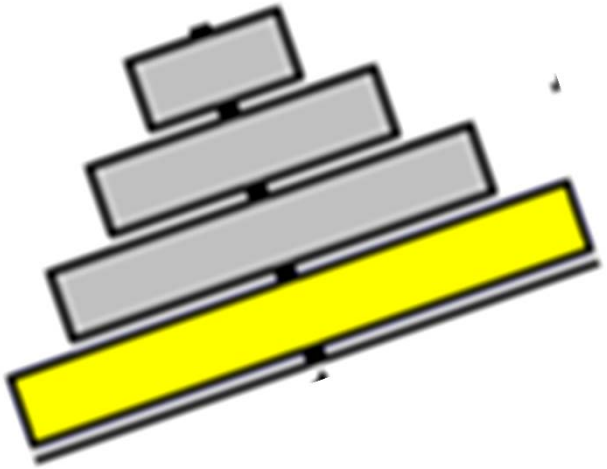
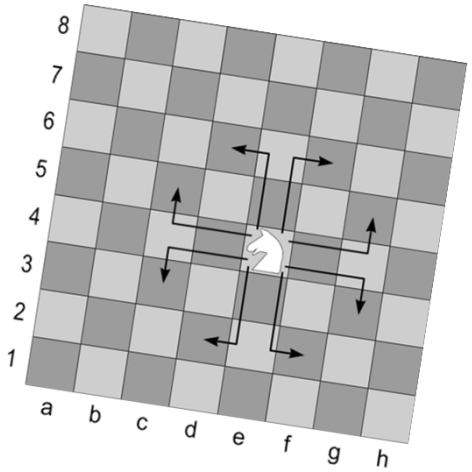
Vorlesung Programmieren
Thema 5: Rekursion und Backtracking

Olaf Herden

Fakultät Technik

Studiengang Informatik





Stand: 11/2023

Gliederung

- Rekursion
- Backtracking

Rekursion

- Definition eines Problems/Verfahrens/Funktion durch sich selbst
- Direkte Rekursion: Methode ruft sich selbst
- Indirekte Rekursion: Zwei (oder mehr Methoden) rufen sich wechselseitig auf
- Beispiel: Berechnung der Fakultät
  - Rekursive Problembeschreibung:
    - Fakultät von von 0 ist 1
    - Fakultät einer Zahl $n \geq 1$ ist die Fakultät der Zahl n-1 multipliziert mit n
  - Iterative Problembeschreibung:
    - Fakultät ist das Produkt der Zahlen von 1 bis n
- Anschaulich: Rekursion zerlegt Problem so lange in kleineres Problem, bis es trivial ist

Fakultät (I)

- Mathematische Beschreibung:
  - Iterative Lösung: $f_{iter}(n) := n! = \prod_{i=1}^{n} i$
  - Rekursive Lösung: $f_{rec}(n) := n! = \begin{cases} f_{rec}(n-1) \cdot n, falls\ n \geq 1 \\ 1, sonst \end{cases}$

- Implementierung:
  - Iterative Lösung:

```
(1) public static int f_iter(int n){
(2)   int erg = 1;
(3)   for (int i=1;i<=n;i++){
(4)     erg = erg *i;
(5)   }
(6)   return erg;
(7) }
```

Fakultät (II)

- Rekursive Lösung:

```
(1) public static int f_rec(int n){
(2)   if (n >= 1){
(3)     return f_rec(n-1)*n;
(4)   }
(5)   else{
(6)     return 1;
(7)   }
(8) }
```

Iterativ vs. rekursiv

- Rekursive Darstellungen meistens kürzer und leichter zu verstehen
- Effizienz/Rechenaufwand:
  - Keine pauschale Aussage möglich:
    - Manchmal sehr effizient (z.B. Quicksort, Baumtraversierungen (siehe Informatik-VL))
    - Manchmal jedoch extrem ineffizient (siehe Übung Fibonacci-Folge)

Rekursionstiefe

- Anzahl geschachtelter Aufrufe bei Rekursion
- Beispiel: Rekursive Berechnung 5!

Aufrufe für f_rec(5)	Rekursionstiefe
Aufruf: f_rec(5)	0
Aufruf: f_rec(4)	1
Aufruf: f_rec(3)	2
Aufruf: f_rec(2)	3
Aufruf: f_rec(1)	4
Aufruf: f_rec(0)	5

Rekursionsabbruch

- Wichtig bei Rekursion ist Abbruchbedingung (synonym: Rekursionsanker, trivialer Fall)
- Sicherstellung, dass Rekursion „irgendwann mal“ endet, d.h. kein weiterer rekursiver Aufruf stattfindet (d.h. keine Erhöhung der Rekursionstiefe)
- Beispiel:

```
(1) public static int f_rec(int n){
(2)   if (n >= 1){
(3)     return f_rec(n-1)*n;
(4)   }
(5)   else{
(6)     return 1;
(7)   }
(8) }
```

- Dekrementierung n bei jedem rekursiven Aufruf $\Rightarrow$ n irgendwann 1 (Annahme: Positiver Wert wurde übergeben)
- Fall $n = 1$ ist Rekursionsabbruch
- Bei

Endlosrekursion (I)

- Ansonsten spricht man von Endlosrekursion
- Beispiel:

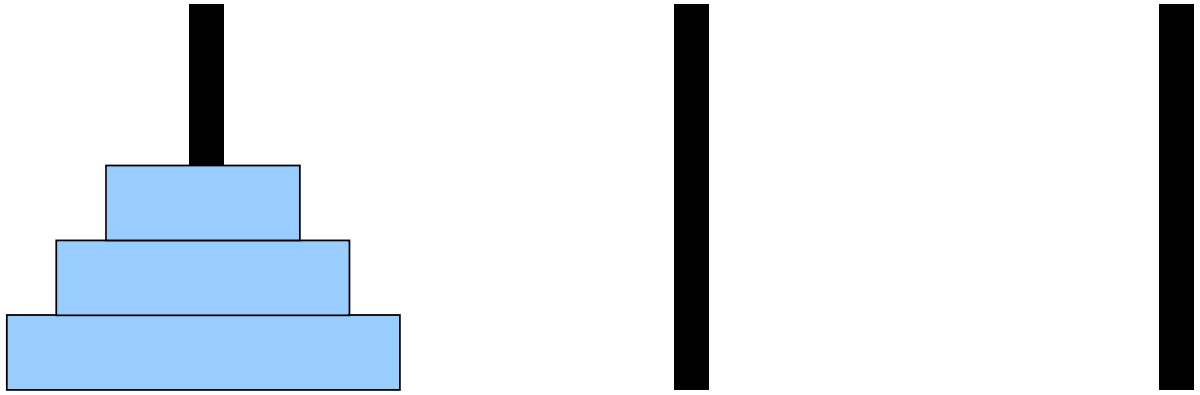
```
(1) public static void f_rec_endless(){
(2)    ...
(3)    f_rec_endless();
(4) }
```

- Führt zu einem Systemabsturz (bzw. bei Java zu einer Exception)
- Ursache: Allokation auf Stack bei jedem rekursiven Aufruf, irgendwann Überlauf

```
Depth of recursion : 4747
Depth of recursion : 4748
Depth of recursion : 4749
Depth of recursion : 4750
Depth of recursion : 4751
Exception in thread "main" java.lang.StackOverflowError
      at java.base/java.nio.CharBuffer.limit(CharBuffer.java:1565)
      at java.base/java.nio.CharBuffer.limit(CharBuffer.java:285)
      at java.base/java.nio.Buffer.<init>(Buffer.java:256)
      at java.base/java.nio.CharBuffer.<init>(CharBuffer.java:316)
      at java.base/java.nio.HeapCharBuffer.<init>(HeapCharBuffer.java:85)
```

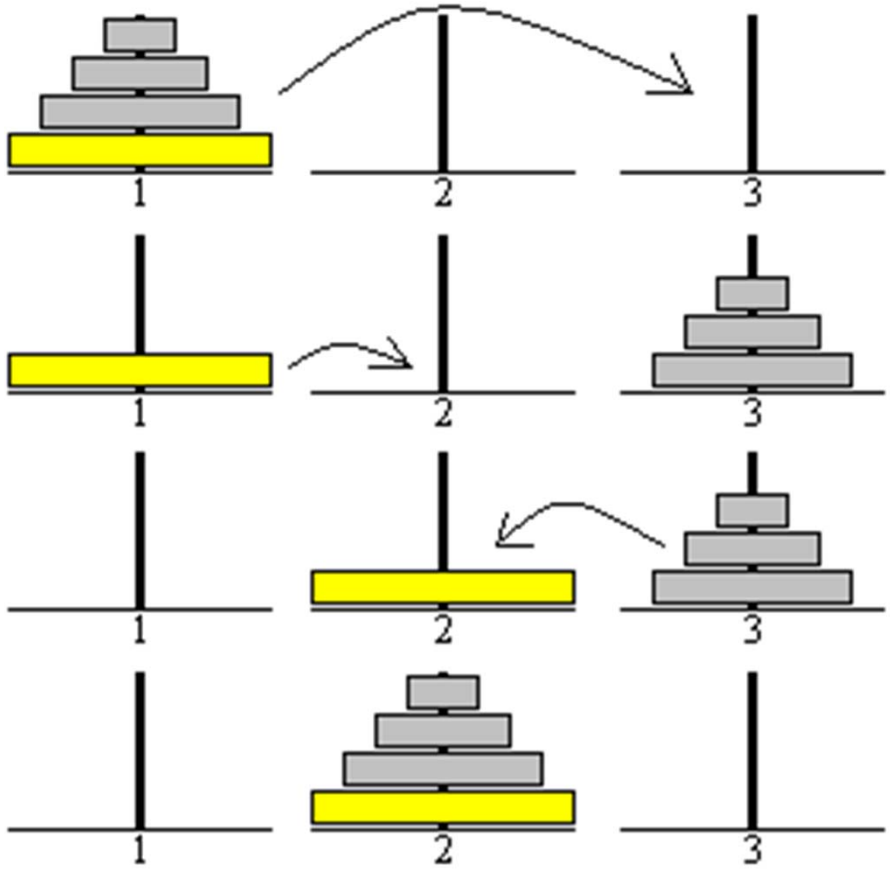
Türme von Hanoi (I)

- Klassisches Problem zur Rekursion:
  - Drei Plätze zum Stapeln von Scheiben
  - n Scheiben unterschiedlichen Durchmessers, der Größe nach zu Turm geschichtet
  - Unterste Scheibe ist die größte
  - Turm anfangs auf Platz 1 (Start), soll unter Verwendung vom Platz 3 (Hilfsplatz) auf Platz 2 (Zielplatz) bewegt werden
  - Randbedingungen:
    - In einem Zug nur oberste Scheibe eines Turmes bewegen
    - Zu keinem Zeitpunkt darf größere Scheibe kleinerer liegen



Türme von Hanoi (II)

- Wünschenswerter Ablauf:



Gesamtproblem P:
„Bewege n Scheiben von Platz 1 nach Platz 2“

Teilproblem P_1:
„Bewege $n-1$ Scheiben von Platz 1 nach Platz 3“

Teilproblem P_2:
„Bewege letzte Scheibe von Platz 1 nach Platz 2“

Teilproblem P_3:
„Transportiere $n-1$ Scheiben von Platz 3 nach Platz 2“

[http://www.mathematische-basteleien.de/]

Türme von Hanoi (III): Implementierung

P_1 = „Bewege $n-1$ Scheiben von Platz 1 nach Platz 3“

P_2 = „Bewege letzte Scheibe von Platz 1 nach Platz 2“

P_3 = „Transportiere $n-1$ Scheiben von Platz 3 nach Platz 2“

```
(1) static void hanoi(int n,int von,int nach ,int ueber){
(2)   if (n > 0){
(3)     hanoi(n-1,von,ueber,nach);
(4)     System.out.println
(5)       ("Scheibe " + n + " von Platz " + von + " nach Platz " + nach);
(6)     hanoi(n-1,ueber,nach,von);
(7)   }
(8) }
```

Türme von Hanoi (IV): Zugzahl

- Messung von Zügen (Scheibenbewegungen):

Scheiben	Züge
1	1
2	3
3	7
4	15
5	31
6	63
7	127
8	255
9	511
10	1023

Scheiben	Züge
11	2047
12	4095
13	8191
14	16.383
15	32.767
16	65.535
17	131.071
18	262.143
19	524.287
20	1.048.575

Allgemein: $2^n - 1$ Züge

Türme von Hanoi (V): Laufzeiten

- Messung von Laufzeiten

Scheiben	Zeit Durchschnitt[sec]	Einzelwerte [sec]
1	0	0
…		…
27	0	0
28	0,5	1/1/0/0
29	0,5	1/1/0/0
30	3,2	4/4/2/3/3
31	3,2	4/3/3/3/3
32	12,6	17/12/11/11/12
33	13,4	17/13/12/12/13
34	51,6	69/43/47/49/50
35	56,4	78/50/50/52/52

- (Stark) steigende Laufzeit
- Allerdings:
  - Absolute Laufzeiten wenig aussagekräftig
  - Abhängig von Rechner
- Besser: Komplexität angeben

$\Rightarrow$ Siehe Informatik-Vorlesung

Ackermann-Funktion

- Definition von Paul Ackermann (1926), modifiziert von Rosza Peter (1935):
  $a(0,m) = m+1$
  $a(n+1,0) = a(n,1)$
  $a(n+1,m+1) = a(n,a(n+1,m))$
- In Java:

```
(1) static long acker(long n,long m){
(2)   if (n == 0){return m+1;}
(3)   if (m == 0){return acker(n-1,1);}
(4)   return acker(n-1, acker(n,m-1));
(5) }
```

- Extrem stark wachsende Anzahl an Berechnungsschritten, z.B. acker(4,4) führt schon zu Stackoverflow
- Wichtige Bedeutung in Theoretischer Informatik (siehe Informatik III-Vorlesung)

Übersicht

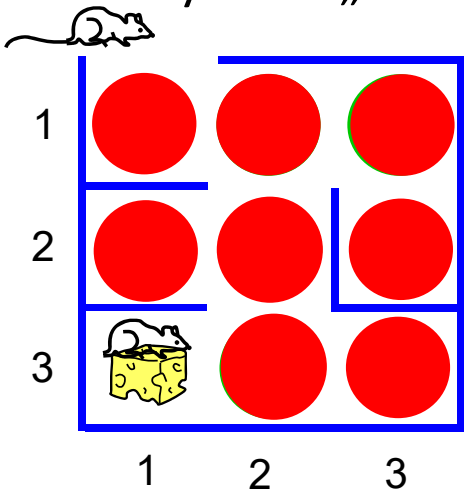
- Rekursion
- Backtracking

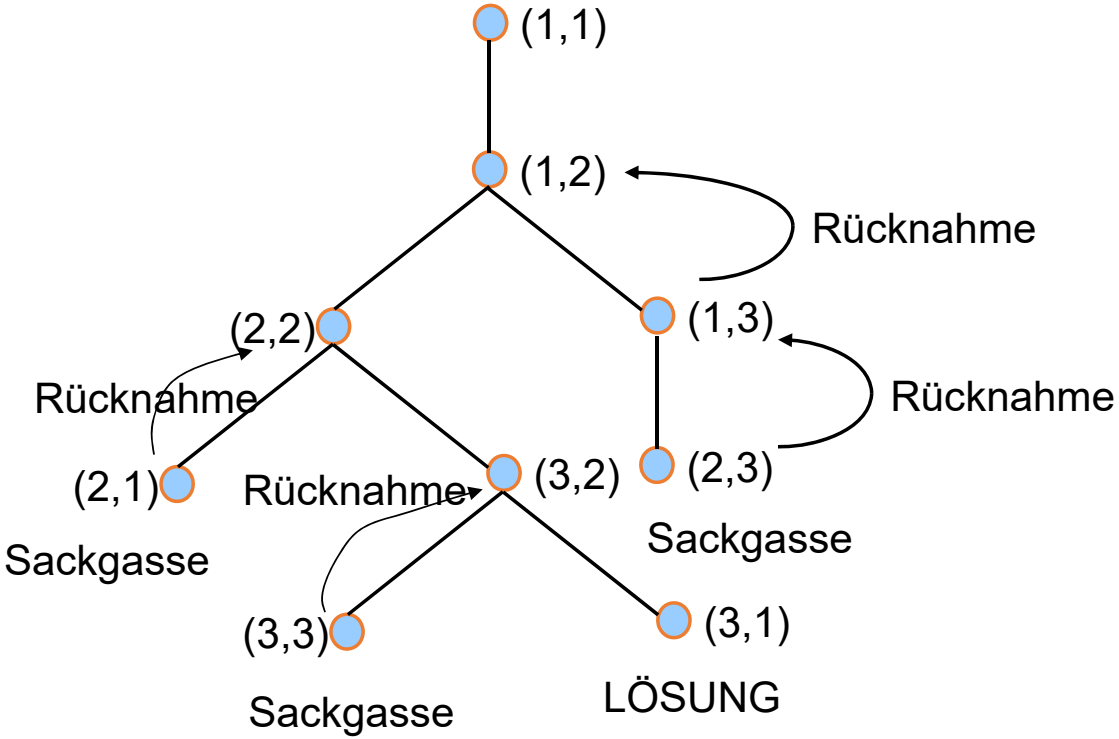
Idee

- Synonym: Trial and Error-Verfahren
- Vorgehen:
  - Systematisches Durchprobieren aller Möglichkeiten (Teilschritt)
  - Wenn keine weitere Möglichkeit gegeben und Ziel noch nicht erreicht (Sackgasse) $\Rightarrow$ Rücknahme von Teilschritten, bis noch alternative Möglichkeit vorhanden
  - Probiere nun alternativen Teilschritt
  - Zurücknehmen von Teilschritten und erneutes Vorangehen wiederholen, bis Lösung gefunden oder Nicht-Lösbarkeit festgestellt

Beispiel (I): Labyrinth

- Suchen im Labyrinth: „Wie kommt die Maus zum Käse?“



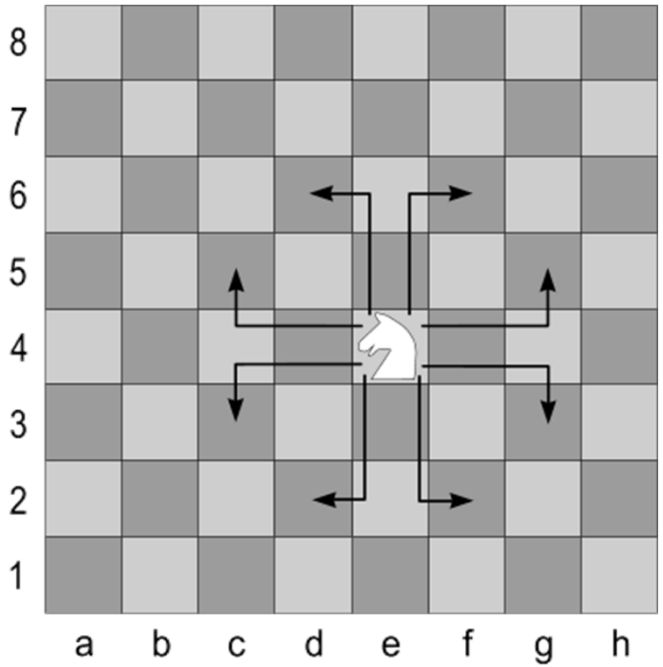


Schon gewesen

Rücknahmen möglich

Beispiel (II): Springerproblem

- Springer auf beliebigen Ausgangsfeld auf Schachbrett
- Soll nacheinander alle Felder des Bretts genau einmal besuchen
- Gangart des Springers:

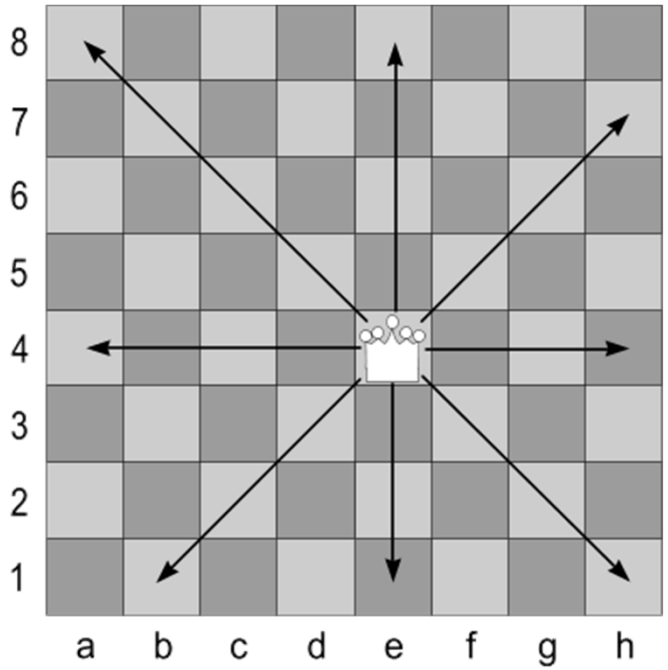


- Implementierung: Siehe Programm

[https://www.schach-lernen.de/blog/323/die-gangart-der-figuren-der-springer/]

Beispiel (III): 8-Damen-Problem

- Platzierung von 8 Damen auf Schachbrett, so dass sie sich nicht gegenseitig schlagen können
- Gangart der Dame:



- Verallgemeinerung: N-Damen-Problem
- Implementierung: Siehe Übung

[https://www.schach-lernen.de/blog/338/die-gangart-der-figuren-die-dame/]

Zusammenfassung

- Rekursion:
  - Defintion
  - Bsp. Fakultätsberechnung
  - Iteration vs. Rekursion
  - Rekursionstiefe
  - Abbruchbedingung (Basisfall, Rekursionsanker)
  - Endlosrekursion
  - Türme von Hanoi

- Backtracking:
  - Definition
  - Labyrinth
  - Springerproblem
  - Damenproblem

Aufgaben (I)

- **Aufgabe 1**

  Die Fibonacci-Zahlenfolge ist so definiert, dass das erste und zweite Folgenglied jeweils 1 ist. Ab dem dritten Glied der Folge gilt, dass sich der Wert aus der Summe der beiden vorherigen Werte berechnet. Damit ergibt sich folgende Zahlenfolge: 1,1,2,3,5,8,13,21, ...

  a) Schreiben Sie eine Methode, die die Fibonaccifolge iterativ berechnet!

  b) Schreiben Sie eine Methode, die die Fibonaccifolge rekursiv berechnet!

  c) Rufen Sie in der `main()`-Methode beide Methoden für die Werte von 1 bis 45 auf. Was passiert? Wie erklären Sie sich dieses Verhalten?

- **Aufgabe 2**

  Realisieren Sie das 8-Damen-Problem als Java-Programm!

Aufgaben (II)

- **Aufgabe 3**

a) Implementieren Sie eine Methode, die endlos rekursiv ist. Im Rumpf der Methode soll die Anzahl der (Selbst)aufrufe ausgegeben werden. Leiten Sie die Ausgabe in eine Datei um. Wie viele Iterationen „schafft“ die Methode, bevor es zum Speicherüberlauf kommt?

b) Mit der Option –Xss<size> lässt sich die Stack-Größe verändern. Geben Sie als Größe nacheinander die Werte 8k, 16k, 32k, 64k, 128k, 256k, 512k, 1m, 2m, 4m, 8m, 16m, 32m, 64m und 128m an und notieren Sie, wie viele Aufrufe der Methode jetzt möglich sind, bis es zum Speicherüberlauf kommt!
Wie groß ist der bei einem Aufruf auf dem Stack allokierte Speicherplatz?
Wie groß ist die minimale Stack-Größe?

c) Probieren Sie auch Zwischenwerte als Stack-Größe aus! Was fällt hier auf?

d) Legen Sie nun im Rumpf der endlosrekursiven Methode einmal ein Feld mit 100 Integer-Werten und einmal 100 einzelne Integer-Variablen an! Wiederholen Sie die Messungen aus Teilaufgabe b) und erläutern Sie die Werte!