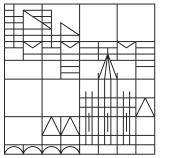
Universität Konstanz



Konzepte der Informatik

Algorithmik
Rekursion

Barbara Pampel

Universität Konstanz, WiSe 2023/2024

Inhalt

Rekursion

Rekursion

Als Rekursion (lat. recurrere *„zurücklaufen”) bezeichnet man die Technik in Mathematik, Logik und Informatik, eine Funktion [Methode] durch sich selbst zu definieren (rekursive Definition).*

Rekursion

Als Rekursion (lat. recurrere *„zurücklaufen") bezeichnet man die Technik in Mathematik, Logik und Informatik, eine Funktion [Methode] durch sich selbst zu definieren (rekursive Definition).*
Das Grundprinzip der rekursiven Definition einer Funktion f ist: Der Funktionswert $f(n+1)$ einer Funktion $f : N_0 \to N_0$ ergibt sich durch Verknüpfung bereits vorher berechneter Werte $f(n), f(n-1), \ldots$ Falls außerdem die Funktionswerte von f für hinreichend viele Startargumente bekannt sind, kann jeder Funktionswert von f berechnet werden. Bei einer rekursiven Definition einer Funktion f ruft sich somit die Funktion so oft selbst auf, bis eine durch den Aufruf der Funktion veränderte Variable einen vorgegebenen Zielwert erreicht oder Grenzwert überschritten hat.

Aus Wikipedia

Iteration vs. Rekursion

Algorithm 1: Binäre Suche iterativ

```
binsearch(M[0, ..., n − 1], k) begin
    l ← 0;   r ← n − 1
    while k ≥ M[l] and k ≤ M[r] do
        m ← ⌊(l+r)/2⌋
        if k > M[m] then l ← m + 1
        else if k < M[m] then r ← m − 1
        else
            return „k ist in M“
    return „k ist nicht in M“
```

Iteration vs. Rekursion

Algorithm 2: Binäre Suche rekursiv

Aufruf: `binsearch`(M, k, 0, $n-1$)

`binsearch`(M, k, l, r) **begin**
  if $r < l$ **then return** „k ist nicht in M
  $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$
  if $k > M[m]$ **then** `binsearch`(M, k, $m+1$, r)
  else if $k < M[m]$ **then** `binsearch`(M, k, l, $m-1$)
  else
    return „k ist in M“

Der größte gemeinsame Teiler

Der größte gemeinsame Teiler

- Mathematische Definition des ggT
  - $ggT(a, b) = ggT(b, a - b)$

Algorithm 4: ggt(a,b)

Input: $a, b \in \mathbb{Z}$

begin

  $t \leftarrow ggt(b, a - b)$

  return t

- Methode ruft sich selbst *mit veränderten Parametern* auf

Der größte gemeinsame Teiler

- Mathematische Definition des ggT
  - $ggT(a, b) = ggT(b, a - b)$

Algorithm 5: ggt(a,b)

Input: $a, b \in \mathbb{Z}$

begin

$t \leftarrow ggt(b, a - b)$

return t

- Methode ruft sich selbst *mit veränderten Parametern* auf
- $ggT(9, 6) \Rightarrow ggT(6, 3) \Rightarrow ggT(3, 3) \Rightarrow ggT(3, 0) \Rightarrow ggT(0, 3) \Rightarrow ggT(3, -3) \Rightarrow ggT(-3, 6) \Rightarrow \ldots$
- Problem: Endlosrekursion — Methode ruft sich *immer* selbst auf

Abbruchbedingung

- Mathematische Definition des ggT
  - $ggT(a, b) = ggT(b, a - b)$
  - $ggT(a, a) = a \quad (a \neq 0)$

Algorithm 6: ggt(a,b)

Input: $a, b \in \mathbb{Z}$

begin

  if $a = b$ **then** $t \leftarrow a$

  else $t \leftarrow ggt(b, a - b)$

  return t

- Eintreten der Abbruchbedingung beendet die Rekursion
- $ggT(9, 6) \Rightarrow ggT(6, 3) \Rightarrow ggT(3, 3) = 3$

Abbruchbedingung

- Mathematische Definition des ggT
  - $ggT(a, b) = ggT(b, a - b)$
  - $ggT(a, a) = a \quad (a \neq 0)$

Algorithm 7: ggt(a,b)

Input: $a, b \in \mathbb{Z}$

begin

  if $a = b$ **then** $t \leftarrow a$

  else $t \leftarrow ggt(b, a - b)$

  return t

- Eintreten der Abbruchbedingung beendet die Rekursion
- $ggT(9, 6) \Rightarrow ggT(6, 3) \Rightarrow ggT(3, 3) = 3$
- $ggT(9, 5) \Rightarrow ggT(5, 4) \Rightarrow ggT(4, 1) \Rightarrow ggT(1, 3) \Rightarrow ggT(3, -2) \Rightarrow ggT(-2, 5) \Rightarrow \ldots$
- Problem: Abbruchbedingung nicht vollständig

ggT komplett

- Mathematische Definition des ggT
  - $ggT(a, b) = ggT(b, a - b)$
  - $ggT(a, a) = a \quad (a \neq 0)$
  - $ggT(a, 0) = a \quad (a \neq 0)$
  - $ggT(a, 1) = 1$
  - $ggT(a, -b) = ggT(a, b)$

Algorithm 8: ggt(a,b)

Input: $a, b \in \mathbb{Z}$

begin

if $a = b = 0$ **then** return *Fehler*
else if $a = b$ **then** $t \leftarrow a$
else if $b = 0$ **then** $t \leftarrow a$
else if $b = 1$ **then** $t \leftarrow 1$
else if $b < 0$ **then** $t \leftarrow ggT(a, -b)$
else $t \leftarrow ggt(b, a - b)$
return t

Algorithmus von Euklid

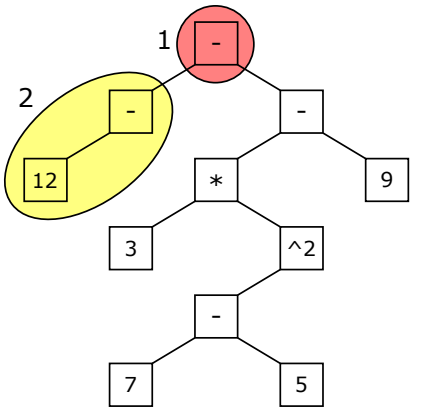
Algorithm 9: EUCLID(a,b)

Input: $a, b \in \mathbb{N}; a \geq b$
Output: t = ggT(a,b)
begin
  if $b = 0$ **then**
    $t \leftarrow a$
  else
    $t \leftarrow EUCLID(b, mod(a, b))$
  return t

Preorder-Traversierung

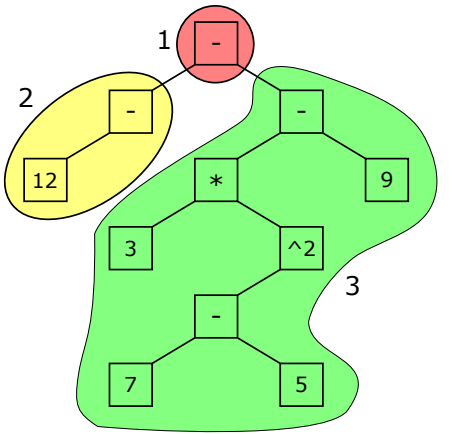
Preorder-Traversierung

- Wurzel besuchen, linken Teilbaum in preorder traversieren, rechten Teilbaum in preorder traversieren



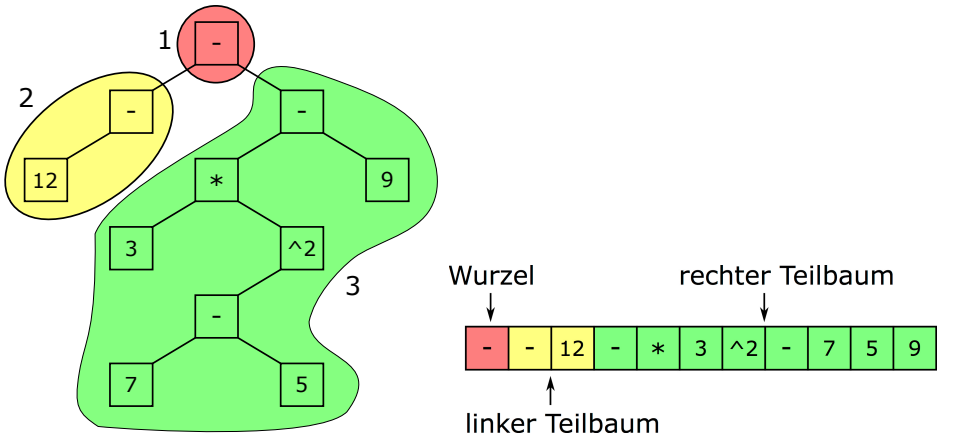
Preorder-Traversierung

- Wurzel besuchen, linken Teilbaum in preorder traversieren, rechten Teilbaum in preorder traversieren



Preorder-Traversierung

- Wurzel besuchen, linken Teilbaum in preorder traversieren, rechten Teilbaum in preorder traversieren



Implementierung Preorder-Traversierung

- Wurzel besuchen, linken Teilbaum in preorder traversieren, rechten Teilbaum in preorder traversieren

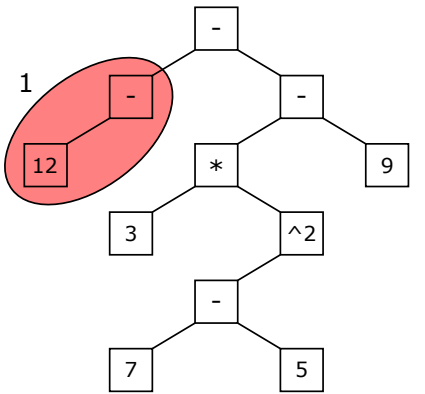
Algorithm 10: preorder

```
order(v) begin
    if v ≠ nil then
        print v.key
        order(v.left)
        order(v.right)
```

Aufruf: `order(root)`

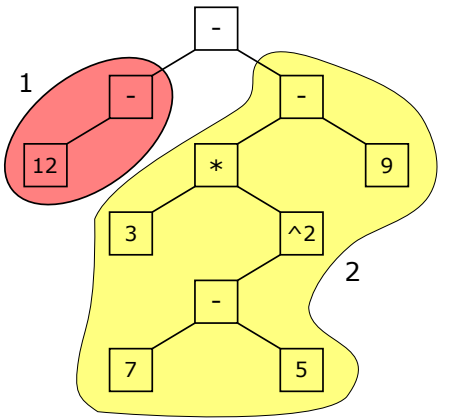
Postorder-Traversierung

- Linken Teilbaum postorder traversieren, rechten Teilbaum postorder traversieren, Wurzel besuchen



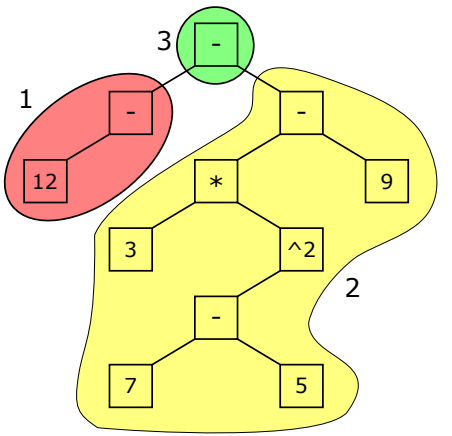
Postorder-Traversierung

- Linken Teilbaum postorder traversieren, rechten Teilbaum postorder traversieren, Wurzel besuchen



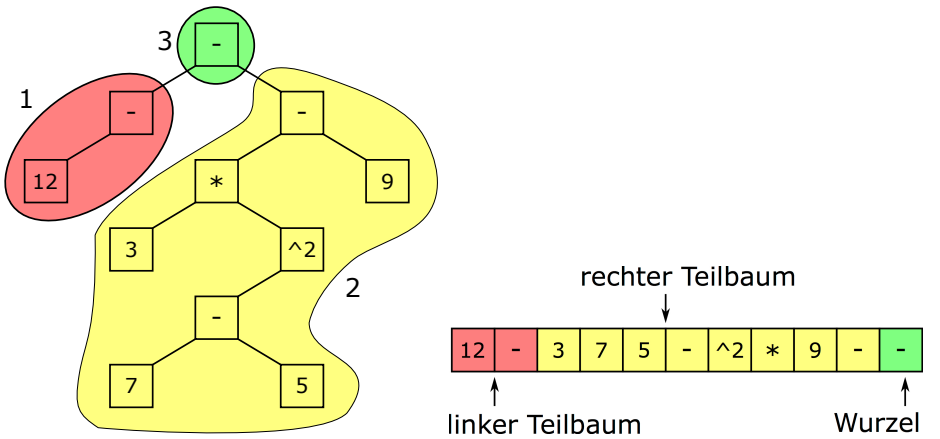
Postorder-Traversierung

- Linken Teilbaum postorder traversieren, rechten Teilbaum postorder traversieren, Wurzel besuchen



Postorder-Traversierung

- Linken Teilbaum postorder traversieren, rechten Teilbaum postorder traversieren, Wurzel besuchen



Implementierung Postorder-Traversierung

- Linken Teilbaum postorder traversieren, rechten Teilbaum postorder traversieren, Wurzel besuchen

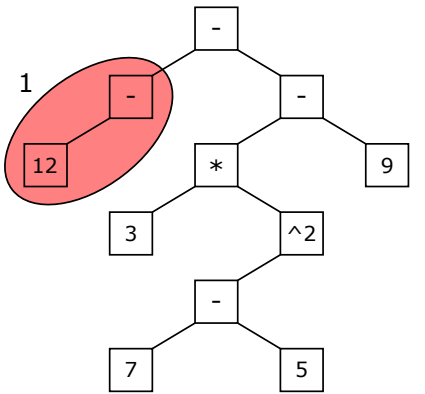
Algorithm 11: postorder

```
order(v) begin
    if v ≠ nil then
        order(v.left)
        order(v.right)
        print v.key
```

Aufruf: `order(root)`

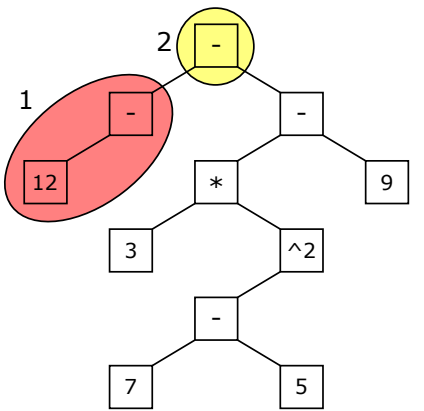
Inorder-Traversierung

- Linken Teilbaum inorder traversieren, Wurzel besuchen, rechten Teilbaum inorder traversieren



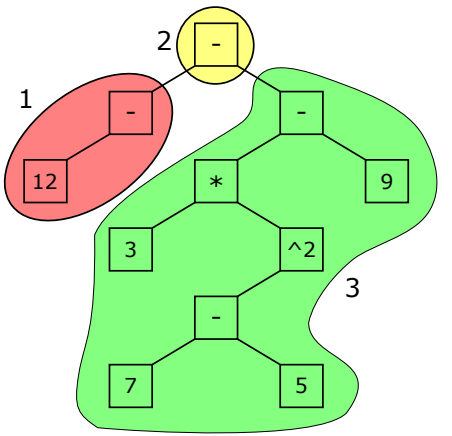
Inorder-Traversierung

- Linken Teilbaum inorder traversieren, Wurzel besuchen, rechten Teilbaum inorder traversieren



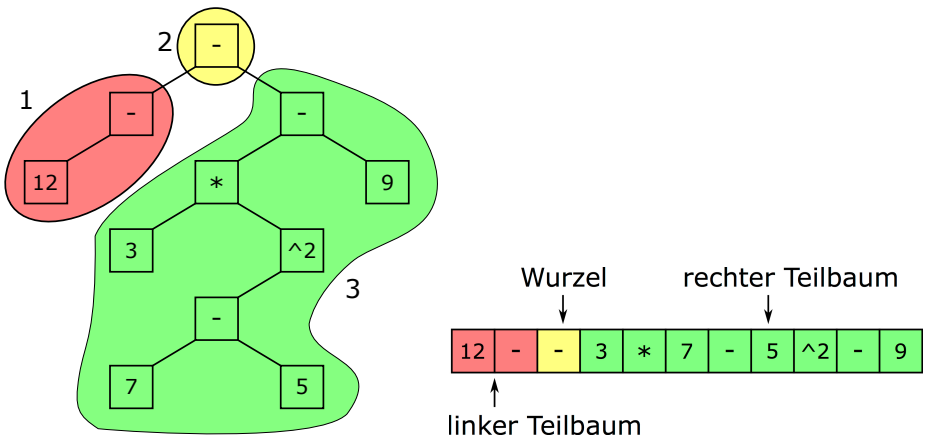
Inorder-Traversierung

- Linken Teilbaum inorder traversieren, Wurzel besuchen, rechten Teilbaum inorder traversieren



Inorder-Traversierung

- Linken Teilbaum inorder traversieren, Wurzel besuchen, rechten Teilbaum inorder traversieren



Implementierung Inorder-Traversierung

- Linken Teilbaum inorder traversieren, Wurzel besuchen, rechten Teilbaum inorder traversieren

Algorithm 12: inorder

```
order(v) begin
    if v ≠ nil then
        order(v.left)
        print v.key
        order(v.right)
```

Aufruf: `order(root)`

Fakultätsfunktion

Fakultätsfunktion

- Mathematische Definition
  - $f(n) = n \cdot f(n-1)$
  - $f(0) = 1$

Fakultätsfunktion

- Mathematische Definition
  - $f(n) = n \cdot f(n-1)$
  - $f(0) = 1$

```
public int fac(int n) {
  if (n < 0) {
    throw new IllegalArgumentException(
      "Negative values are not allowed"); }
  if (n == 0) { return 1; }
  return n * fac(n - 1);
}
```

Wie funktioniert Rekursion?

- Ausgabe während der Ausführung

```
public int fac(int n) {
  if (n == 0) { return 1; }
  System.out.println("> fac(" + n + ")");
  int result = n * fac(n - 1);
  System.out.println("< fac(" + n + ") = " + result);
  return result;
}
```

Wie funktioniert Rekursion?

> fac(5)
> fac(4)
> fac(3)
> fac(2)
> fac(1)
< fac(1) = 1
< fac(2) = 2
< fac(3) = 6
< fac(4) = 24
< fac(5) = 120

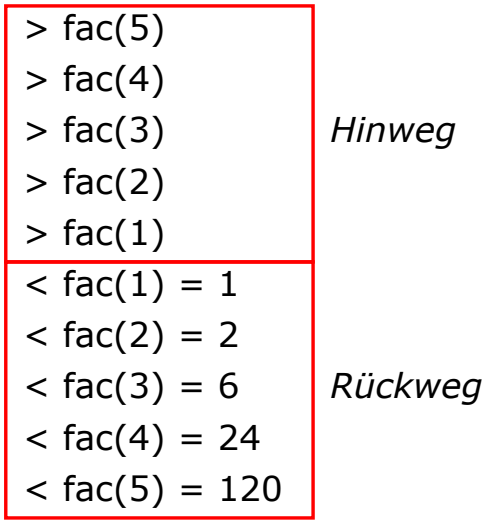
Wie funktioniert Rekursion?

> fac(5)
> fac(4)
> fac(3)
> fac(2)
> fac(1)

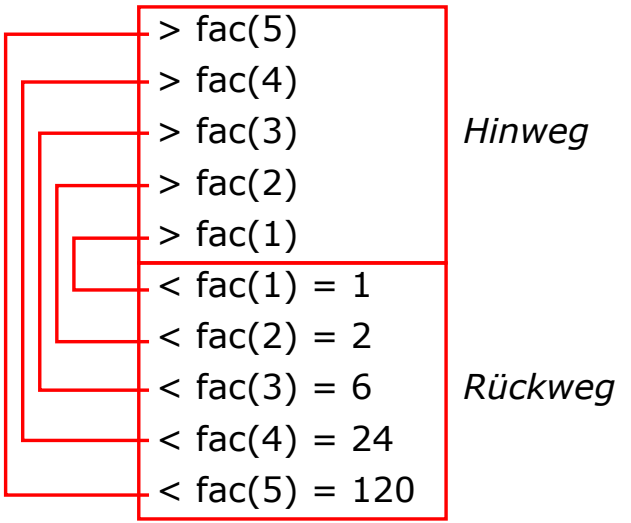
Hinweg

< fac(1) = 1
< fac(2) = 2
< fac(3) = 6
< fac(4) = 24
< fac(5) = 120

Wie funktioniert Rekursion?

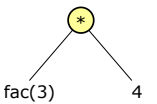


Wie funktioniert Rekursion?



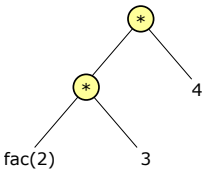
Entfaltung der Berechnung

- Berechnung lässt sich als Baum darstellen



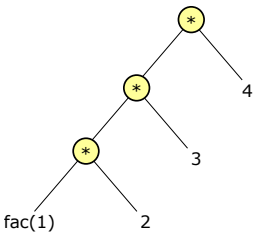
Entfaltung der Berechnung

- Berechnung lässt sich als Baum darstellen



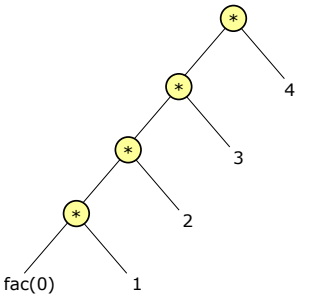
Entfaltung der Berechnung

- Berechnung lässt sich als Baum darstellen



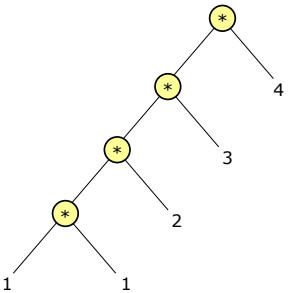
Entfaltung der Berechnung

- Berechnung lässt sich als Baum darstellen



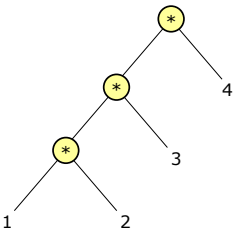
Berechnen des Baumes

- Baum lässt sich berechnen



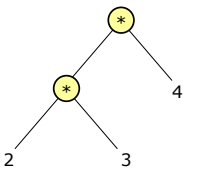
Berechnen des Baumes

- Baum lässt sich berechnen



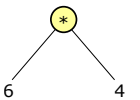
Berechnen des Baumes

- Baum lässt sich berechnen



Berechnen des Baumes

- Baum lässt sich berechnen



Das Wichtigste in Kürze

- Rekursion
  - Aufruf einer Methode durch sich selbst
  - benötigt zwingend passende Abbruchbedingung
  - Darstellung als Baum

Inhalt

Binomialkoeffizienten

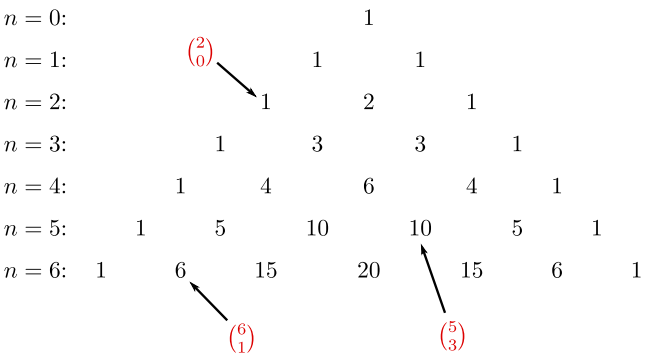
- Aus der Schule bekannt — binomische Formel
  - $(a+b)^2 = a^2 + 2ab + b^2$
  - $(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$
  - $(a+b)^n = a^n + \binom{n}{1} a^{n-1}b^1 + \binom{n}{2} a^{n-2}b^2 + \ldots + \binom{n}{n-1} a^1 b^{n-1} + b^n$

Binomialkoeffizienten

- Aus der Schule bekannt — binomische Formel
  - $(a+b)^2 = a^2 + 2ab + b^2$
  - $(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$
  - $(a+b)^n = a^n + \binom{n}{1} a^{n-1}b^1 + \binom{n}{2} a^{n-2}b^2 + \ldots + \binom{n}{n-1} a^1 b^{n-1} + b^n$
  - $\binom{n}{k}$ ist Binomialkoeffizient

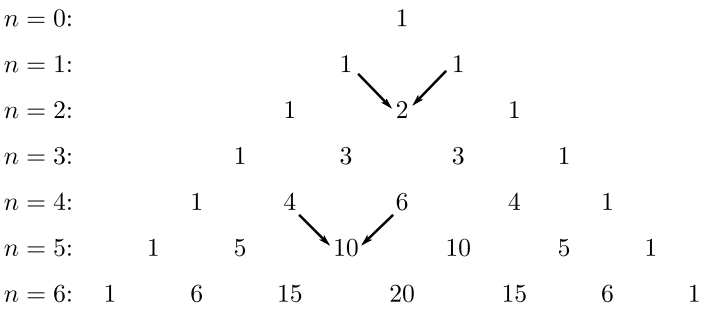
Binomialkoeffizienten

- Aus der Schule bekannt — binomische Formel
  - $(a+b)^2 = a^2 + 2ab + b^2$
  - $(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$
  - $(a+b)^n = a^n + \binom{n}{1} a^{n-1}b^1 + \binom{n}{2} a^{n-2}b^2 + \ldots + \binom{n}{n-1} a^1 b^{n-1} + b^n$
  - $\binom{n}{k}$ ist Binomialkoeffizient
- Berechnung z.B. durch Pascalsches Dreieck



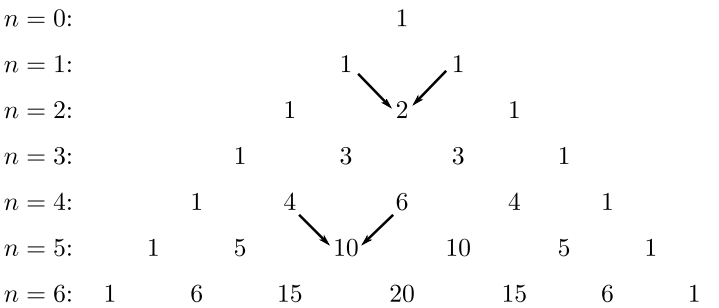
Berechnung der Binomialkoeffizienten

- $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$



Berechnung der Binomialkoeffizienten

- $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$



- Berechnungsvorschrift offensichtlich rekursiv

Rekursive Berechnung

- Rekursionsformel

```
public int binom(int n, int k) {
  return binom(n - 1, k - 1) + binom(n - 1, k);
}
```

Rekursive Berechnung

- Rekursionsformel

```
public int binom(int n, int k) {
  return binom(n - 1, k - 1) + binom(n - 1, k);
}
```

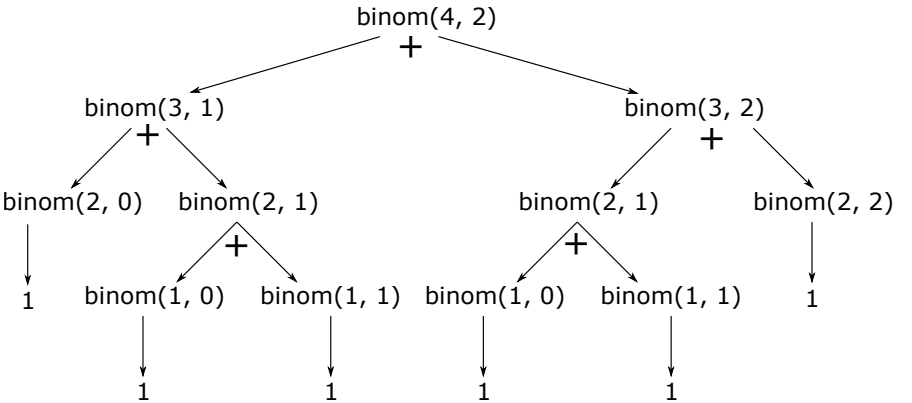
- Abbruchbedingung

```
public int binom(int n, int k) {
  if ((k == 0) || (k == n)) { return 1; }
  if (n == 0) { return 1; }

  return binom(n - 1, k - 1) + binom(n - 1, k);
}
```

- Es fehlt die Behandlung von unzulässigen Parametern

Rekursionsentfaltung



Die Ackermannfunktion

- Bekannte Funktion aus der theoretischen Informatik
- Wächst extrem schnell
  - $ack(0, m) = m + 1$
  - $ack(n + 1, 0) = ack(n, 1)$
  - $ack(n + 1, m + 1) = ack(n, ack(n + 1, m))$
- Verschachtelte rekursive Aufrufe
  - $ack(3, 3) = ack(2, ack(3, 2)) = ack(2, ack(2, ack(3, 1))) = \ldots = 61$
  - $ack(4, 2) > 10^{21000}$
  - $ack(4, 4) > 10^{10^{10^{21000}}}$
- Verschachtelt rekursive Funktionen sind schwer zu durchschauen

Rekursionsarten

- Lineare Rekursion
  - ggT — Methode ruft sich selber einmal rekursiv auf
- Baumartige oder kaskadenartige Rekursion
  - Binomialkoeffizienten — Methode ruft sich mehrmals rekursiv auf
- Verschachtelte Rekursion
  - Ackermannfunktion — Parameter der Methode enthalten rekursiven Aufruf
- Endrekursion
  - Rekursiver Aufruf ist letzte Aktion in der Methode
    - ggT ist endrekursiv
    - Binomialkoeffizienten ist nicht endrekursiv, letzte Anweisung ist Addition
  - Ergebnis des letzten rekursiven Aufrufs ist Gesamtergebnis

Das Wichtigste in Kürze

- Rekursion
  - Aufruf einer Methode durch sich selbst
  - benötigt zwingend passende Abbruchbedingung
  - Darstellung als Baum
- Lineare, baumartige und verschachtelte Rekursion

Inhalt

Der Stapel

- Speicherbereich, der von aufgerufenen Methoden verwendet wird; auch Stack genannt
  - übergebene Parameter
  - Rücksprungadresse
  - lokale Variablen
  - Rückgabewerte

Der Stapel

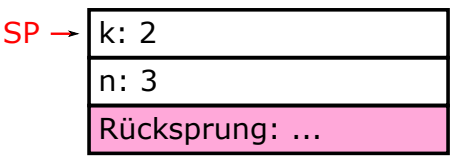
- Speicherbereich, der von aufgerufenen Methoden verwendet wird; auch Stack genannt
  - übergebene Parameter
  - Rücksprungadresse
  - lokale Variablen
  - Rückgabewerte
- Stapel wächst und schrumpft automatisch bei jedem Methodenaufruf
- Stapel hat eine voreingestellte Maximalgröße
- Stapelzeiger markiert den aktuell obersten Eintrag
  - Adressierung von Variablen und Parametern relativ zum Stapelzeiger
  - Methodenaufruf setzt den Zeiger nach oben und schafft Platz für neue Einträge
  - nach Rückkehr wird Zeiger wieder nach unten verschoben

Stapelbenutzung

- Erster Aufruf der Methode von „außen”
  - Rücksprungadresse zeigt auf Aufrufstelle
  - Parameter von „außen” gesetzt

```
   int binom(int n, int k) {
1    if (k == 0) {
2      return 1;
     }
3    if (k == n) {
4      return 1;
     }
5    if (n == 0) {
6      return 1;
     }

7    int result =
8      binom(n - 1, k - 1)
9      + binom(n - 1, k);
10   return result;
   }
```

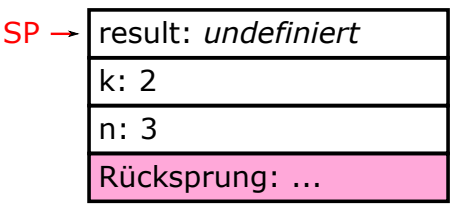


Stapelbenutzung

- Platz für lokale Variablen schaffen
  - Adressierung relativ zum SP, z.B. $addr(k) = SP - 1$

```
   int binom(int n, int k) {
1    if (k == 0) {
2      return 1;
     }
3    if (k == n) {
4      return 1;
     }
5    if (n == 0) {
6      return 1;
     }

7    int result =
8      binom(n - 1, k - 1)
9      + binom(n - 1, k);
10   return result;
   }
```

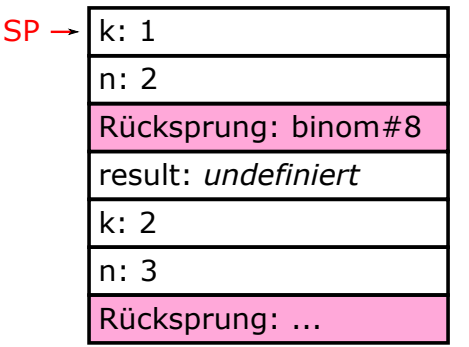


Stapelbenutzung

- Rücksprungadresse und Parameter auf Stapel legen
- Danach Aufruf der Methode

```
   int binom(int n, int k) {
1    if (k == 0) {
2      return 1;
     }
3    if (k == n) {
4      return 1;
     }
5    if (n == 0) {
6      return 1;
     }

7    int result =
8      binom(n - 1, k - 1)
9      + binom(n - 1, k);
10   return result;
   }
```

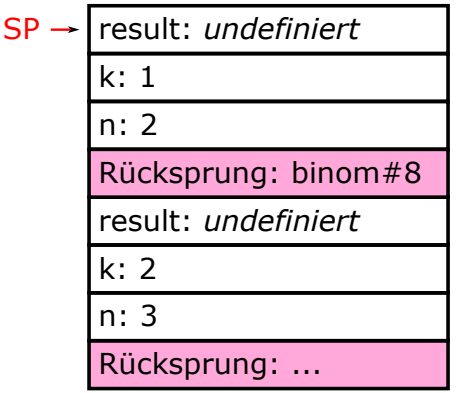


Stapelbenutzung

- Platz für lokale Variablen schaffen

```
   int binom(int n, int k) {
 1   if (k == 0) {
 2     return 1;
     }
 3   if (k == n) {
 4     return 1;
     }
 5   if (n == 0) {
 6     return 1;
     }

 7   int result =
 8     binom(n - 1, k - 1)
 9     + binom(n - 1, k);
10   return result;
   }
```

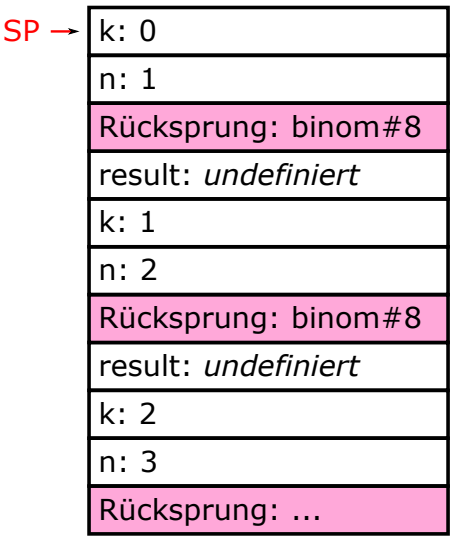


Stapelbenutzung

- Rücksprungadresse und Parameter auf den Stapel legen
- Danach Aufruf der Methode

```
   int binom(int n, int k) {
1    if (k == 0) {
2      return 1;
     }
3    if (k == n) {
4      return 1;
     }
5    if (n == 0) {
6      return 1;
     }

7    int result =
8      binom(n - 1, k - 1)
9      + binom(n - 1, k);
10   return result;
   }
```

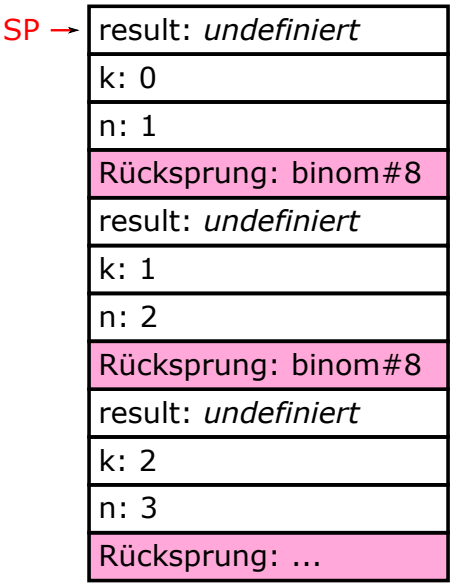


Stapelbenutzung

- Platz für lokale Variablen schaffen

```
   int binom(int n, int k) {
1    if (k == 0) {
2      return 1;
     }
3    if (k == n) {
4      return 1;
     }
5    if (n == 0) {
6      return 1;
     }

7    int result =
8      binom(n - 1, k - 1)
9      + binom(n - 1, k);
10   return result;
   }
```

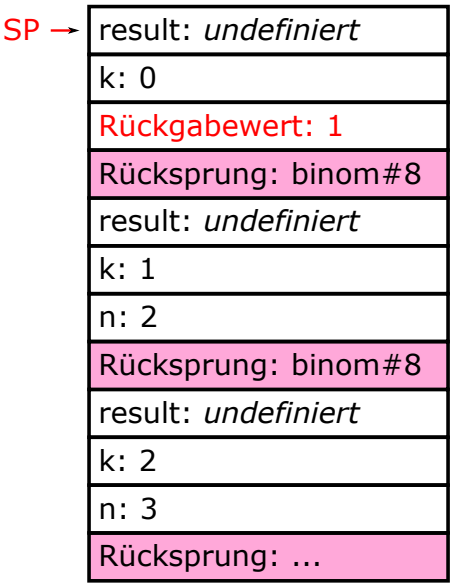


Stapelbenutzung

- Rückgabewert auf Stapel schreiben
  - überschreibt (nicht mehr benötigte) Variablen

```
   int binom(int n, int k) {
1    if (k == 0) {
2      return 1;
     }
3    if (k == n) {
4      return 1;
     }
5    if (n == 0) {
6      return 1;
     }

7    int result =
8      binom(n - 1, k - 1)
9      + binom(n - 1, k);
10   return result;
   }
```

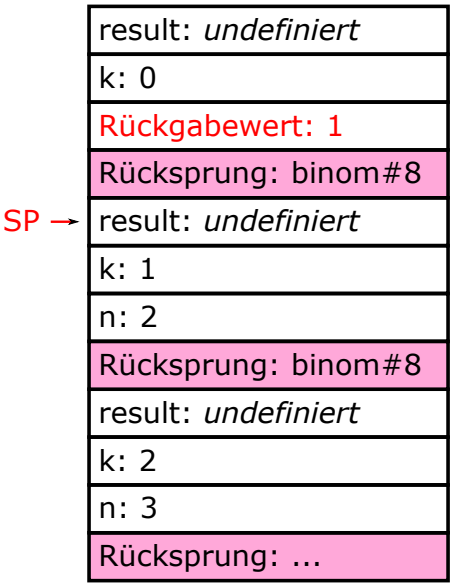


Stapelbenutzung

- Rücksprung an hinterlegte Adresse

```
   int binom(int n, int k) {
1    if (k == 0) {
2      return 1;
     }
3    if (k == n) {
4      return 1;
     }
5    if (n == 0) {
6      return 1;
     }

7    int result =
8      binom(n - 1, k - 1)
9      + binom(n - 1, k);
10   return result;
   }
```

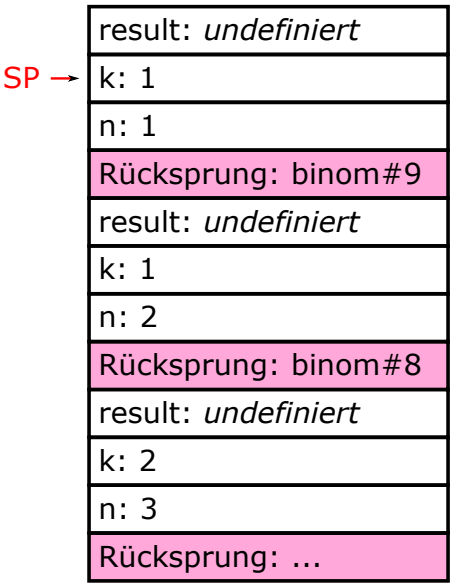


Stapelbenutzung

- Rücksprungadresse und Parameter auf den Stapel legen
- Danach Aufruf der Methode

```
   int binom(int n, int k) {
1    if (k == 0) {
2      return 1;
     }
3    if (k == n) {
4      return 1;
     }
5    if (n == 0) {
6      return 1;
     }

7    int result =
8      binom(n - 1, k - 1)
9      + binom(n - 1, k);
10   return result;
   }
```

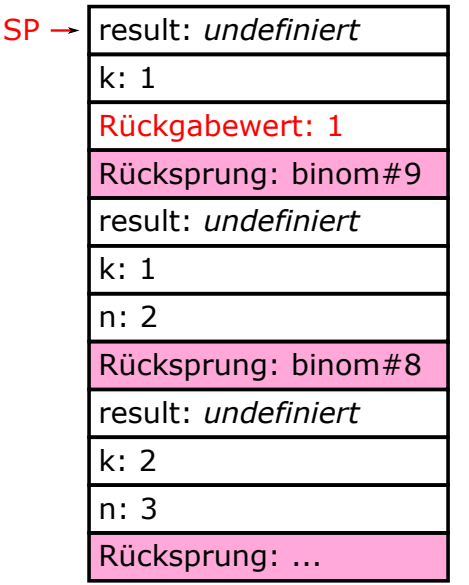


Stapelbenutzung

- Rückgabewert auf Stapel schreiben
  - überschreibt (nicht mehr benötigte) Variablen

```
   int binom(int n, int k) {
1    if (k == 0) {
2      return 1;
     }
3    if (k == n) {
4      return 1;
     }
5    if (n == 0) {
6      return 1;
     }

7    int result =
8      binom(n - 1, k - 1)
9      + binom(n - 1, k);
10   return result;
   }
```

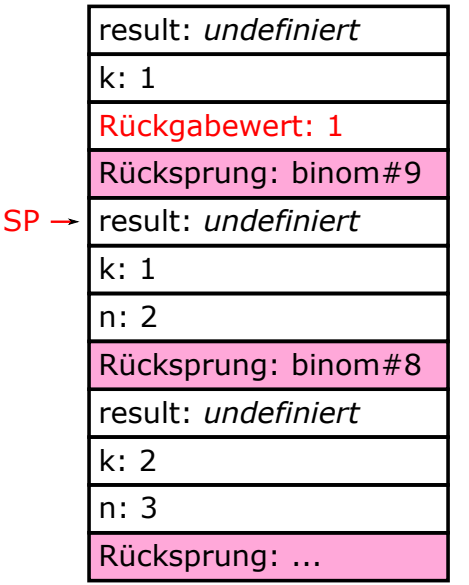


Stapelbenutzung

- Rücksprung an hinterlegte Adresse

```
   int binom(int n, int k) {
 1   if (k == 0) {
 2     return 1;
     }
 3   if (k == n) {
 4     return 1;
     }
 5   if (n == 0) {
 6     return 1;
     }

 7   int result =
 8     binom(n - 1, k - 1)
 9     + binom(n - 1, k);
10   return result;
   }
```

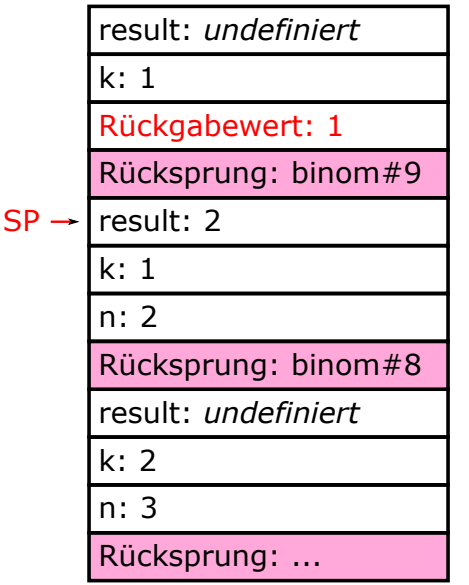


Stapelbenutzung

- Zugriff auf Rückgabewert über $SP + 2$

```
   int binom(int n, int k) {
1    if (k == 0) {
2      return 1;
     }
3    if (k == n) {
4      return 1;
     }
5    if (n == 0) {
6      return 1;
     }

7    int result =
8      binom(n - 1, k - 1)
9      + binom(n - 1, k);
10   return result;
   }
```

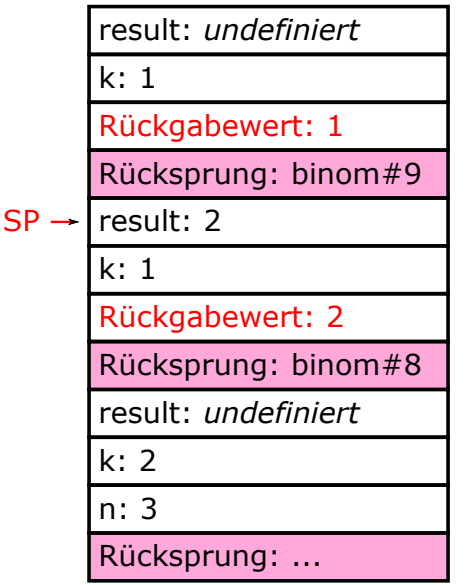


Stapelbenutzung

- Rückgabewert auf Stapel schreiben
  - überschreibt (nicht mehr benötigte) Variablen

```
   int binom(int n, int k) {
1    if (k == 0) {
2      return 1;
     }
3    if (k == n) {
4      return 1;
     }
5    if (n == 0) {
6      return 1;
     }

7    int result =
8      binom(n - 1, k - 1)
9      + binom(n - 1, k);
10   return result;
   }
```

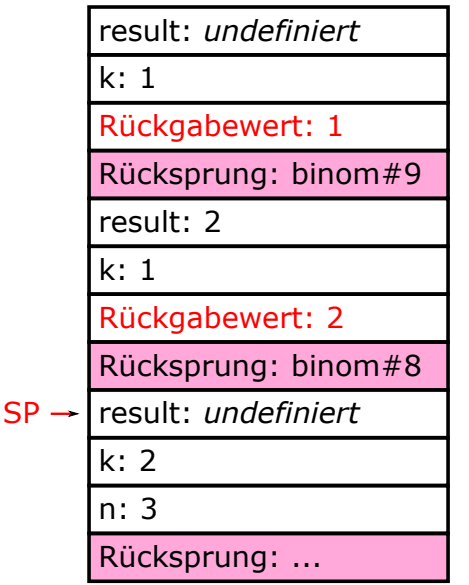


Stapelbenutzung

- Rücksprung an hinterlegte Adresse

```
   int binom(int n, int k) {
1    if (k == 0) {
2      return 1;
     }
3    if (k == n) {
4      return 1;
     }
5    if (n == 0) {
6      return 1;
     }

7    int result =
8      binom(n - 1, k - 1)
9      + binom(n - 1, k);
10   return result;
   }
```



- Und so weiter...

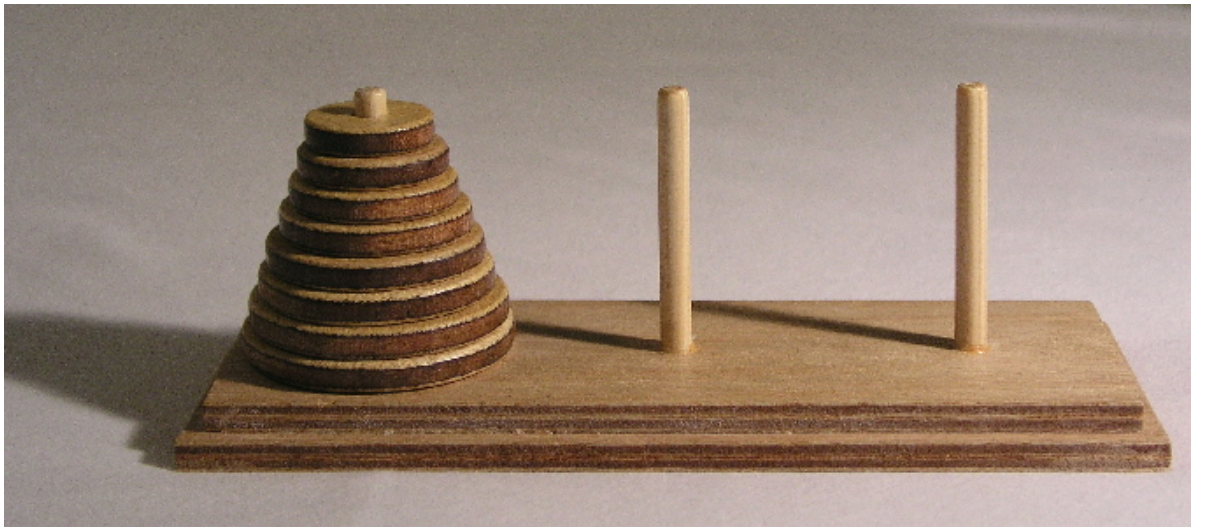
Das Wichtigste in Kürze

- Rekursion
  - Aufruf einer Methode durch sich selbst
  - benötigt zwingend passende Abbruchbedingung
  - Darstellung als Baum
- Lineare, baumartige und verschachtelte Rekursion
- Verwendung eines Stapels bei Methodenaufrufen

Inhalt

Die Türme von Hanoi

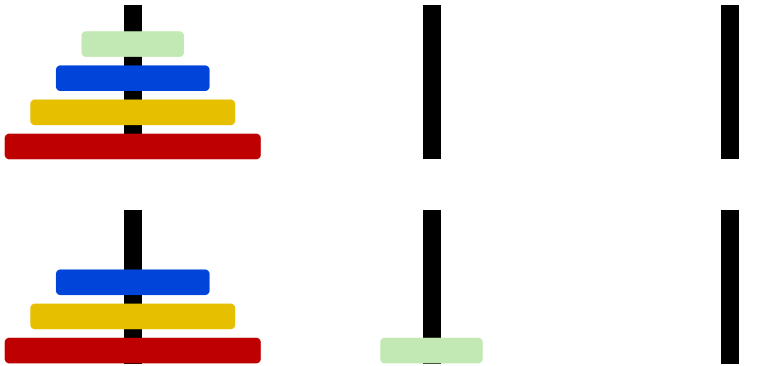
- Bekanntes Knobelspiel



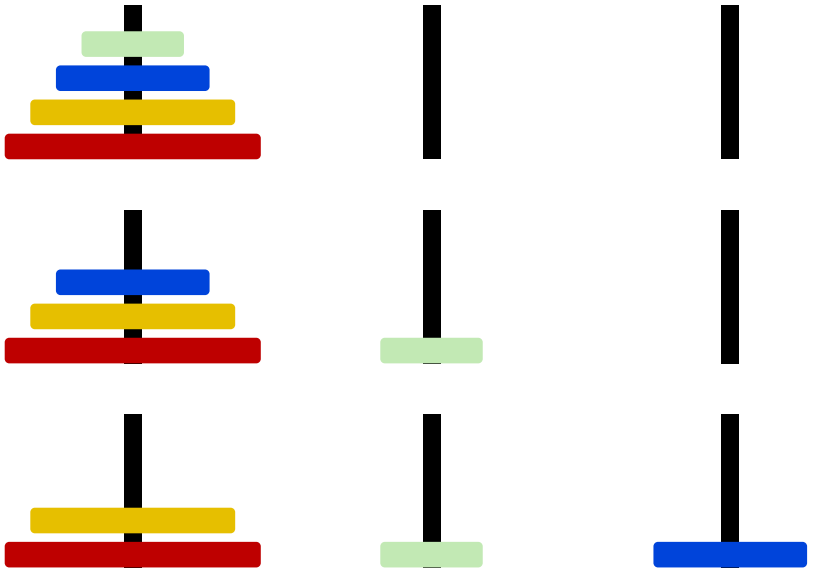
- Scheiben müssen einzeln von einem Stab auf einen anderen verschoben werden
- Größere Scheiben müssen immer unten liegen

Beispiel mit vier Scheiben

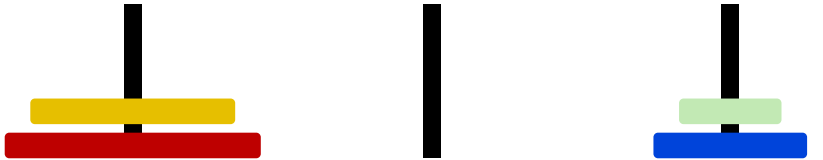
Beispiel mit vier Scheiben



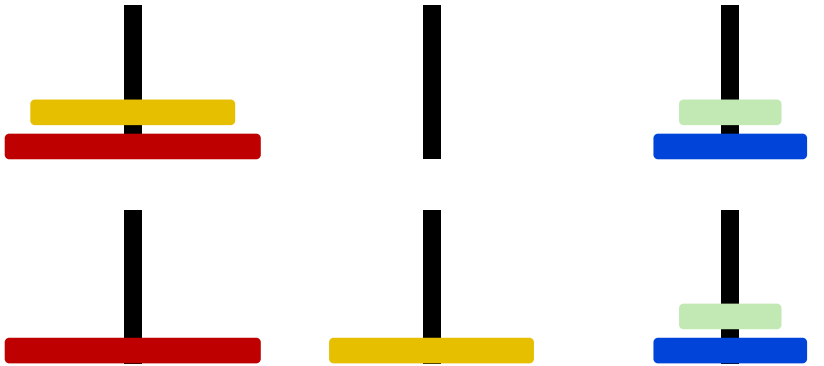
Beispiel mit vier Scheiben



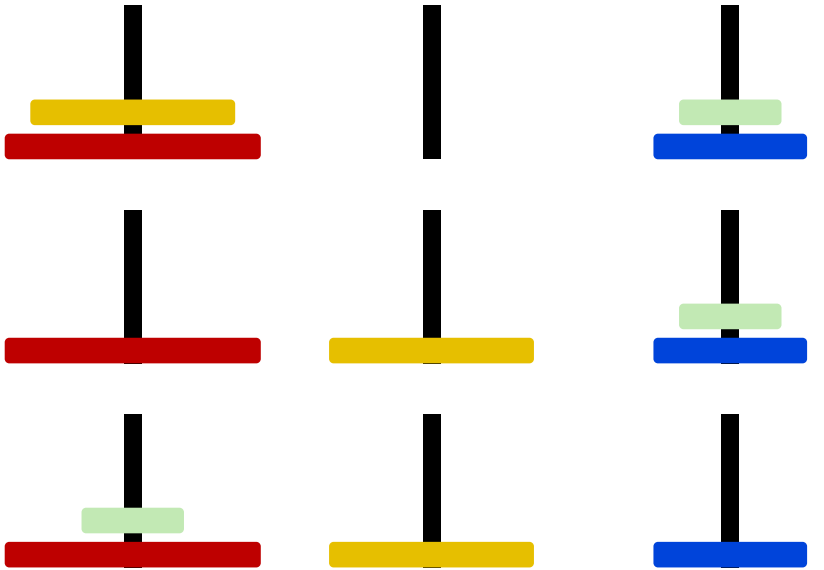
Beispiel mit vier Scheiben



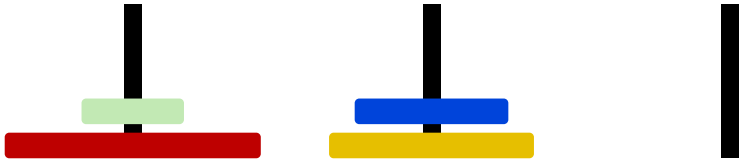
Beispiel mit vier Scheiben



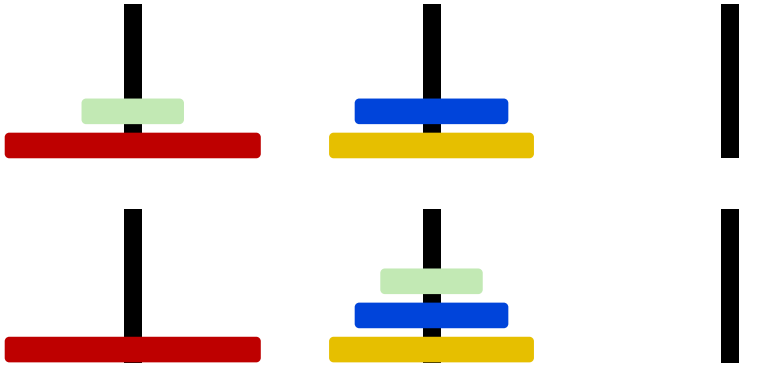
Beispiel mit vier Scheiben



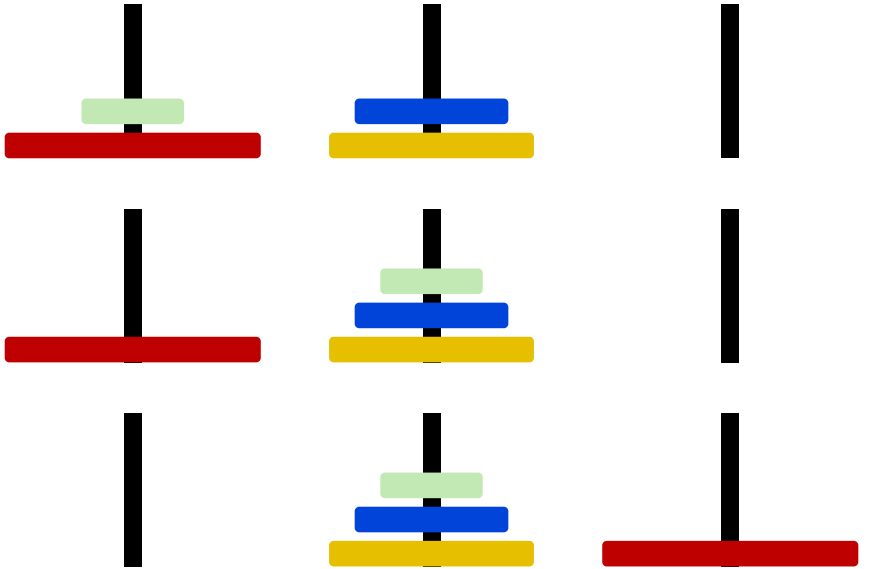
Beispiel mit vier Scheiben



Beispiel mit vier Scheiben

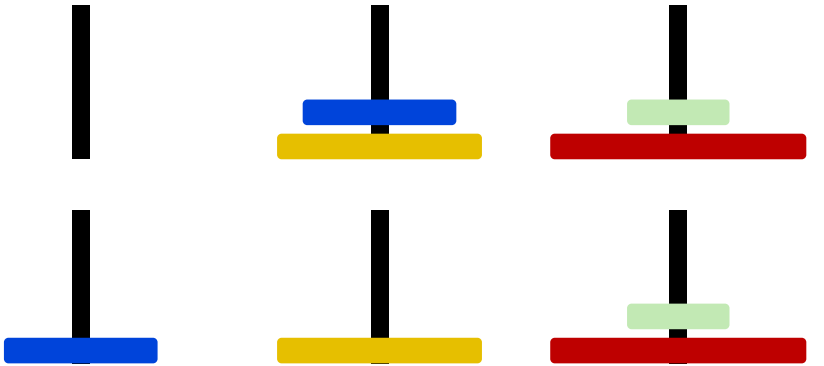


Beispiel mit vier Scheiben

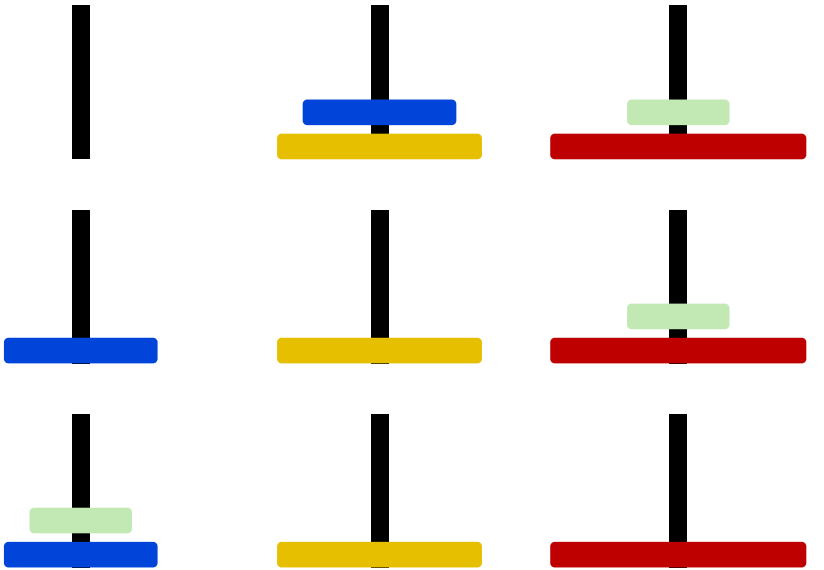


Beispiel mit vier Scheiben

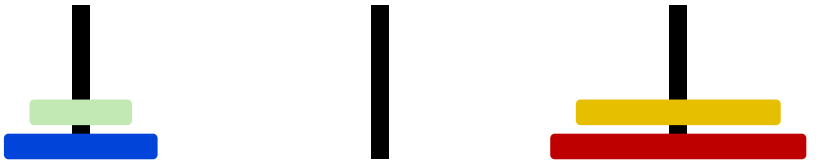
Beispiel mit vier Scheiben



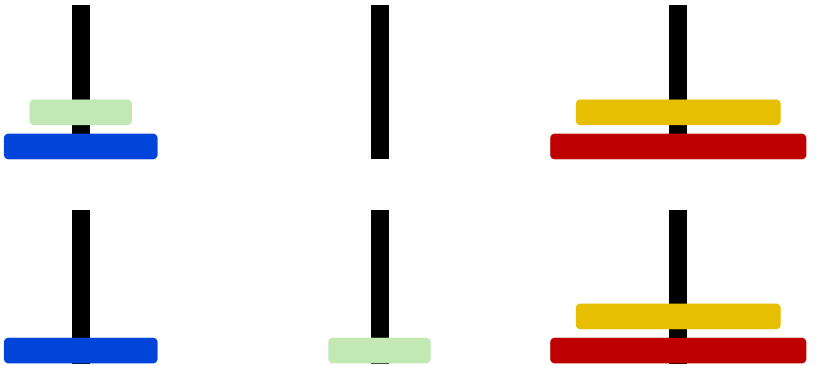
Beispiel mit vier Scheiben



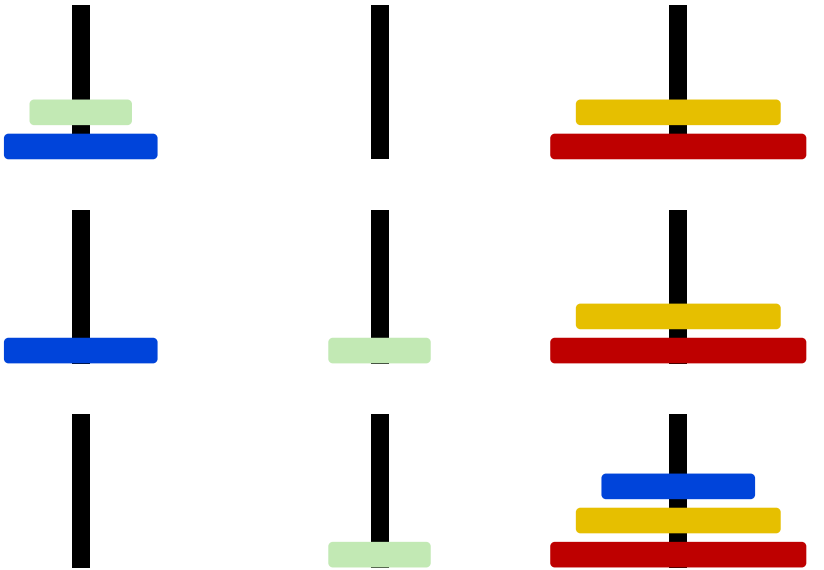
Beispiel mit vier Scheiben



Beispiel mit vier Scheiben



Beispiel mit vier Scheiben



Beispiel mit vier Scheiben

Lösungsidee

- Zurückführung auf kleineres Problem
  - Bewegen von n Scheiben von Stapel a nach Stapel c unter Benutzungs eines Zwischenstapels b
  - Falls $n = 1$
    - (oberste) Scheibe von a nach c bewegen
  - Falls $n > 1$
    - erst Stapel der Größe $n - 1$ von a auf den Zwischenstapel b verschieben (mit c als Zwischenstapel)
    - dann letzte Scheibe von a nach c bewegen
    - zum Schluß die $n - 1$ Scheiben auf dem Zwischenstapel b nach c verschieben (mit a als Zwischenstapel)

Rekursive Lösung

```
public void moveTower(int disks, Bar a, Bar b, Bar c) {
  if (disks > 0) {
    moveTower(disks - 1, a, c, b);
    System.out.println("Move top disk from bar "
      + a  + " to bar " + c);
    moveTower(disks - 1, b, a, c);
  }
}
public class Bar {
  private final String name;
  public Bar(String name) { this.name = name; }
  public String toString() { return name; }
}
```

Beispiellauf

```
hanoi.moveTower(4, new Bar("1"), new Bar("2"), new Bar("3"));
```

```
Move top disk from bar 1 to bar 2
Move top disk from bar 1 to bar 3
Move top disk from bar 2 to bar 3
Move top disk from bar 1 to bar 2
Move top disk from bar 3 to bar 1
Move top disk from bar 3 to bar 2
Move top disk from bar 1 to bar 2
Move top disk from bar 1 to bar 3
Move top disk from bar 2 to bar 3
Move top disk from bar 2 to bar 1
Move top disk from bar 3 to bar 1
Move top disk from bar 2 to bar 3
Move top disk from bar 1 to bar 2
Move top disk from bar 1 to bar 3
Move top disk from bar 2 to bar 3
```

Aufwandsbetrachtung

- Wieviele Scheiben werden während der Lösung bewegt?

Aufwandsbetrachtung

- Wieviele Scheiben werden während der Lösung bewegt?
- $moves(n)$: Anzahl der Verschiebungen für n Scheiben auf dem Ausgangsstapel
  - $moves(1) = 1$
  - $moves(n+1) = moves(n) + 1 + moves(n) = 1 + 2 * moves(n)$
- Annahme: $moves(n) = 2^n - 1$
- Beweis durch Induktion

Aufwandsbetrachtung

- Wieviele Scheiben werden während der Lösung bewegt?
- $moves(n)$: Anzahl der Verschiebungen für n Scheiben auf dem Ausgangsstapel
  - $moves(1) = 1$
  - $moves(n+1) = moves(n) + 1 + moves(n) = 1 + 2 * moves(n)$
- Annahme: $moves(n) = 2^n - 1$
- Beweis durch Induktion
  - $moves(1) = 2^1 - 1 = 1$
  - $moves(n+1) = 1 + 2 * (2^n - 1) = 1 + 2^{n+1} - 2 = 2^{n+1} - 1$

Das Wichtigste in Kürze

- Rekursion
  - Aufruf einer Methode durch sich selbst
  - benötigt zwingend passende Abbruchbedingung
  - Darstellung als Baum
- Lineare, baumartige und verschachtelte Rekursion
- Verwendung eines Stapels bei Methodenaufrufen
- Türme von Hanoi: Einfache Beschreibung der Methode durch Rekursion

Das Wichtigste in Kürze

- Rekursion
  - Aufruf einer Methode durch sich selbst
  - benötigt zwingend passende Abbruchbedingung
  - Darstellung als Baum
- Lineare, baumartige und verschachtelte Rekursion
- Verwendung eines Stapels bei Methodenaufrufen
- Türme von Hanoi: Einfache Beschreibung der Methode durch Rekursion
- Dynamische Programmierung
  - Rekursionsformel
  - Speichern und Wiederverwenden von Zwischenergebnissen

Literatur

H. P. Gumm und M. Sommer.
Einführung in die Informatik — Kapitel 2.8.
Oldenburg Verlag, 7. Ausgabe, 2006, ISBN 978-3-486-58115-7.