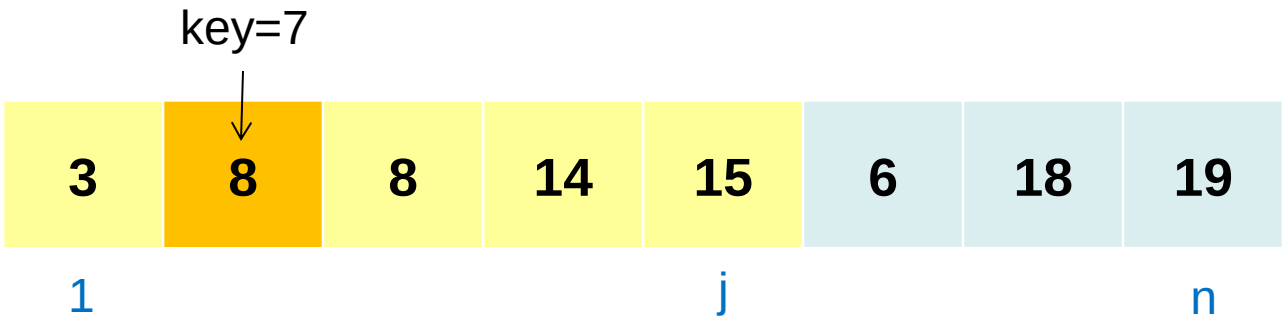
- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts
- Speichere key in Lücke



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
```

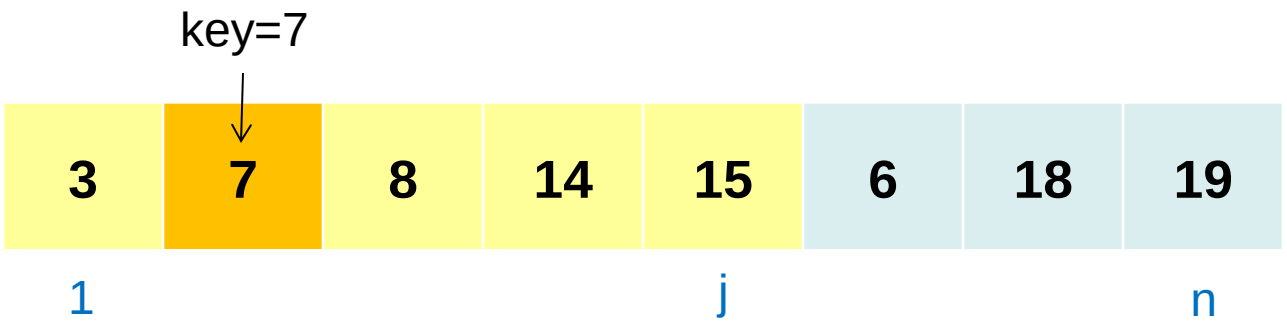
- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts
- Speichere key in Lücke



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
```

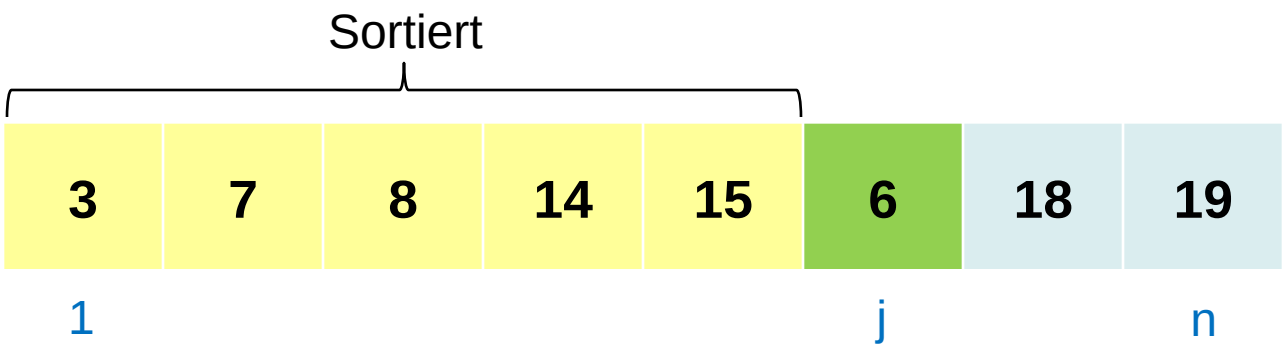
- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts
- Speichere key in Lücke



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
```

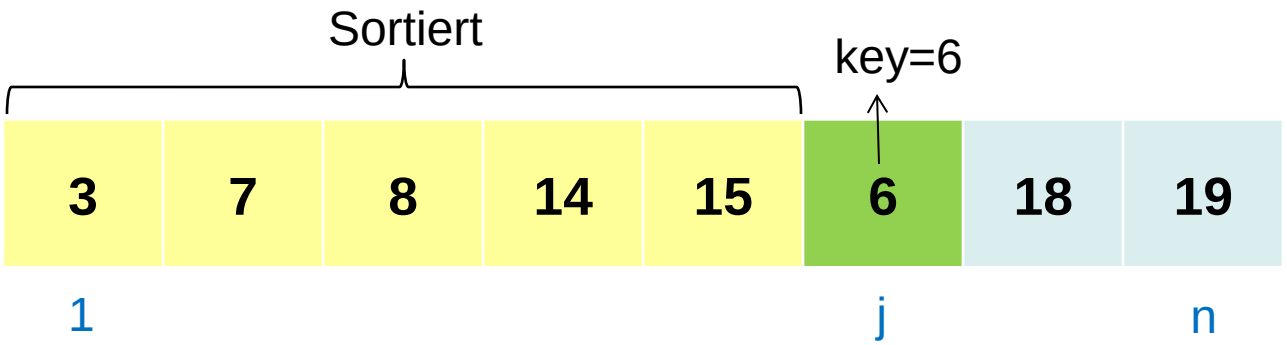
- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts
- Speichere key in Lücke



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
```

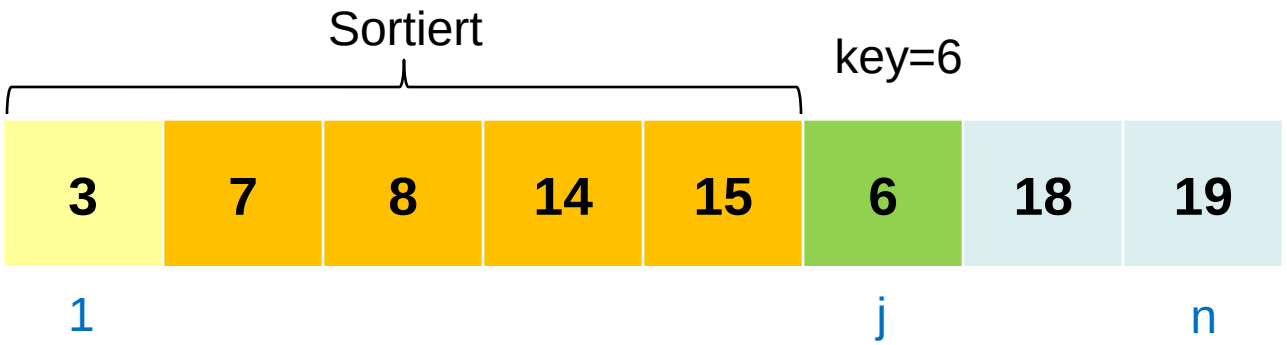
- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus A[1…j-1], die größer als key sind eine Stelle nach rechts
- Speichere key in Lücke



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
```

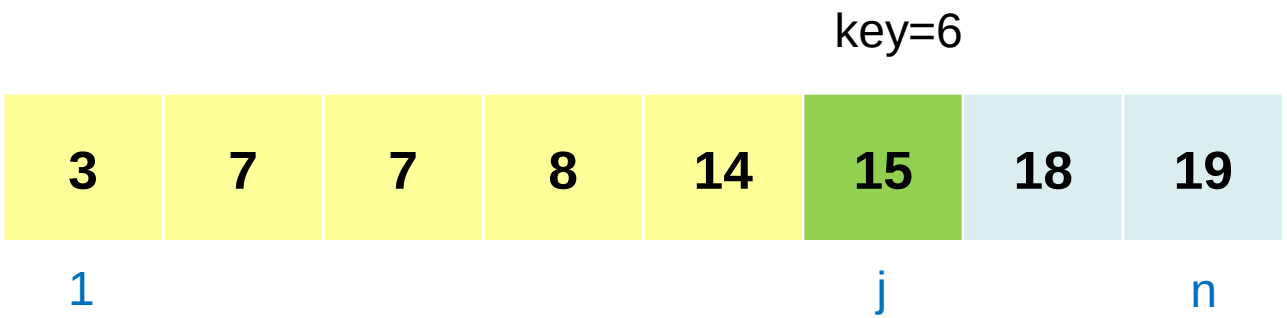
- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts
- Speichere key in Lücke



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
```

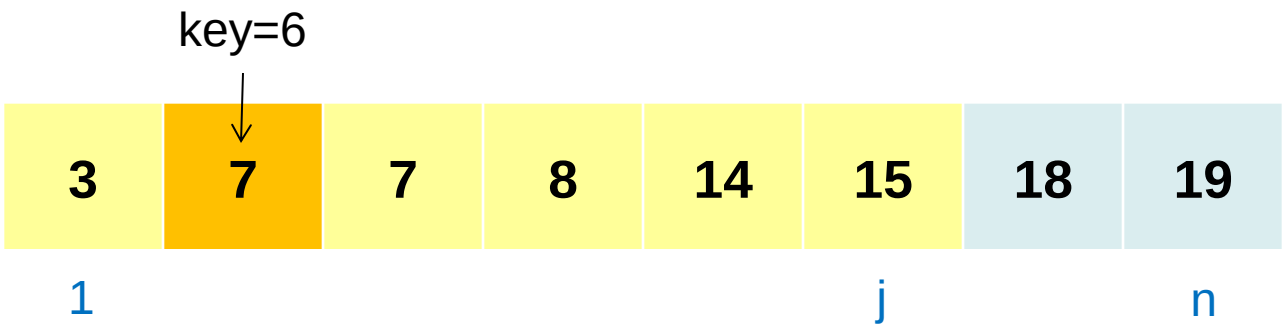
- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts
- Speichere key in Lücke



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
```

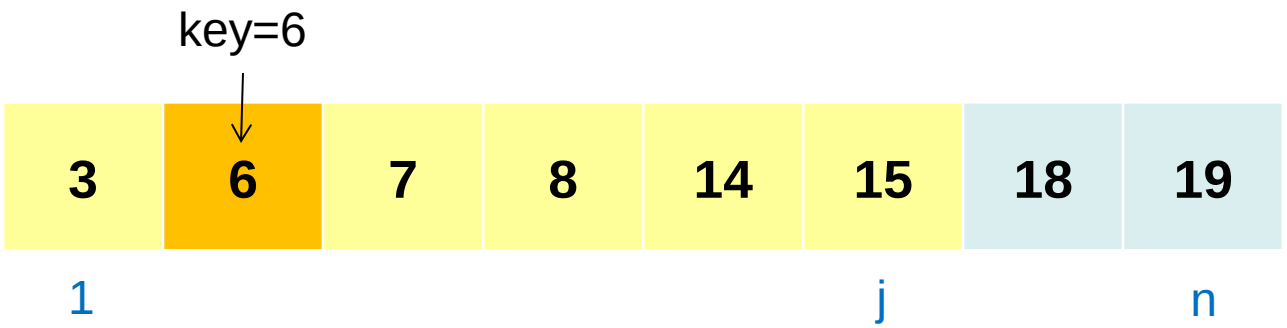
- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts
- Speichere key in Lücke



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
```

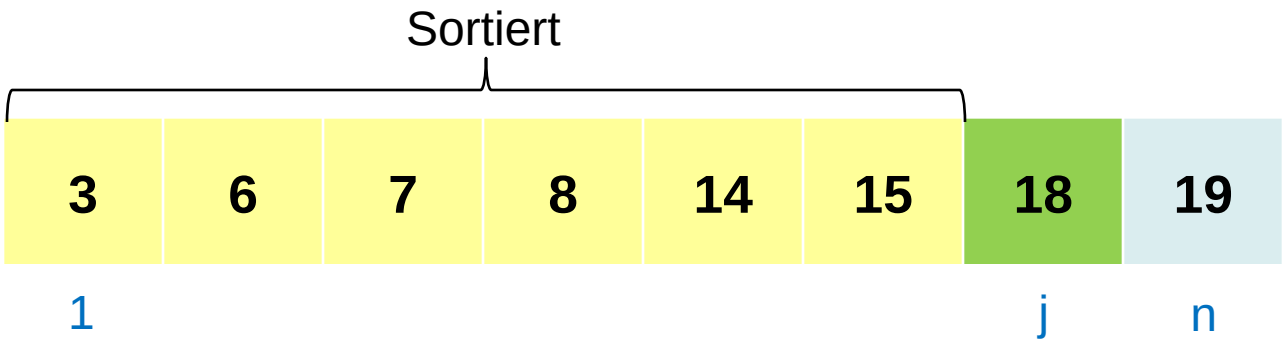
- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts
- Speichere key in Lücke



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



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
```

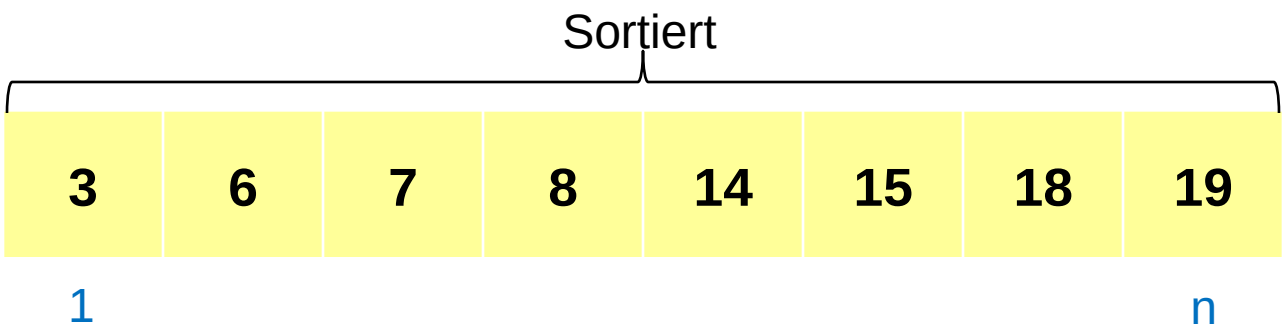
- Eingabegröße n
- length[A] = n
- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts
- Speichere key in Lücke

3	6	7	8	14	15	18	19
1						j	n

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



Laufzeitanalyse

```
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

Fragestellung

Wie kann man die Laufzeit eines Algorithmus vorhersagen?