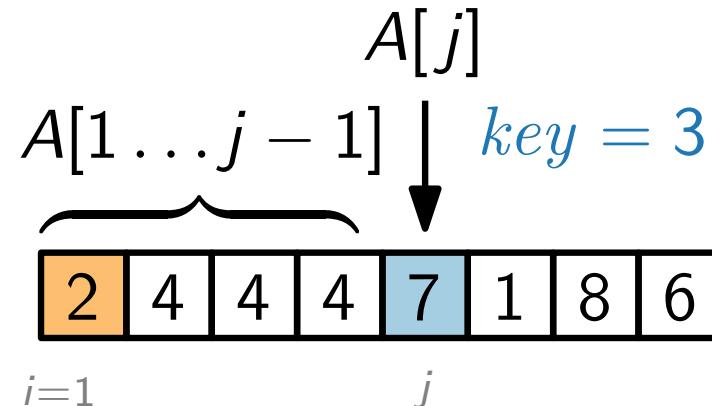
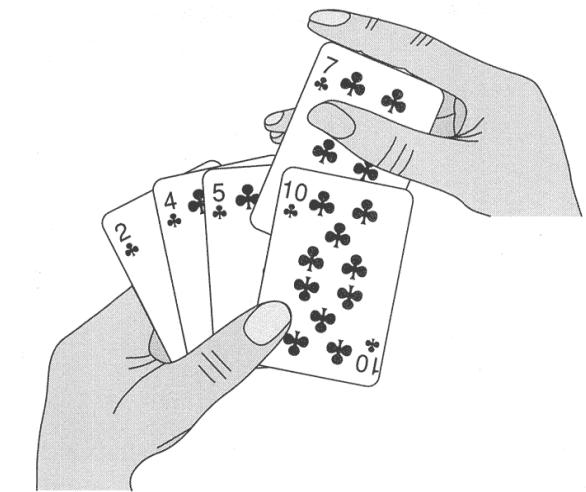




Algorithmen und Datenstrukturen



Vorlesung 1: Sortieren



Das Problem

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

Eingabe

Umordnung

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

↓

Algorithmus

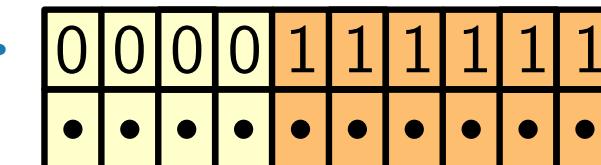
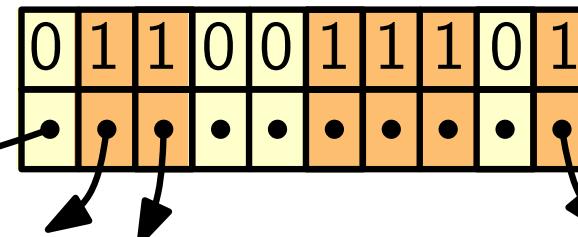
Ausgabe

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

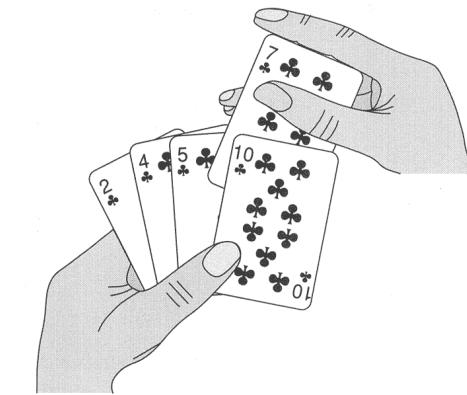
Bild von Kundan Kumar



Wie sortieren
Sie Karten?

Mögliche Lösungen

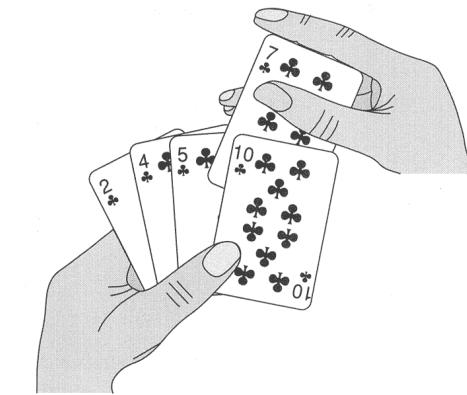
INSERTION SORT



- 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.

Mögliche Lösungen

SELECTIONSORT

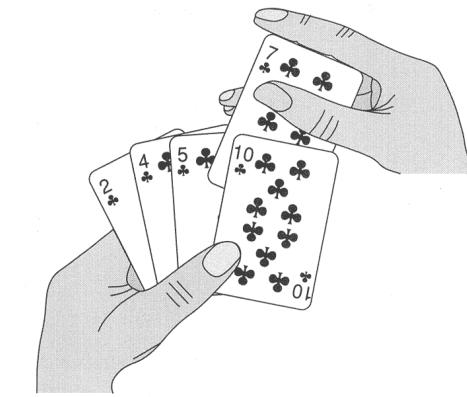


- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand sucht kleinste Karte und nimmt sie in die linke Hand.
- Linke Hand hält **immer eine sortierte Reihenfolge** aufrecht.

Mögliche Lösungen

BUBBLE SORT

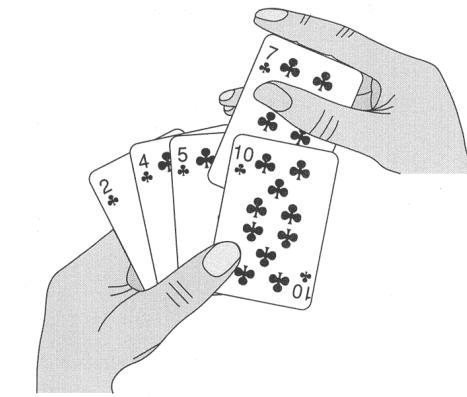
- Linke Hand hält **alle Karten**.
- Vergleiche erste mit zweiter Karte. Wenn erste Karte größer, tausche die Karten.
- Vergleiche nun zweite mit dritter Karte, usw...
- Beginne wieder von vorne und wiederhole, **bis Karten in linker Hand sortiert**.



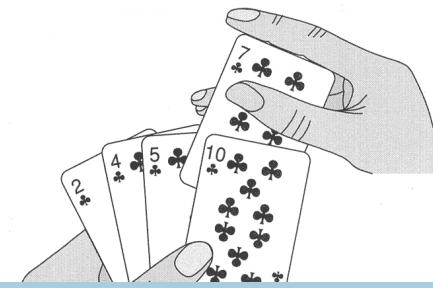
Mögliche Lösungen

BOGO SORT

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



Mögliche Lösungen



INSERTION SORT

- 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.

BUBBLE SORT

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

SELECTION SORT

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

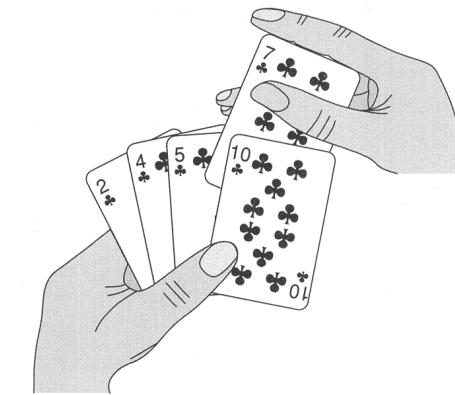
BOGO SORT

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

Inkrementelle Algorithmen

Inkrementeller Algorithmus

INSERTION SORT



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

Invariante



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

Inkrementeller Algorithmus

// In Pseudocode

INCREMENTALALG(array of ... A)

Name des Alg.

Eingabe

Variable

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

Inkrementeller Algorithmus

// In Pseudocode

INCREMENTALALG(array of ... A)

berechne Lösung für Anzahl der Elemente des Feldes A

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

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

Inkrementeller Algorithmus

// In Pseudocode

INCREMENTALALG(array of ... A)

berechne Lösung für $A[1]$ // Initialisierung
for $j = 2$ **to** $A.length$ **do** // Schleifenkopf

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

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

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

return Lösung // Ergebnisrückgabe

Inkrementeller Algorithmus

INSERTION SORT

```
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
```

Inkrementeller Algorithmus

INSERTION SORT

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

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

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

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

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

key = $A[j]$

$i = j - 1$

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

Wie verschieben wir die Einträge größer *key* nach rechts?

$A[j]$
 $A[1 \dots j - 1]$ ↓ *key* = 3

2	4	4	7	3	1	8	6
---	---	---	---	---	---	---	---

1 i j

Inkrementeller Algorithmus

INSERTION SORT

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

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

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

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

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

key = $A[j]$

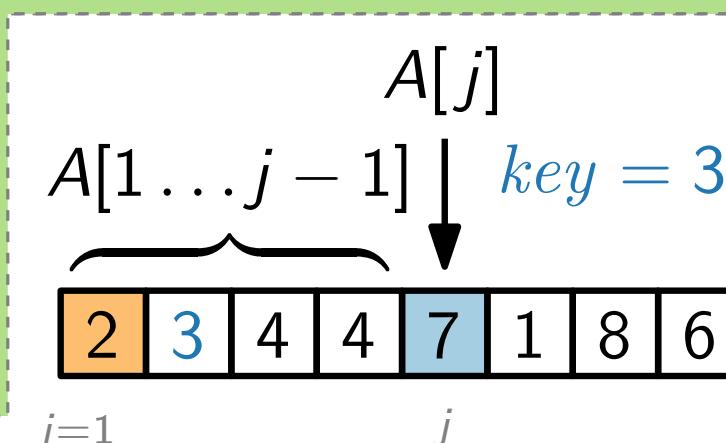
$i = j - 1$

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

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

$i = i - 1$

$A[i + 1] = key$



Fertig?

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
```

Schleifeninvariante

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

Idee der Schleifeninvariante:

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

Was?

WANTED: Bedingung, die

- an dieser Stelle immer erfüllt ist und
- bei Abbruch der Schleife Korrektheit liefert

Korrektheit beweisen

```
INSERTIONSORT(int[] A)
```

```
for j = 2 to A.length do
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Schleifeninvariante

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

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

1.) Initialisierung

Zeige: Invariante ist beim 1. Schleifendurchlauf erfüllt.

Hier: klar, denn für $j = 2$ gilt:

$A[1 \dots j - 1] = A[1 \dots 1]$ ist unverändert und sortiert.

Korrektheit beweisen

```
INSERTIONSORT(int[] A)
```

```
for j = 2 to A.length do
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Schleifeninvariante

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

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

1.) Initialisierung ✓ 2.) Aufrechterhaltung

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

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

Korrektheit beweisen

```
INSERTIONSORT(int[] A)
```

```
for j = 2 to A.length do
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Schleifeninvariante

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

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

1.) Initialisierung ✓ 2.) Aufrechterhaltung ✓

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

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

Korrektheit beweisen

```
INSERTIONSORT(int[] A)
```

```
for j = 2 to A.length do ←
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Schleifeninvariante

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

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

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

Zeige: Zusammengenommen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

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

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 $= k! = 1 \cdot 2 \cdot \dots \cdot (k-1) \cdot k$,
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  $j \leq k$  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(....)
f = 1
j = 2
while j ≤ k do
    f = f · j
    j = j + 1
return f
```

Schleifeninvariante:

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

1.) Initialisierung ✓ 2.) Aufrechterhaltung ✓

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

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

Noch ein Beispiel: Fakultät berechnen

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

Schleifeninvariante:

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

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

Zeige: Algo terminiert.

Zusammen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

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

Verletzte Schleifenbedingung: $j > k$, also $j = k + 1$.

Einsetzen von „ $j = k + 1$ “ in **Invariante** liefert $f = k!$

Noch ein Beispiel: Fakultät berechnen

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

Schleifeninvariante:

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

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

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

Selbstkontrolle

- Programmieren Sie *InsertionSort* in Java!
- Lesen Sie Kapitel 1 und Anhang A des Buchs von Cormen et al. durch und machen Sie dazu so viel Übungsaufgaben wie möglich!
- Zählen Sie die Anzahl von Vergleichsoperationen der Art „ $A[i] \leq A[j]$ “ für verschiedene Eingaben.
- 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