

Tafelübung 04

Algorithmen und Datenstrukturen

Lehrstuhl für Informatik 2 (Programmiersysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Wintersemester 2016/2017

Lehrstuhl Informatik 2
Programmiersysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Übersicht

Methoden

Rekursion

- Grundbegriffe

- Funktionsweise

- Arten der Rekursion

- Algorithmenentwurf

- Türme von Hanoi

Hinweise zu den Aufgaben

- Maze

Methoden

Lehrstuhl Informatik 2
Programmiersysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Motivation (I)

Beispiel

```
public static void main(String[] args) {  
    int j = Integer.parseInt(args[0]);  
    if (j % 4 != 0) {  
        System.out.println(false);  
    } else {  
        if (j % 400 == 0 || j % 100 != 0) {  
            System.out.println(true);  
        } else {  
            System.out.println(false);  
        }  
    }  
}
```

Was macht dieses Programm?

~> evtl. auf den ersten Blick nicht ersichtlich

Motivation (II)

Was macht *dieses* Programm?

```
public static boolean istSchaltjahr(int j) {  
    if (j % 4 != 0) {  
        return false;  
    } else {  
        if (j % 400 == 0 || j % 100 != 0) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
}  
  
public static void main(String[] args) {  
    int j = Integer.parseInt(args[0]);  
    System.out.println(istSchaltjahr(j));  
}
```

Warum Methoden?

- Methoden können ...
 - ein Programm (optisch) **gliedern** und **übersichtlicher** gestalten
 - helfen, **Code-Duplikationen** zu vermeiden

Beispiel mit Code-Duplikation

```
static int anzahlTageImJahr (int j) {  
    if (j % 4 != 0) {  
        return 365;  
    } else {  
        if (j % 400 == 0  
            || j % 100 != 0) {  
            return 366;  
        } else {  
            return 365;  
        }  
    }  
}
```

Beispiel ohne Code-Duplikation

```
static int anzahlTageImJahr (int j) {  
    if (istSchaltjahr(j)) {  
        return 366;  
    } else {  
        return 365;  
    }  
}
```

Tipp

Code in einer eigenen Methode lässt sich leicht wiederverwenden.

Nachteil von dupliziertem Code

- warum ist duplizierter Code schlecht?
 - meistens wird es **unübersichtlich**
 - Änderungen in dupliziertem Code sind **aufwändig** und **fehleranfällig**

Beispiel

Der bestehende Code soll angepasst werden. Die obige Berechnung ist nur für Jahreszahlen ab 1582 gültig, zuvor hat noch der julianische Kalender gegolten. Im julianischen Kalender werden Schaltjahre anders bestimmt.

- wenn Code **dupliziert** wurde, muss er an **mehreren Stellen** angepasst werden
~> auch die Methode `anzahlTageImMonat` muss angepasst werden!
- bei **einer** Implementierung und der Verwendung durch **Methodenaufruf** reichen hingegen Änderungen an **einer Stelle**

Rekursion

Lehrstuhl Informatik 2
Programmiersysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Rekursion (I)

Kleiner Exkurs: Rekursive Funktionen in der Mathematik

Eine **rekursive Funktion** ist eine **durch sich selbst definierte Funktion**.

Beispiel: Fakultät

- Fakultät $n!$ einer Zahl $n \in \mathbb{N}$: $n! := 1 \cdot 2 \cdot \dots \cdot n$
- rekursive Definition:
$$n! := \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

Rekursion in der Programmierung

Eine **rekursive Methode** ruft sich (unter gewissen Bedingungen) **selbst auf**.

Rekursion (II)

Von der rekursiven Definition...

$$n! := \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

...zur (möglichen) Umsetzung in Java

```
public static int fak(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fak(n-1); // rekursiver Aufruf  
    }  
}
```

Rekursion (III)

- **rekursiver Aufruf**: Aufruf einer Methode von sich selbst
 - dabei werden (fast) immer von Aufruf zu Aufruf die **Argumente verändert**
- \leadsto rekursiver Aufruf bearbeitet „**anderes**“ (meist: „kleineres“) **Problem**
- Intention:
 - **komplexes Problem** in **kleinere Probleme** aufteilen
 - Ziel: kleinere Probleme **einfacher zu lösen** als ursprüngliches Problem

Beispiel: Fakultät

$$n! := \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

Die Berechnung von $n!$ wird auf das kleinere, einfachere Problem der Berechnung von $(n - 1)!$ zurückgeführt. Für $n = 0$ ist das Problem trivial lösbar.

Basisfall \leftrightarrow Rekursionsfall (I)

- bei der Ausführung einer rekursiven Methode unterscheidet man **zwei Fälle**:
 - **Rekursionsfall**: Funktion ruft sich tatsächlich selbst auf
 - **Basisfall**: Funktion ruft sich nicht (mehr) selbst auf

Basisfall und Rekursionsfall im Beispiel

```
public static int fak(int n) {  
    if (n == 0) {  
        return 1; // Basisfall  
    } else {  
        return n * fak(n-1); // Rekursionsfall  
    }  
}
```

Basisfall \leftrightarrow Rekursionsfall (II)

Basisfall

Der Basisfall stellt die **Abbruchbedingung** dar und sorgt dafür, dass die Rekursion irgendwann abbricht. Es muss sichergestellt werden, dass der Basisfall immer nach **endlich vielen Schritten** erreicht wird, andernfalls entsteht eine **Endlos-Rekursion**.

Unsere Implementierung

```
public static int fak(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fak(n-1);  
    }  
}
```

Geht kaputt...

Wird die Methode initial mit einem **negativen Wert** für n aufgerufen, wird der **Basisfall nie erreicht**.

Besser...

Negative Startwerte im Basisfall abfangen.

Beispiel: Summe von 0 bis n

Ziel

Rekursive Methode, die die Summe von 0 bis n berechnet.

Idee

Problem auf einfacheres Problem reduzieren: Um die Summe von 0 bis n zu berechnen, berechnet man zunächst die Summe von 0 bis $(n - 1)$ und addiert anschließend n .

Mögliche Lösung

```
public static int sum(int n) {  
    if (n <= 0) { // Basisfall; negative Startwerte abfangen  
        return 0;  
    }  
    return n + sum(n-1); // Rekursionsfall  
}
```

Einschub: Methodenschachtel (vereinfacht)

- bei jedem Methodenaufruf zur **Laufzeit** des Programms:
 - Anlegen einer **Methodenschachtel** auf dem **Stack**
- Methodenschachtel beinhaltet die zur Ausführung benötigten **Daten**:
 - Werte der übergebenen **Argumente**
 - Werte **lokaler Variablen**
 - ggf. Platz für **Funktionsergebnis**
 - **Rücksprungadresse** (*von wo aus wurde die Methode aufgerufen?*)
 - ...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2      if (n <= 0) {
3          return 0;
4      }
5      int rec_res = sum(n-1);
6      return n + rec_res;
7  }
8
9  ● public static void func() {
10     sum(3);
11 }

```

func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2      if (n <= 0) {
3          return 0;
4      }
5      int rec_res = sum(n-1);
6      return n + rec_res;
7  }
8
9  public static void func() {
10 ●    sum(3);
11  }
    
```

func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1 • public static int sum(int n) {
2     if (n <= 0) {
3         return 0;
4     }
5     int rec_res = sum(n-1);
6     return n + rec_res;
7 }
8
9 public static void func() {
10     sum(3);
11 }
    
```

sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2  •    if (n <= 0) {
3        return 0;
4    }
5    int rec_res = sum(n-1);
6    return n + rec_res;
7  }
8
9  public static void func() {
10     sum(3);
11 }

```

sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2      if (n <= 0) {
3          return 0;
4      }
5  ●  int rec_res = sum(n-1);
6      return n + rec_res;
7  }
8
9  public static void func() {
10     sum(3);
11 }

```

sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1 • public static int sum(int n) {
2     if (n <= 0) {
3         return 0;
4     }
5     int rec_res = sum(n-1);
6     return n + rec_res;
7 }
8
9 public static void func() {
10     sum(3);
11 }
    
```

sum(2)	ret: 5
sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2  •    if (n <= 0) {
3        return 0;
4    }
5    int rec_res = sum(n-1);
6    return n + rec_res;
7  }
8
9  public static void func() {
10     sum(3);
11 }

```

sum(2)	ret: 5
sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2      if (n <= 0) {
3          return 0;
4      }
5  ●  int rec_res = sum(n-1);
6      return n + rec_res;
7  }
8
9  public static void func() {
10     sum(3);
11 }

```

sum(2)	ret: 5
sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1 • public static int sum(int n) {
2     if (n <= 0) {
3         return 0;
4     }
5     int rec_res = sum(n-1);
6     return n + rec_res;
7 }
8
9 public static void func() {
10     sum(3);
11 }

```

sum(1)	ret: 5
sum(2)	ret: 5
sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2  •    if (n <= 0) {
3        return 0;
4    }
5    int rec_res = sum(n-1);
6    return n + rec_res;
7  }
8
9  public static void func() {
10     sum(3);
11 }

```

sum(1)	ret: 5
sum(2)	ret: 5
sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2      if (n <= 0) {
3          return 0;
4      }
5  ●  int rec_res = sum(n-1);
6      return n + rec_res;
7  }
8
9  public static void func() {
10     sum(3);
11 }

```

sum(1)	ret: 5
sum(2)	ret: 5
sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1 • public static int sum(int n) {
2     if (n <= 0) {
3         return 0;
4     }
5     int rec_res = sum(n-1);
6     return n + rec_res;
7 }
8
9 public static void func() {
10     sum(3);
11 }

```

sum(0)	ret: 5
sum(1)	ret: 5
sum(2)	ret: 5
sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2  •    if (n <= 0) {
3        return 0;
4    }
5    int rec_res = sum(n-1);
6    return n + rec_res;
7  }
8
9  public static void func() {
10     sum(3);
11 }

```

sum(0)	ret: 5
sum(1)	ret: 5
sum(2)	ret: 5
sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2      if (n <= 0) {
3          return 0;
4      }
5      int rec_res = sum(n-1);
6      return n + rec_res;
7  }
8
9  public static void func() {
10     sum(3);
11 }

```

sum(0)	ret: 5
sum(1)	ret: 5
sum(2)	ret: 5
sum(3)	ret: 10
func()	ret: ?
...	...


Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2      if (n <= 0) {
3          return 0;
4      }
5  ●  int rec_res = sum(n-1);
6      return n + rec_res;
7  }
8
9  public static void func() {
10     sum(3);
11 }

```



sum(0)	ret: 5
sum(1)	ret: 5
sum(2)	ret: 5
sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2      if (n <= 0) {
3          return 0;
4      }
5      int rec_res = sum(n-1);
6  ●  return n + rec_res;
7  }
8
9  public static void func() {
10     sum(3);
11 }

```

sum(1)	ret: 5
sum(2)	ret: 5
sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2      if (n <= 0) {
3          return 0;
4      }
5  ●  int rec_res = sum(n-1);
6      return n + rec_res;
7  }
8
9  public static void func() {
10     sum(3);
11 }

```

1 (

sum(1)	ret: 5
sum(2)	ret: 5
sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2      if (n <= 0) {
3          return 0;
4      }
5      int rec_res = sum(n-1);
6  ●  return n + rec_res;
7  }
8
9  public static void func() {
10     sum(3);
11 }

```

sum(2)	ret: 5
sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2      if (n <= 0) {
3          return 0;
4      }
5  ●  int rec_res = sum(n-1);
6      return n + rec_res;
7  }
8
9  public static void func() {
10     sum(3);
11 }

```

3 ↻

sum(2)	ret: 5
sum(3)	ret: 10
func()	ret: ?
...	...

Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2      if (n <= 0) {
3          return 0;
4      }
5      int rec_res = sum(n-1);
6  ●  return n + rec_res;
7  }
8
9  public static void func() {
10     sum(3);
11 }

```

sum(3)	ret: 10
func()	ret: ?
...	...

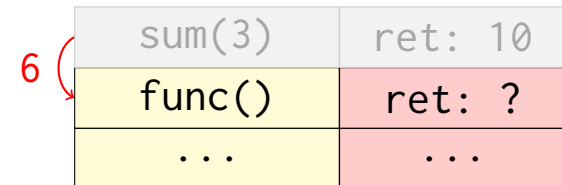
Methodenschachteln bei einer Rekursion

Code

```

1  public static int sum(int n) {
2      if (n <= 0) {
3          return 0;
4      }
5      int rec_res = sum(n-1);
6      return n + rec_res;
7  }
8
9  public static void func() {
10 ●    sum(3);
11  }

```



sum(3)	ret: 10
func()	ret: ?
...	...

Nachteile der Rekursion

- bei jedem Aufruf: **neue Methodenschachtel** auf Stack anlegen
- Stack liegt im Hauptspeicher und hat nur eine **begrenzte Größe**
- ~ bei zu vielen rekursiven Aufrufen: **StackOverflowError**
- ~ Programmausführung bricht mit **Fehlermeldung** ab
- jeder Methodenaufruf und jeder Rücksprung zum Aufrufer **kosten Zeit**
 - Kopieren der Parameter/Ergebnisse auf den Stack
 - Sprung in der Befehlsausführung
 - ...
- ~ Rekursion häufig **langsamer** als beispielsweise äquivalente Schleife

Vorteile der Rekursion

- viele Probleme sind **zu komplex**, um sie „als Ganzes“ lösen zu können
- mittels Rekursion: schrittweise Aufteilung in immer **kleinere Probleme**
 - kleinere Probleme \leadsto einfacher zu lösen
 - irgendwann: Teilprobleme **trivial** bzw. **einfach** zu lösen
 - dann: Lösungen der Teilprobleme schrittweise zur Lösung des ursprünglichen Problems **zusammenfassen**

\leadsto Grund für die Verwendung von Rekursion

Durch Rekursion lassen sich viele Probleme **eleganter** und/oder **intuitiver** lösen.

Arten der Rekursion

- man unterscheidet verschiedene **Arten der Rekursion**
 - abhängig von **Anzahl** und **Art** der rekursiven Aufrufe im Rekursionsfall
- im Folgenden lernen wir diese Arten kennen:
 - lineare Rekursion
 - Endrekursion (*tail recursion*)
 - kaskadenförmige Rekursion
 - verschachtelte Rekursion
 - verschränkte Rekursion

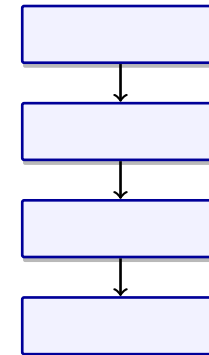
Lineare Rekursion

Lineare Rekursion

Die Funktion ruft sich im Rekursionsfall **genau einmal** selbst auf.

Beispiel

```
public static int sum(int n) {  
    if (n <= 0) {  
        return 0;  
    }  
    return n + sum(n-1);  
}
```



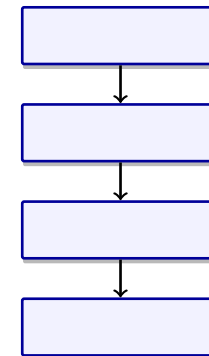
Endrekursion

Endrekursion (*tail recursion*)

Spezialfall der linearen Rekursion: im Rekursionsfall ist der (einzige) rekursive Aufruf die **letzte Aktion**. Endrekursive Funktionen lassen sich einfach **entrekursivieren**.

Beispiel

```
public static int ggt(int a, int b) {  
    if (a == b)  
        return a;  
  
    if (a < b)  
        return ggt(a, b-a);  
  
    return ggt(a-b, b);  
}
```



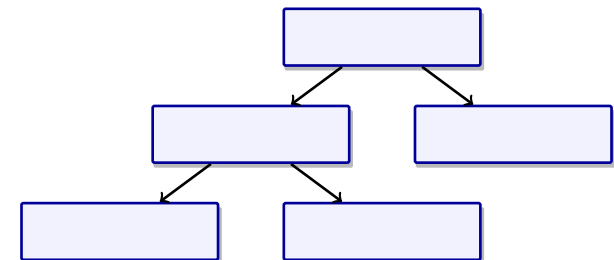
Kaskadenförmige Rekursion

Kaskadenförmige Rekursion

Die Funktion ruft sich im Rekursionsfall unter Umständen **mehrfach** selbst auf.

Beispiel

```
public static int fib(int n) {  
    if (n == 0 | n == 1) {  
        return 1;  
    }  
  
    return fib(n-1) + fib(n-2);  
}
```



Verschachtelte Rekursion

Verschachtelte Rekursion

Rekursiver Aufruf der Funktion zur Bestimmung der Parameter des rekursiven Aufrufs.
Kommt in der Praxis quasi nie vor...

Beispiel

```
public static int foo(int i) {  
    // ...  
    return foo(foo(i));  
}
```

kann man nicht
vernünftig zeichnen 😊

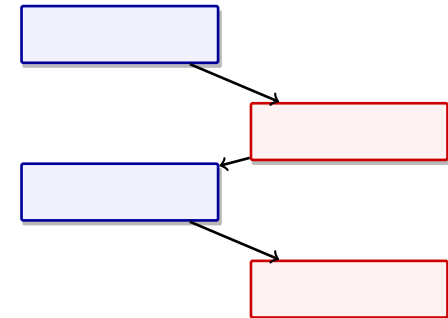
Verschränkte Rekursion

Verschränkte Rekursion

Zwei verschiedene Funktionen rufen sich im Rekursionsfall gegenseitig auf.

Beispiel

```
public static int foo(int i) {  
    // ...  
    return bar(i-1);  
}  
  
public static void bar(int i) {  
    // ...  
    return foo(i*2);  
}
```



Algorithmenentwurf: Palindrome

Palindrom

Ein Palindrom ist eine Zeichenkette, die vorwärts und rückwärts gelesen gleich ist (auch die leere Zeichenkette ist per Definition ein Palindrom).

Beispiele

- OTTO
- LEGINEINESOHELLEHOSENIENIGEL
(*leg in eine so helle Hose nie n'lgel*)

Gesucht

Rekursiver Algorithmus, der für eine gegebene Zeichenkette der Länge n entscheidet, ob es sich um ein Palindrom handelt.

Palindrome: Basis- und Rekursionsfall

Basisfall ($n = 0, n = 1$)

- $n = 0$: Leere Zeichenkette ist Palindrom
- $n = 1$: Zeichenkette mit einem Zeichen ist Palindrom

Rekursionsfall ($n > 1$)

- wenn erstes und letztes Zeichen gleich sind:
 - Palindrom, falls der „Rest“ ein Palindrom ist
- andernfalls:
 - kein Palindrom

Aufgepasst!

Der Fall n wird auf den Fall $n - 2$ zurückgeführt, also braucht man **zwei Basisfälle** (einen für gerade n und einen für ungerade n).

Palindrome: Implementierung

Mögliche Implementierung

```
// prüft Zeichenkette s auf Palindrom-Eigenschaft
public static boolean istPalindrom(String s) {
    // Basisfälle: Zeichenketten der Länge 0 / 1 sind Palindrome
    if (s.length() == 0 || s.length() == 1) {
        return true;
    }

    // Testen der Palindrom-Eigenschaft für "äußere" Zeichen
    if (s.charAt(0) != s.charAt(s.length()-1)) {
        // Zeichen sind unterschiedlich
        // -> kein Palindrom
        return false;
    }

    // Rekursionsfall: Palindrom, falls "Rest" Palindrom
    return istPalindrom(s.substring(1, s.length()-1));
}
```

Türme von Hanoi

- k Scheiben auf Position A , der Größe nach geordnet
- Positionen B und C haben keine Scheiben

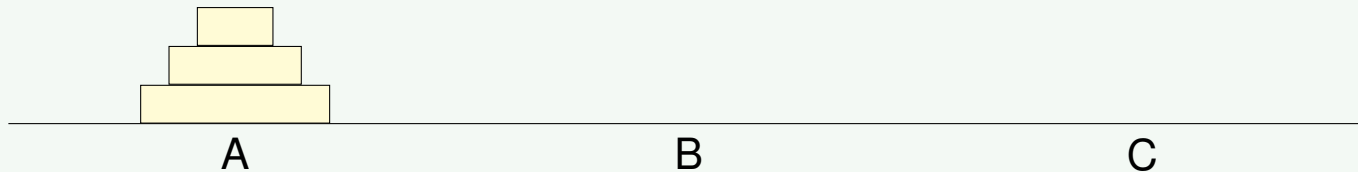


- jetzt soll der Turm von A nach B versetzt werden, wobei...
 - jede Scheibe nur einzeln bewegt werden kann
 - nur die oberste Scheibe eines Turms bewegt werden kann
 - keine größere Scheibe auf einer kleineren liegen darf
- dabei kann Position C als Zwischenablage verwendet werden

Beispiel

Beispiel: $k = 3$

Der Turm soll von A nach B bewegt werden.



Bewegungen:

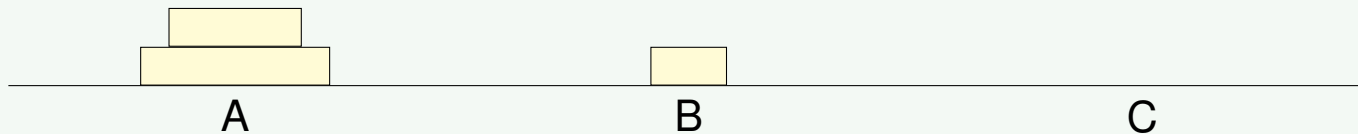
Ansatz für $k = 4$

- zuerst oberste 3 Scheiben nach C verschieben
- dann größte Scheibe von A nach B verschieben
- zuletzt Turm von C nach B verschieben

Beispiel

Beispiel: $k = 3$

Der Turm soll von A nach B bewegt werden.



Bewegungen: $A \rightarrow B$

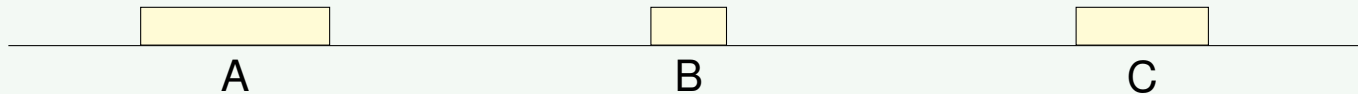
Ansatz für $k = 4$

- zuerst oberste 3 Scheiben nach C verschieben
- dann größte Scheibe von A nach B verschieben
- zuletzt Turm von C nach B verschieben

Beispiel

Beispiel: $k = 3$

Der Turm soll von A nach B bewegt werden.



Bewegungen: $A \rightarrow B$ $A \rightarrow C$

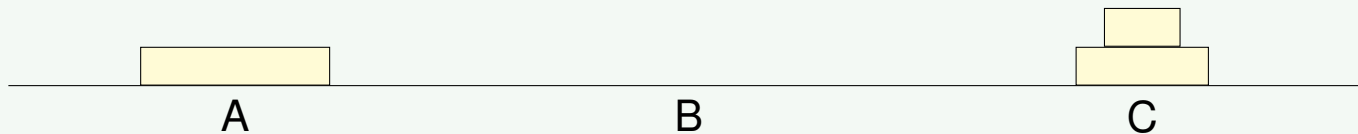
Ansatz für $k = 4$

- zuerst oberste 3 Scheiben nach C verschieben
- dann größte Scheibe von A nach B verschieben
- zuletzt Turm von C nach B verschieben

Beispiel

Beispiel: $k = 3$

Der Turm soll von A nach B bewegt werden.



Bewegungen: $A \rightarrow B$ $A \rightarrow C$ $B \rightarrow C$

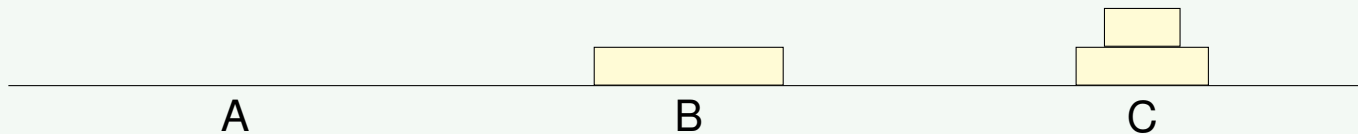
Ansatz für $k = 4$

- zuerst oberste 3 Scheiben nach C verschieben
- dann größte Scheibe von A nach B verschieben
- zuletzt Turm von C nach B verschieben

Beispiel

Beispiel: $k = 3$

Der Turm soll von A nach B bewegt werden.



Bewegungen: $A \rightarrow B$ $A \rightarrow C$ $B \rightarrow C$ $A \rightarrow B$

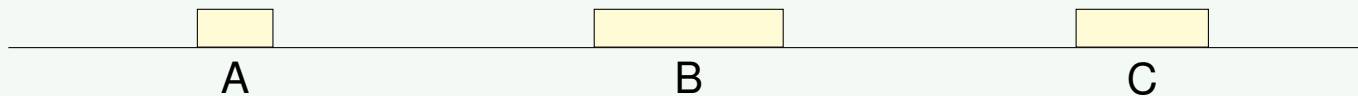
Ansatz für $k = 4$

- zuerst oberste 3 Scheiben nach C verschieben
- dann größte Scheibe von A nach B verschieben
- zuletzt Turm von C nach B verschieben

Beispiel

Beispiel: $k = 3$

Der Turm soll von A nach B bewegt werden.



Bewegungen: $A \rightarrow B$ $A \rightarrow C$ $B \rightarrow C$ $A \rightarrow B$ $C \rightarrow A$

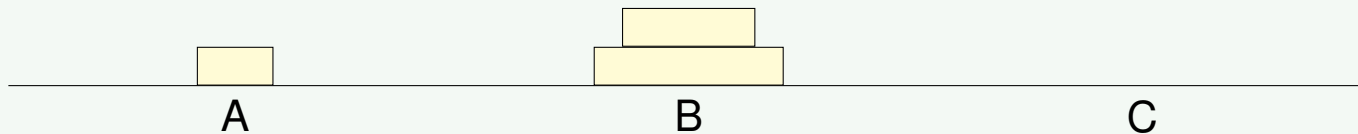
Ansatz für $k = 4$

- zuerst oberste 3 Scheiben nach C verschieben
- dann größte Scheibe von A nach B verschieben
- zuletzt Turm von C nach B verschieben

Beispiel

Beispiel: $k = 3$

Der Turm soll von A nach B bewegt werden.



Bewegungen: $A \rightarrow B$ $A \rightarrow C$ $B \rightarrow C$ $A \rightarrow B$ $C \rightarrow A$ $C \rightarrow B$

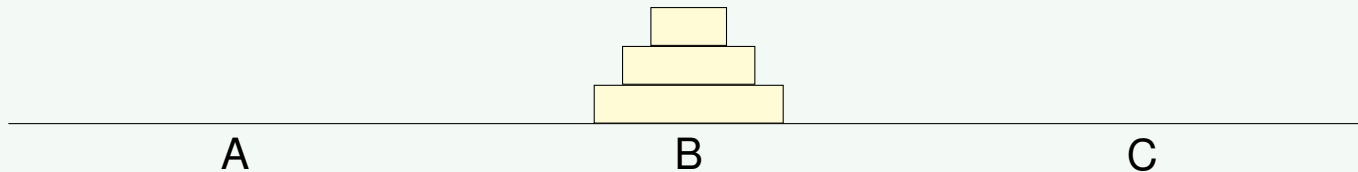
Ansatz für $k = 4$

- zuerst oberste 3 Scheiben nach C verschieben
- dann größte Scheibe von A nach B verschieben
- zuletzt Turm von C nach B verschieben

Beispiel

Beispiel: $k = 3$

Der Turm soll von A nach B bewegt werden.



Bewegungen: $A \rightarrow B$ $A \rightarrow C$ $B \rightarrow C$ $A \rightarrow B$ $C \rightarrow A$ $C \rightarrow B$ $A \rightarrow B$

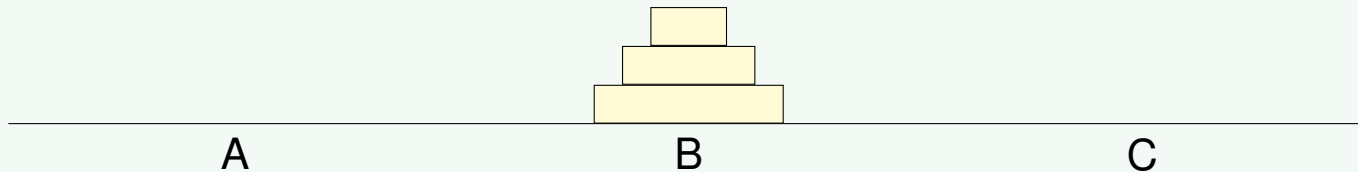
Ansatz für $k = 4$

- zuerst oberste 3 Scheiben nach C verschieben
- dann größte Scheibe von A nach B verschieben
- zuletzt Turm von C nach B verschieben

Beispiel

Beispiel: $k = 3$

Der Turm soll von A nach B bewegt werden.



Bewegungen: $A \rightarrow B$ $A \rightarrow C$ $B \rightarrow C$ $A \rightarrow B$ $C \rightarrow A$ $C \rightarrow B$ $A \rightarrow B$

Ansatz für $k = 4$

- zuerst oberste 3 Scheiben nach C verschieben
- dann größte Scheibe von A nach B verschieben
- zuletzt Turm von C nach B verschieben

Hinweise zu den Aufgaben

Lehrstuhl Informatik 2
Programmiersysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

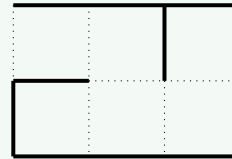
TECHNISCHE FAKULTÄT

Aufgabe „Maze“

- Ziel: Weg durch ein Labyrinth vom Ausgang zum Ziel finden
 - das Labyrinth ist als dreidimensionales boolean-Array `maze` gegeben
 - die erste Dimension ist die Zeile (wächst von oben nach unten)
 - die zweite Dimension ist die Spalte (wächst von links nach rechts)
 - in der dritten Dimension steht, in welchen Himmelsrichtungen (NORTH, EAST, SOUTH, WEST) das entsprechende Feld eine Wand hat
- ↪ `maze[z][s][h]` gibt an, ob
- in der Zeile `z` und Spalte `s`
 - in der Himmelsrichtung `h`
- eine Wand existiert (`true`) oder nicht (`false`)

Aufgabe „Maze“: Beispiel

Beispiel für ein 2×3 -Labyrinth



Wände für das obere rechte Feld ausgeben:

```
System.out.println(maze[0][2][NORTH]); //true
System.out.println(maze[0][2][EAST]);  //true
System.out.println(maze[0][2][SOUTH]); //false
System.out.println(maze[0][2][WEST]);  //true
```

Fragen? Fragen!

(hilft auch den anderen)

Lehrstuhl Informatik 2
Programmiersysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT