

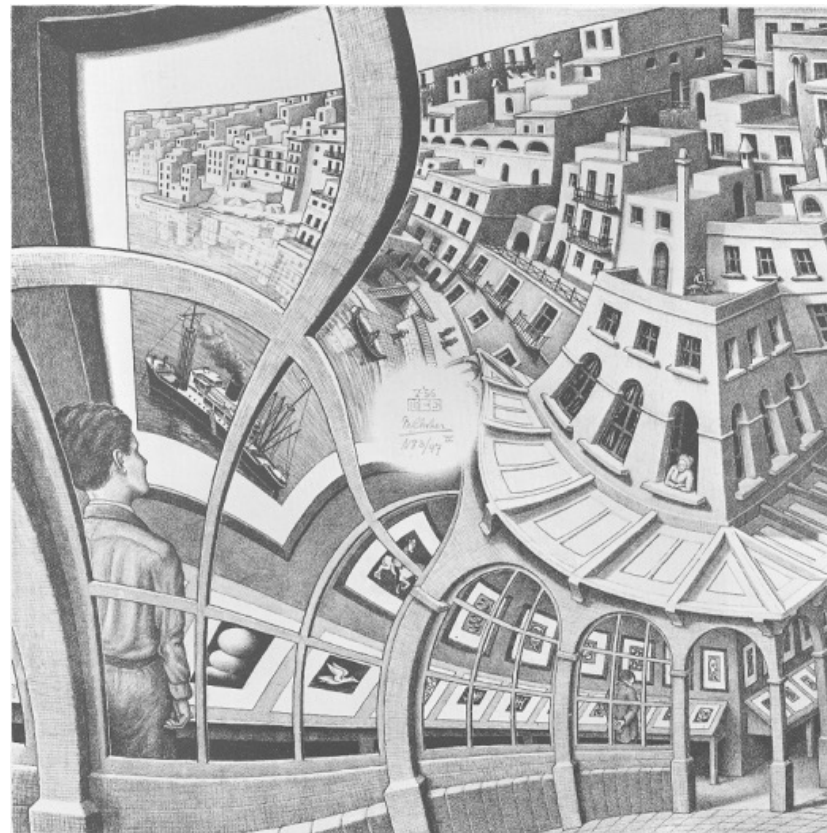
# Kapitel 7: Rekursion

- Grundbegriffe
- Beispiele
  - Türme von Hanoi
  - Größter gemeinsamer Teiler
  - Graphische Darstellung von Bäumen
  - Linear verkettete Listen
- Teile-und-Herrsche-Verfahren
  - Potenzfunktion
  - Binäre Suche
- Endrekursion
- Rekursion und Keller

# Rekursion

---

- **Rekursion** bedeutet wörtlich **Zurückführen**.
- Rekursion liegt dann vor, wenn eine Funktion, ein Algorithmus, eine Datenstruktur, ein Begriff, etc. durch sich selbst definiert wird.



M. C. Escher, Bildergalerie, 1956.

# Rekursive Datentypen und Funktionen

- Linear verkettete Listen und Bäume (später) sind **rekursiv definierte Datentypen**.
- **Beispiel:** linear verkettete Liste

```
class Node {  
    Node next;  
    int data;  
    // ...  
}
```

Node wird durch sich selbst definiert.

- Eine **rekursive Funktion** ist eine Funktion, die sich selbst aufruft.
- **Beispiel:** Fakultätsfunktion

$fak(0) = 1$

$fak(n) = n * (n-1) * \dots * 2 * 1 = n * fak(n-1)$  falls  $n \geq 1$

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

fak ruft sich selbst auf.

# Aufruf einer rekursiven Funktion - Beispiel

---

```
void main() {  
    fak(3);  
}  
  
static int fak(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n*fak(n-1) ;  
}
```

main()



fak(3):

```
if(3 == 0)  
    return 1;  
else  
    return 3*fak(2);
```

# Aufruf einer rekursiven Funktion - Beispiel

```
void main() {  
    fak(3);  
}  
  
static int fak(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n*fak(n-1) ;  
}
```

main()



fak(3):

```
if(3 == 0)  
    return 1;  
else  
    return 3*fak(2);
```

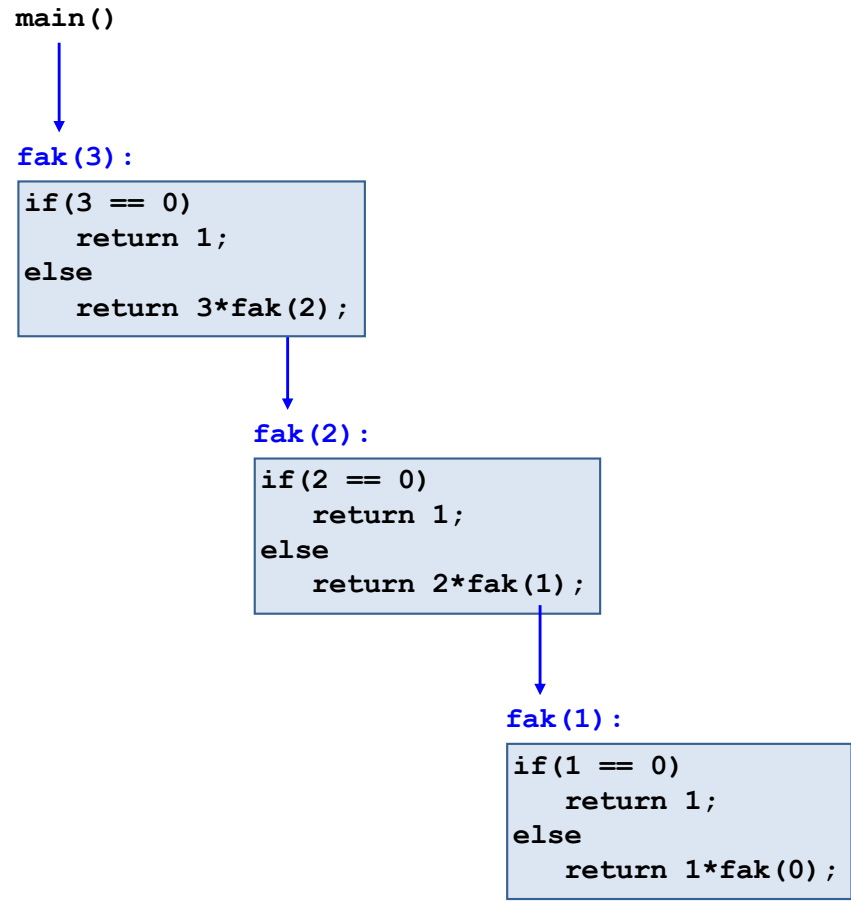


fak(2):

```
if(2 == 0)  
    return 1;  
else  
    return 2*fak(1);
```

# Aufruf einer rekursiven Funktion - Beispiel

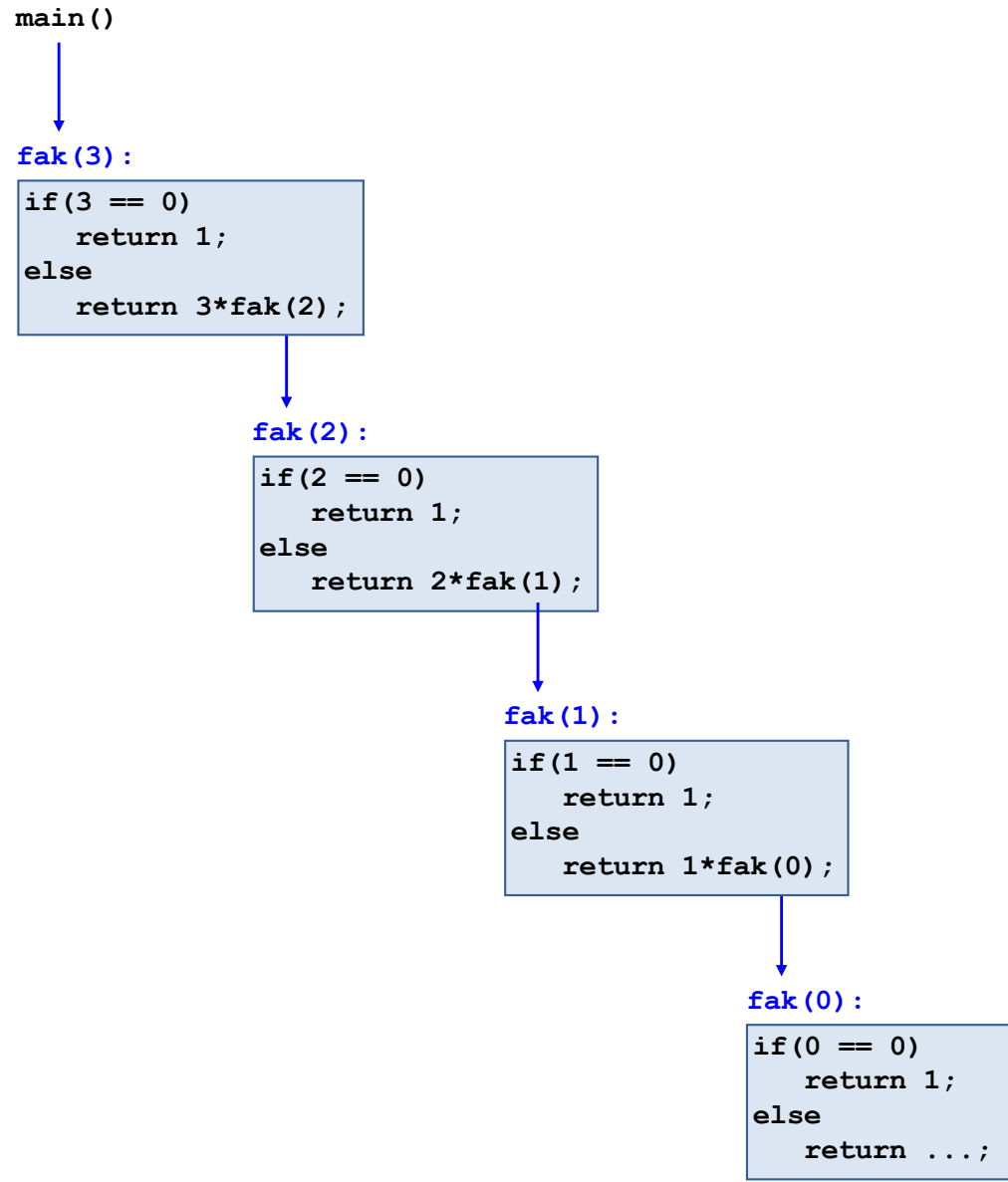
```
void main() {  
    fak(3);  
}  
  
static int fak(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n*fak(n-1) ;  
}
```



# Aufruf einer rekursiven Funktion - Beispiel

```
void main() {
    fak(3);
}

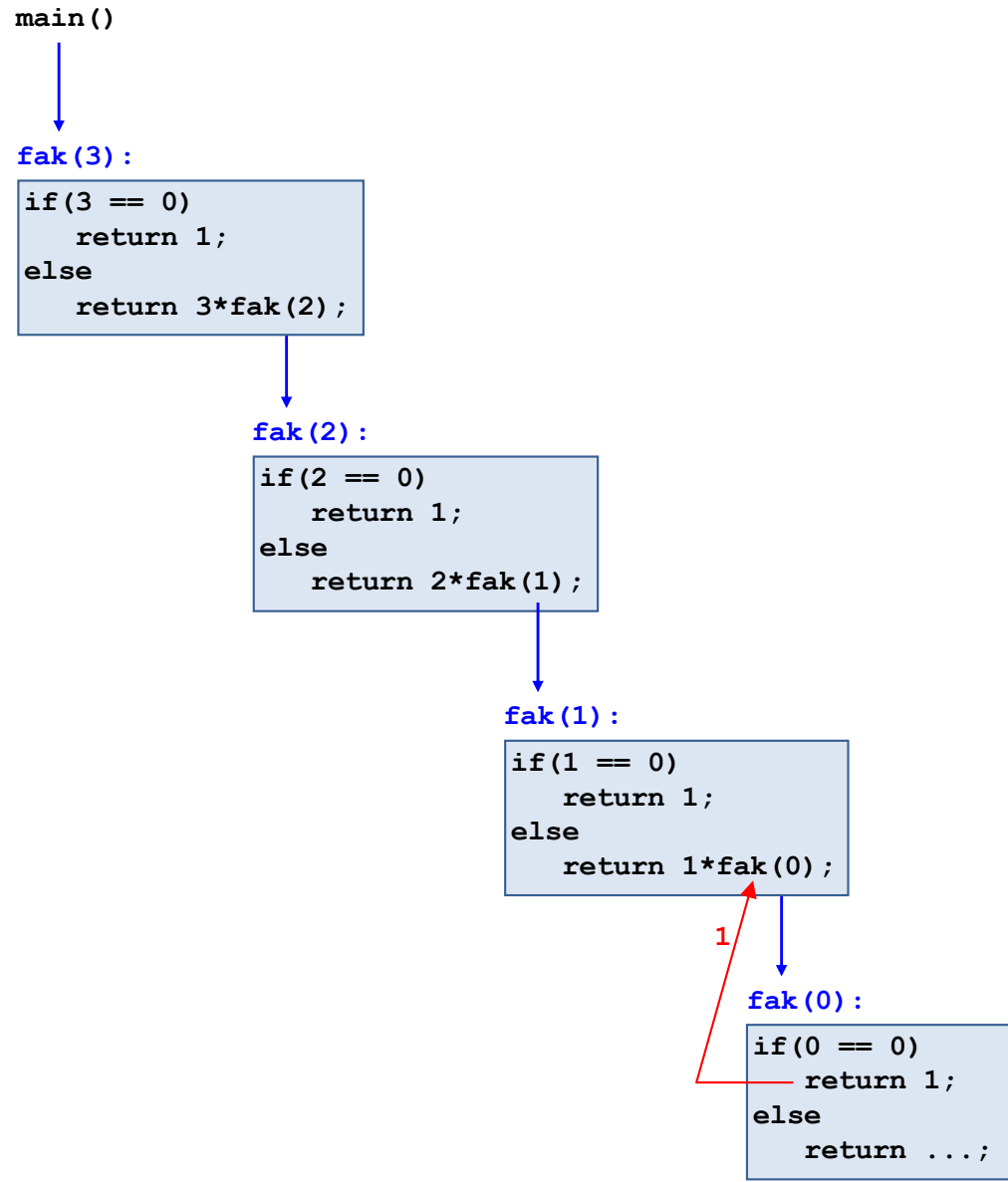
static int fak(int n) {
    if (n == 0)
        return 1;
    else
        return n*fak(n-1) ;
}
```



# Aufruf einer rekursiven Funktion - Beispiel

```
void main() {
    fak(3);
}

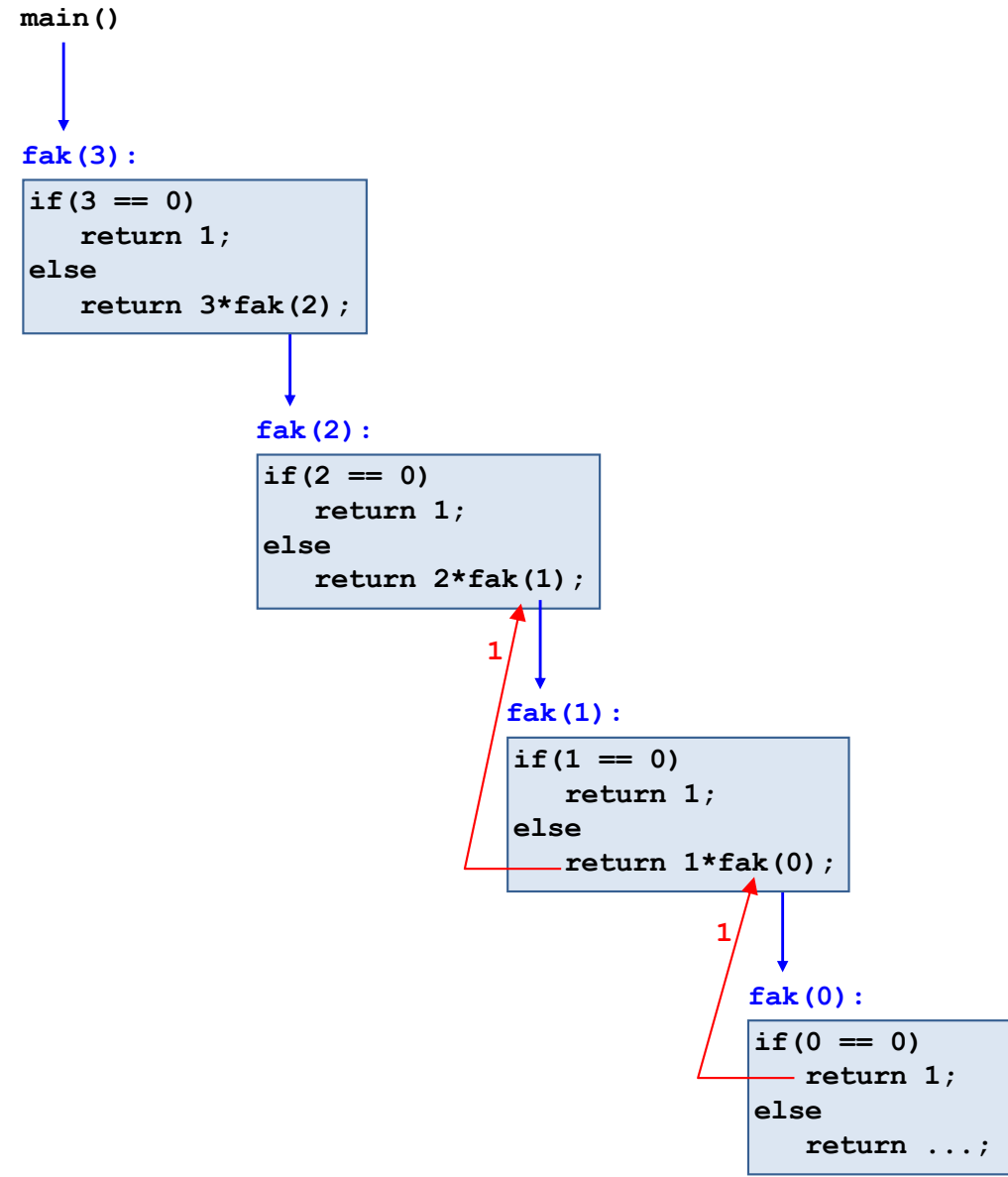
static int fak(int n) {
    if (n == 0)
        return 1;
    else
        return n*fak(n-1) ;
}
```





# Aufruf einer rekursiven Funktion - Beispiel

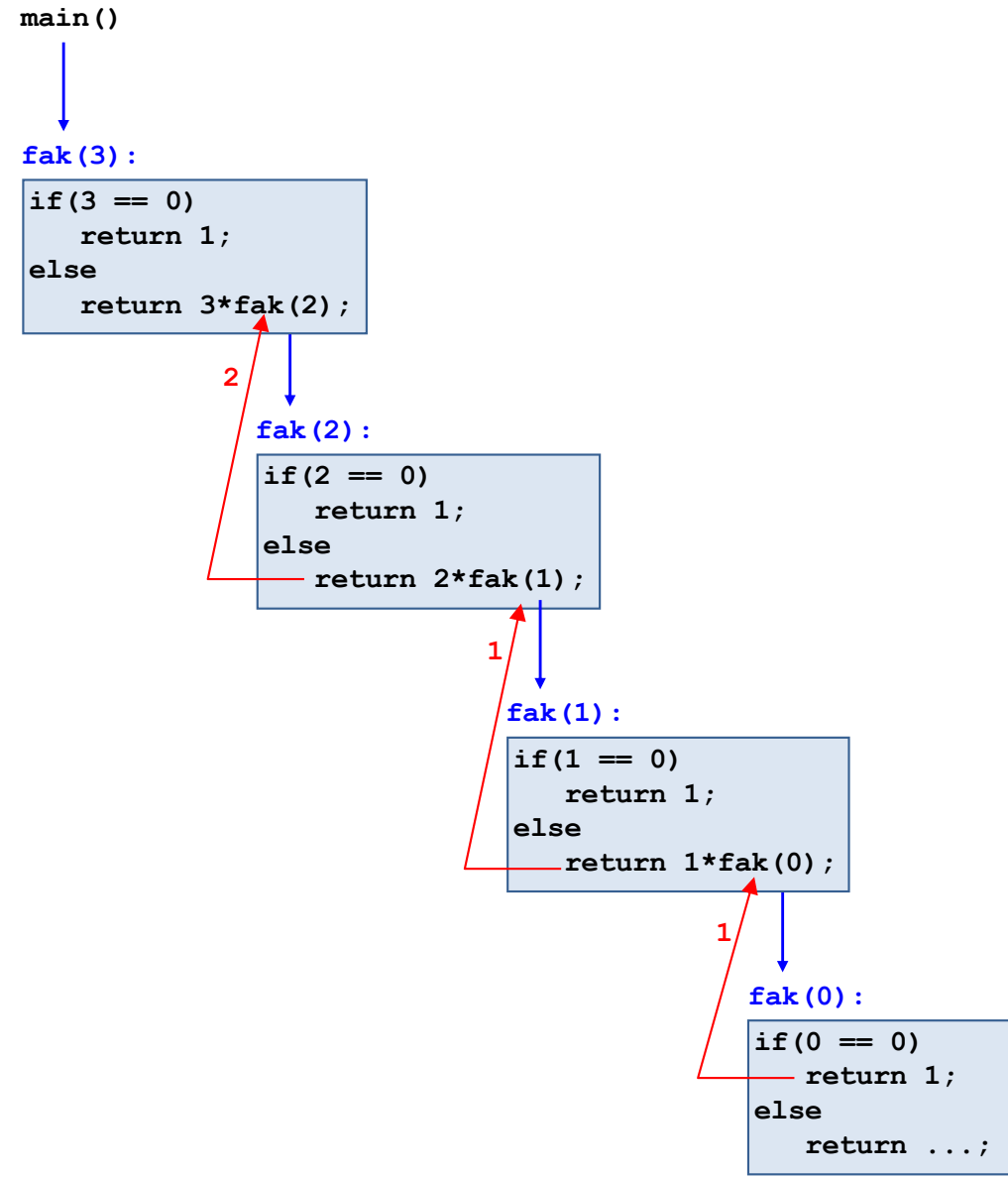
```
void main() {  
    fak(3);  
}  
  
static int fak(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n*fak(n-1) ;  
}
```



# Aufruf einer rekursiven Funktion - Beispiel

```
void main() {
    fak(3);
}

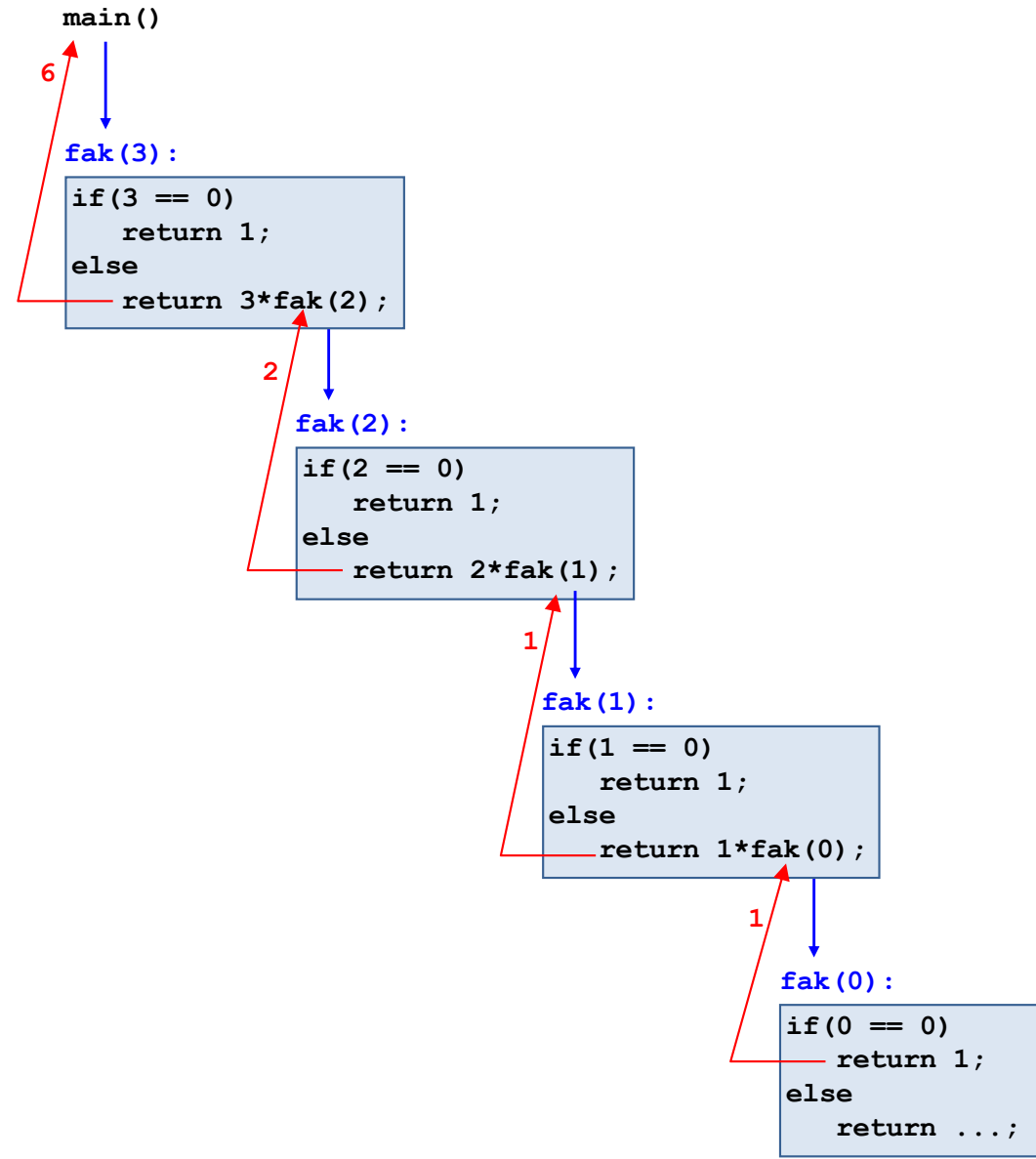
static int fak(int n) {
    if (n == 0)
        return 1;
    else
        return n*fak(n-1) ;
}
```



# Aufruf einer rekursiven Funktion - Beispiel

```
void main() {
    fak(3);
}

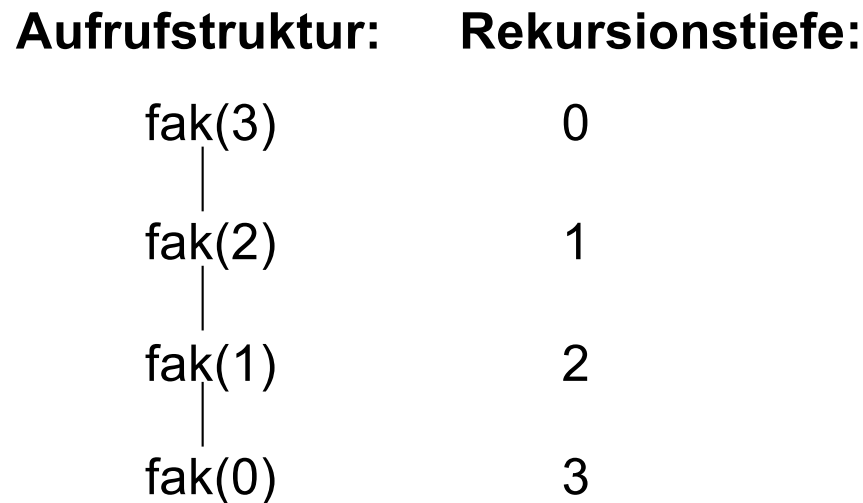
static int fak(int n) {
    if (n == 0)
        return 1;
    else
        return n*fak(n-1) ;
}
```



# Aufrufstruktur und Rekursionstiefe

---

- **Aufrufstruktur:**  
Kompakte Darstellung sämtlicher rekursiver Aufrufe einer rekursiven Funktion
- **Rekursionstiefe:**  
Anzahl der geschachtelten Aufrufe einer rekursiven Funktion.



# Rekursionstiefe und Laufzeitstack

- Zur Laufzeit wird bei jedem Funktionsaufruf ein **Call-Frame** bestehend aus
  - Parameter,
  - Rücksprungadresse und
  - lokale Variablenin den Laufzeit-Stack abgelegt.
- Das bedeutet, dass bei **zu großen Rekursionstiefen** der Laufzeit-Stack überläuft (Stack Overflow Error Exception).
- Also: Zu große Rekursionstiefen vermeiden und insbesondere auf Endlos-Rekursion achten:

```
void main() {  
    fak(3);  
}  
  
static int fak(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n*fak(n) ;  
}
```

**Endlos-Rekursion:  
Stack Overflow Error Exception**

# Einschub: große Zahlen mit BigInteger

- fak(13) führt bereits zu einem arithmetischen Überlauf:
  - 13! = 6\_227\_020\_800
  - fak(13) → 1\_932\_053\_504
- Mit **BigInteger** lassen sich beliebig große ganze Zahlen darstellen und arithmetische Operationen ausführen.

```
public static void main() {  
    System.out.println(fak(new BigInteger("100")));  
}
```

```
public static BigInteger fak(BigInteger n) {  
    if (n.equals(BigInteger.ZERO))  
        return BigInteger.ONE;  
    else return n.multiply(fak(n.subtract(BigInteger.ONE)));  
}
```

```
933262154439441526816992388562667  
004907159682643816214685929638952  
175999932299156089414639761565182  
862536979208272237582511852109168  
64000000000000000000000000000000
```

# Kapitel 7: Rekursion

- Grundbegriffe
- Beispiele
  - Türme von Hanoi
  - Größter gemeinsamer Teiler
  - Graphische Darstellung von Bäumen
  - Linear verkettete Listen
- Teile-und-Herrsche-Verfahren
  - Potenzfunktion
  - Binäre Suche
- Endrekursion
- Rekursion und Keller

# Vorgehensweise bei rekursiver Programmierung

---

## Problemstellung

- Gesucht ist eine rekursive Funktion zur Lösung eines Problems  $P$  der Größe  $n$  ( $n \geq 0$ ).
- Beispiele: fak( $n$ ), Sortieren von  $n$  Zahlen, Suchen von  $x$  in  $n$  Zahlen, ...

## Vorgehensweise

- **Rekursionsfall:**

Reduziere Problem der Größe  $n$  auf ein Problem der Größe  $k$  mit  $0 \leq k < n$  (oder evtl. mehrere Probleme).

Beispiel: bei der Fakultätsfunktion wird fak( $n$ ) zurückgeführt auf  $n \cdot \text{fak}(n-1)$

- **Basisfall (bzw. Basisfälle):**

Löse  $P$  für alle Werte  $n$  direkt, die sich im Rekursionsfall nicht weiter reduzieren lassen.

Beispiel: bei der Fakultätsfunktion ist der Basisfall fak(0).

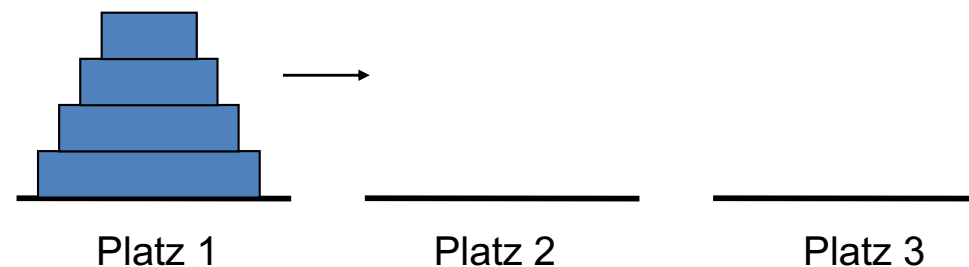


# Türme von Hanoi (1)

---

## Aufgabenstellung

- n Scheiben unterschiedlichen Durchmessers, die der Größe nach sortiert übereinander liegen, bilden mit der größten Scheibe unten einen Turm. Der Turm soll von einem Platz 1 nach einem Platz 2 transportiert werden.
- Dabei steht ein Hilfsplatz 3 zur Verfügung.
- Es darf jeweils nur die oberste Scheibe eines Turms bewegt werden.
- Außerdem darf auf eine Scheibe nur eine kleinere Scheibe gelegt werden.



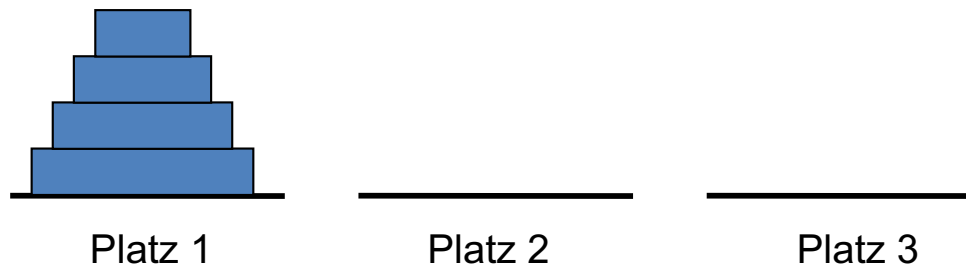
# Türme von Hanoi (2)

---

## Methode `bewegeTurm`

```
void bewegeTurm(int n, int s, int z, int h);
```

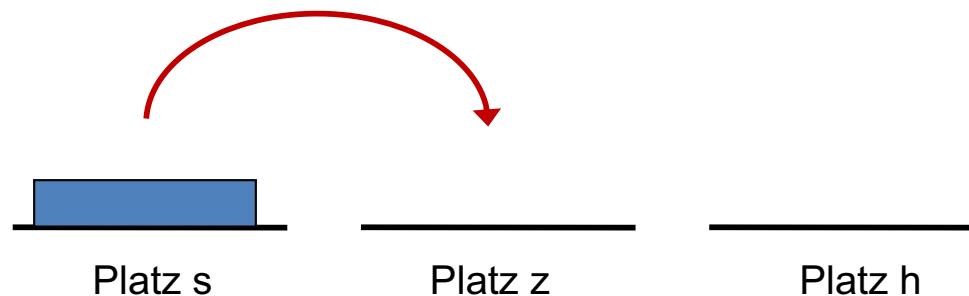
gibt die notwendigen Scheibenbewegungen aus, um ein Turm mit  $n$  Scheiben vom Startplatz  $s$  zum Zielplatz  $z$  zu bewegen. Dabei ist  $h$  ein zusätzlicher Hilfsplatz.



**Ziel: Rekursive Lösung für `bewegeTurm`**

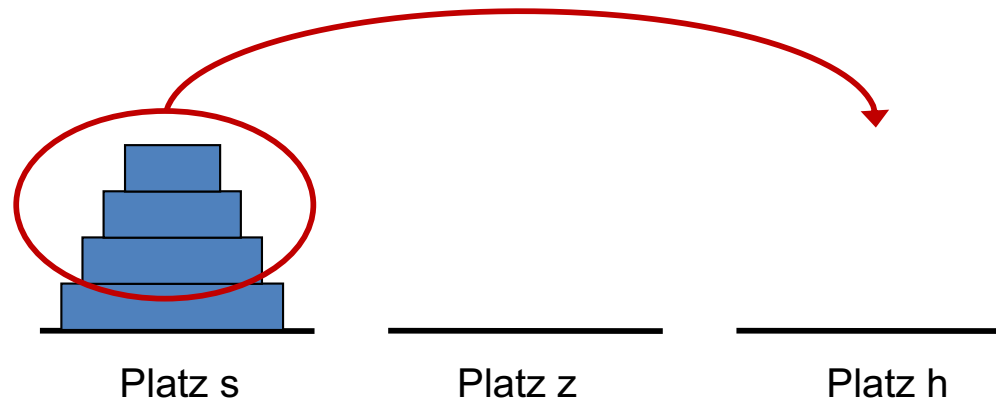
# Türme von Hanoi – Rekursive Lösung

```
static void bewegeTurm(int n, int s, int z, int h)
{
    if (n == 1)
        System.out.println("Bewege Scheibe von " + s + " nach " + z);
    else {
        bewegeTurm(n-1, s, h, z);
        System.out.println("Bewege Scheibe von " + s + " nach " + z);
        bewegeTurm(n-1, h, z, s);
    }
}
```



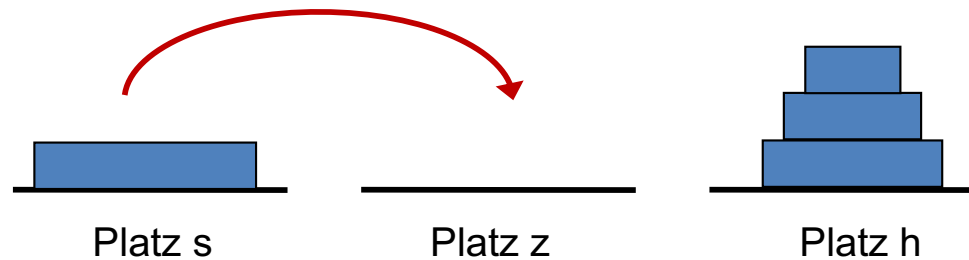
# Türme von Hanoi – Rekursive Lösung

```
static void bewegeTurm(int n, int s, int z, int h)
{
    if (n == 1)
        System.out.println("Bewege Scheibe von " + s + " nach " + z);
    else {
        bewegeTurm(n-1, s, h, z);
        System.out.println("Bewege Scheibe von " + s + " nach " + z);
        bewegeTurm(n-1, h, z, s);
    }
}
```



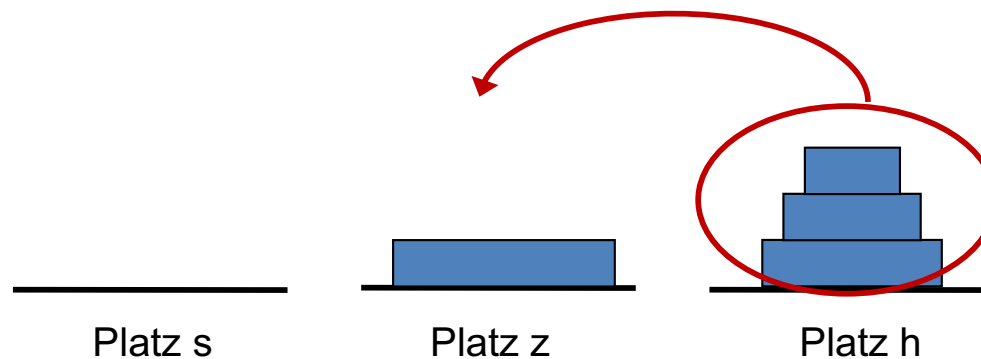
# Türme von Hanoi – Rekursive Lösung

```
static void bewegeTurm(int n, int s, int z, int h)
{
    if (n == 1)
        System.out.println("Bewege Scheibe von " + s + " nach " + z);
    else {
        bewegeTurm(n-1, s, h, z);
        System.out.println("Bewege Scheibe von " + s + " nach " + z);
        bewegeTurm(n-1, h, z, s);
    }
}
```



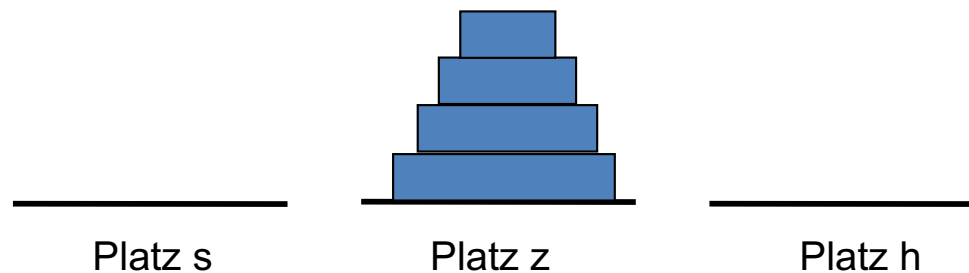
# Türme von Hanoi – Rekursive Lösung

```
static void bewegeTurm(int n, int s, int z, int h)
{
    if (n == 1)
        System.out.println("Bewege Scheibe von " + s + " nach " + z);
    else {
        bewegeTurm(n-1, s, h, z);
        System.out.println("Bewege Scheibe von " + s + " nach " + z);
        bewegeTurm(n-1, h, z, s);
    }
}
```



# Türme von Hanoi – Rekursive Lösung

```
static void bewegeTurm(int n, int s, int z, int h)
{
    if (n == 1)
        System.out.println("Bewege Scheibe von " + s + " nach " + z);
    else {
        bewegeTurm(n-1, s, h, z);
        System.out.println("Bewege Scheibe von " + s + " nach " + z);
        bewegeTurm(n-1, h, z, s);
    }
}
```



# Animation des Aufrufs `bewegeTurm(3,1,2,3)`

---



`bTurm(3,1,2,3)`

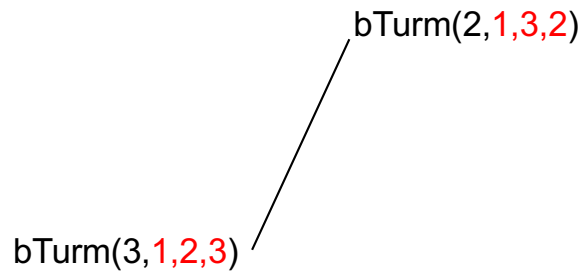
Plätze

Zeitschritte



# Animation des Aufrufs `bewegeTurm(3,1,2,3)`

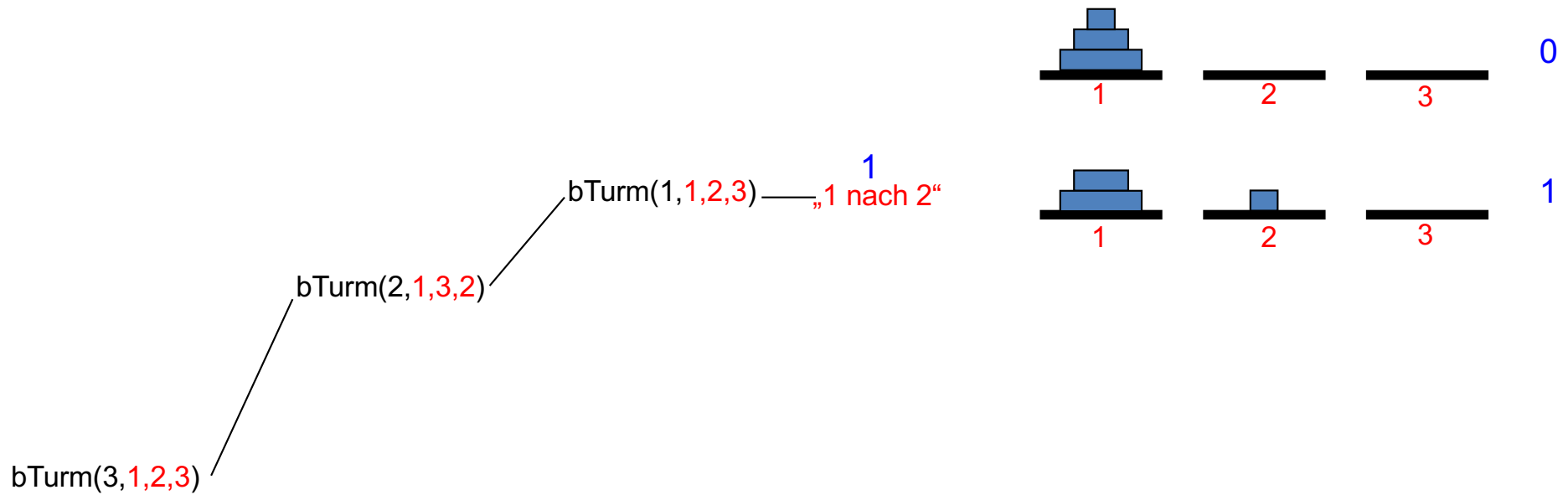
---



Plätze

Zeitschritte

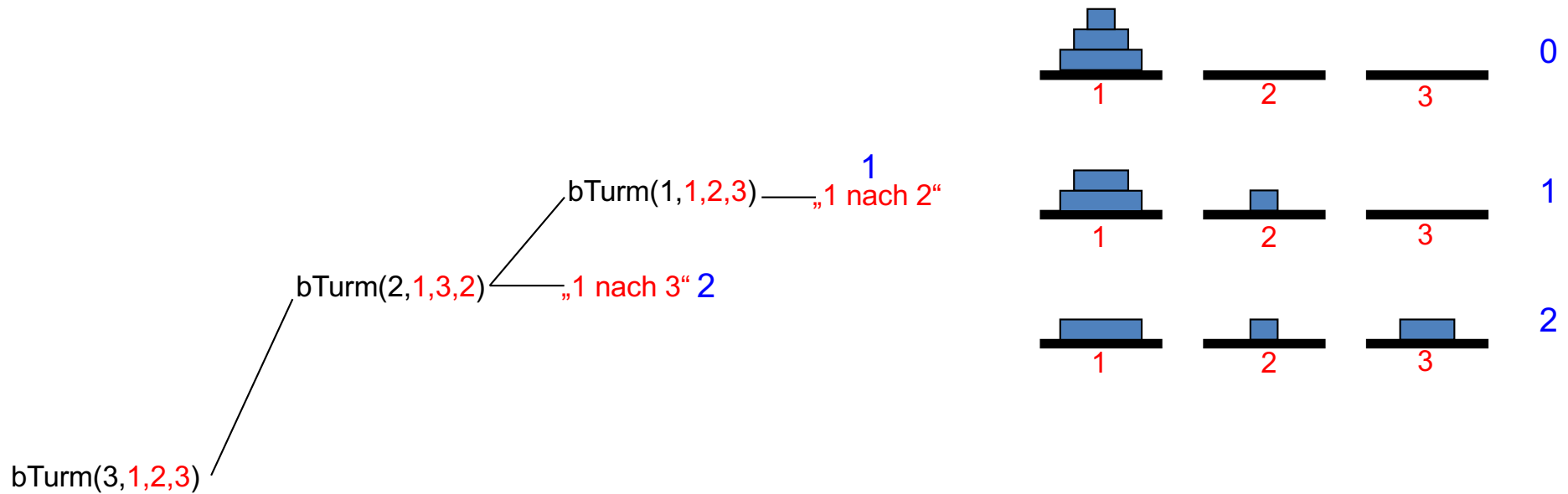
# Animation des Aufrufs `bewegeTurm(3,1,2,3)`



Plätze

Zeitschritte

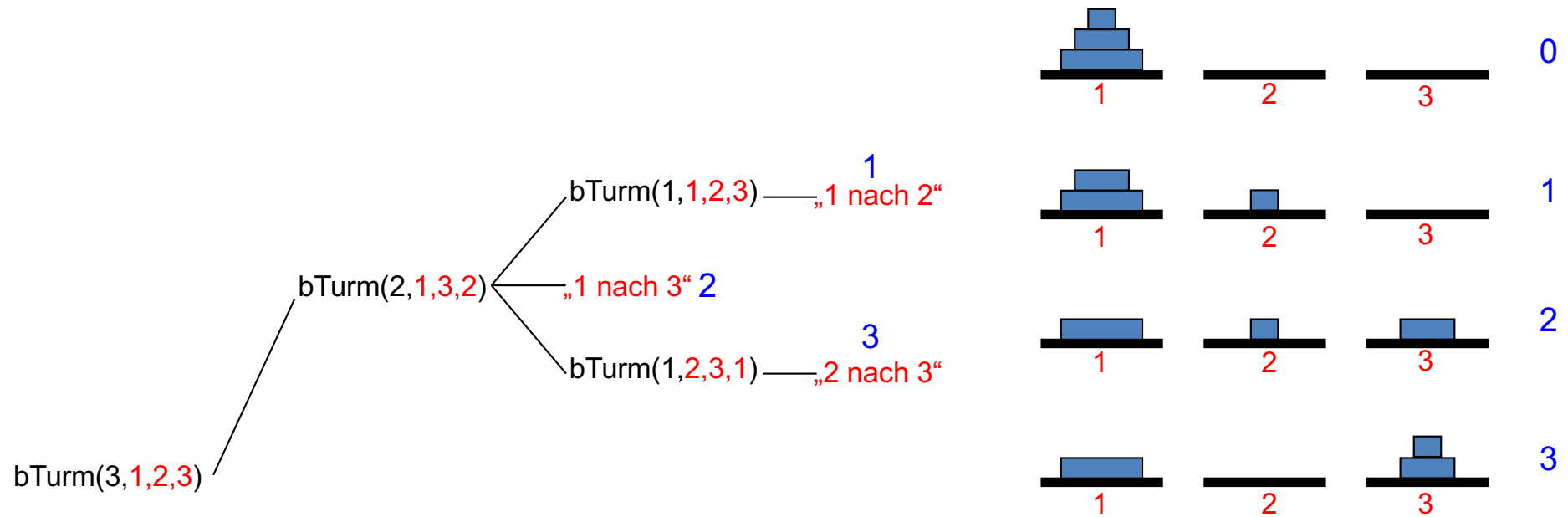
# Animation des Aufrufs `bewegeTurm(3,1,2,3)`



Plätze

Zeitschritte

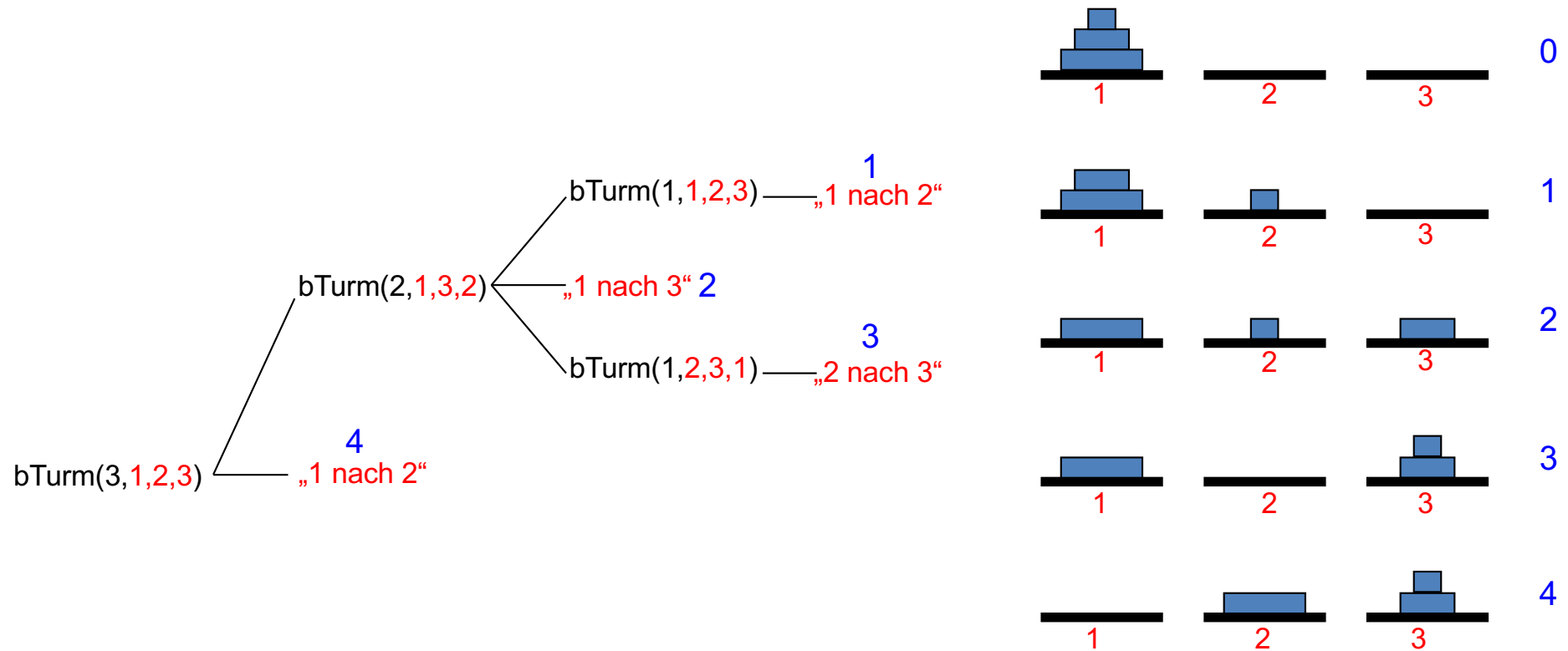
# Animation des Aufrufs `bewegeTurm(3,1,2,3)`



Plätze

Zeitschritte

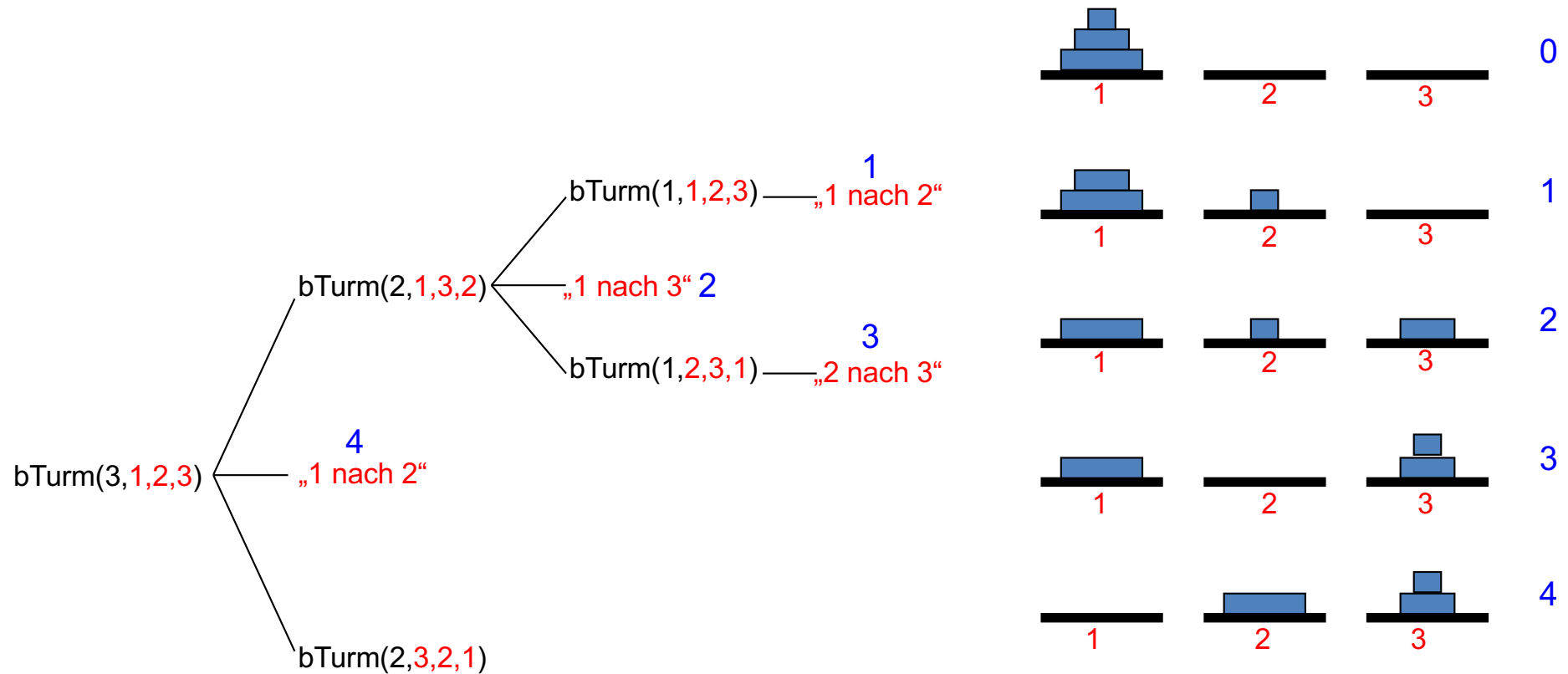
# Animation des Aufrufs `bewegeTurm(3,1,2,3)`



Plätze

Zeitschritte

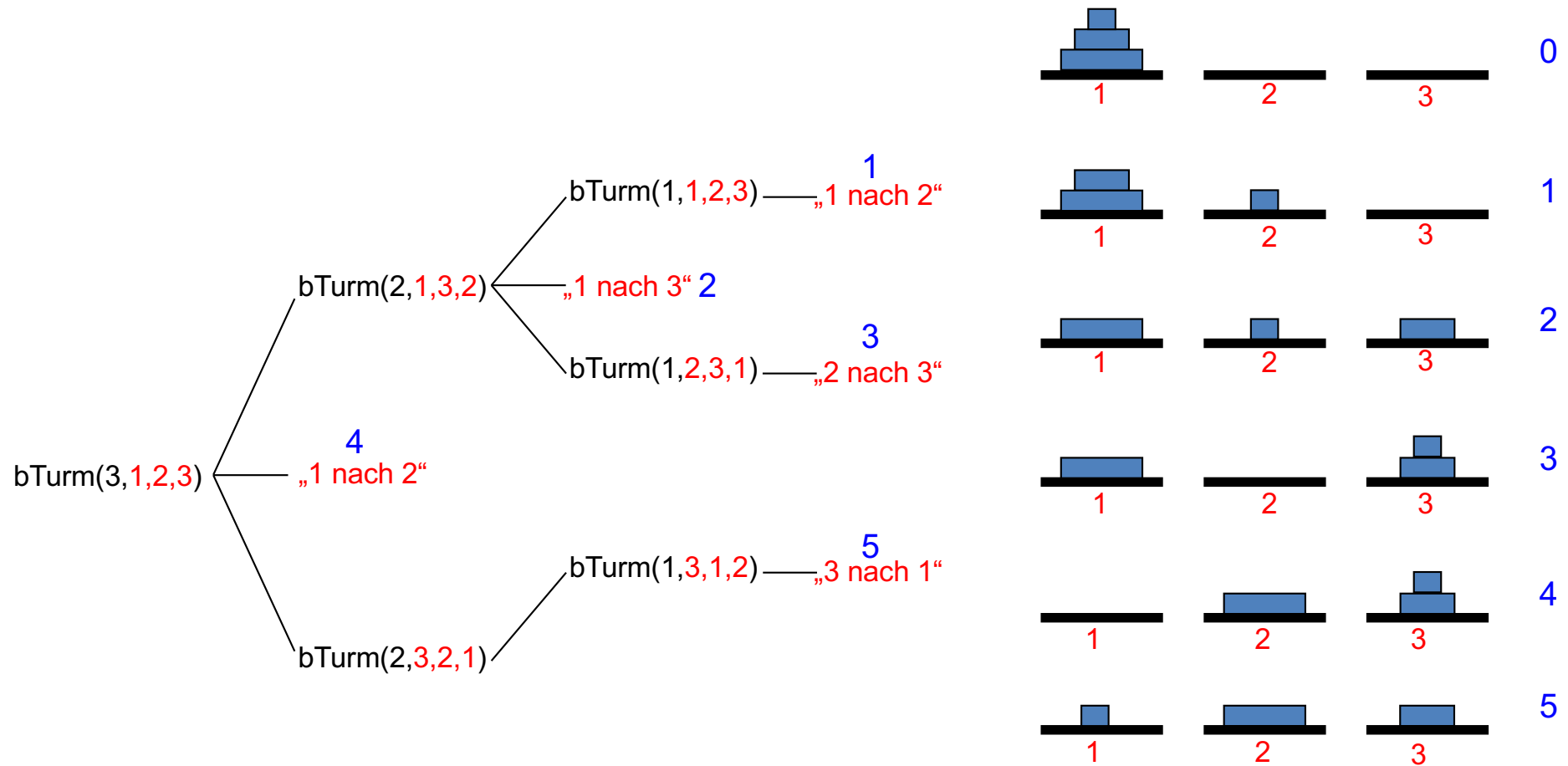
# Animation des Aufrufs `bewegeTurm(3,1,2,3)`



Plätze

Zeitschritte

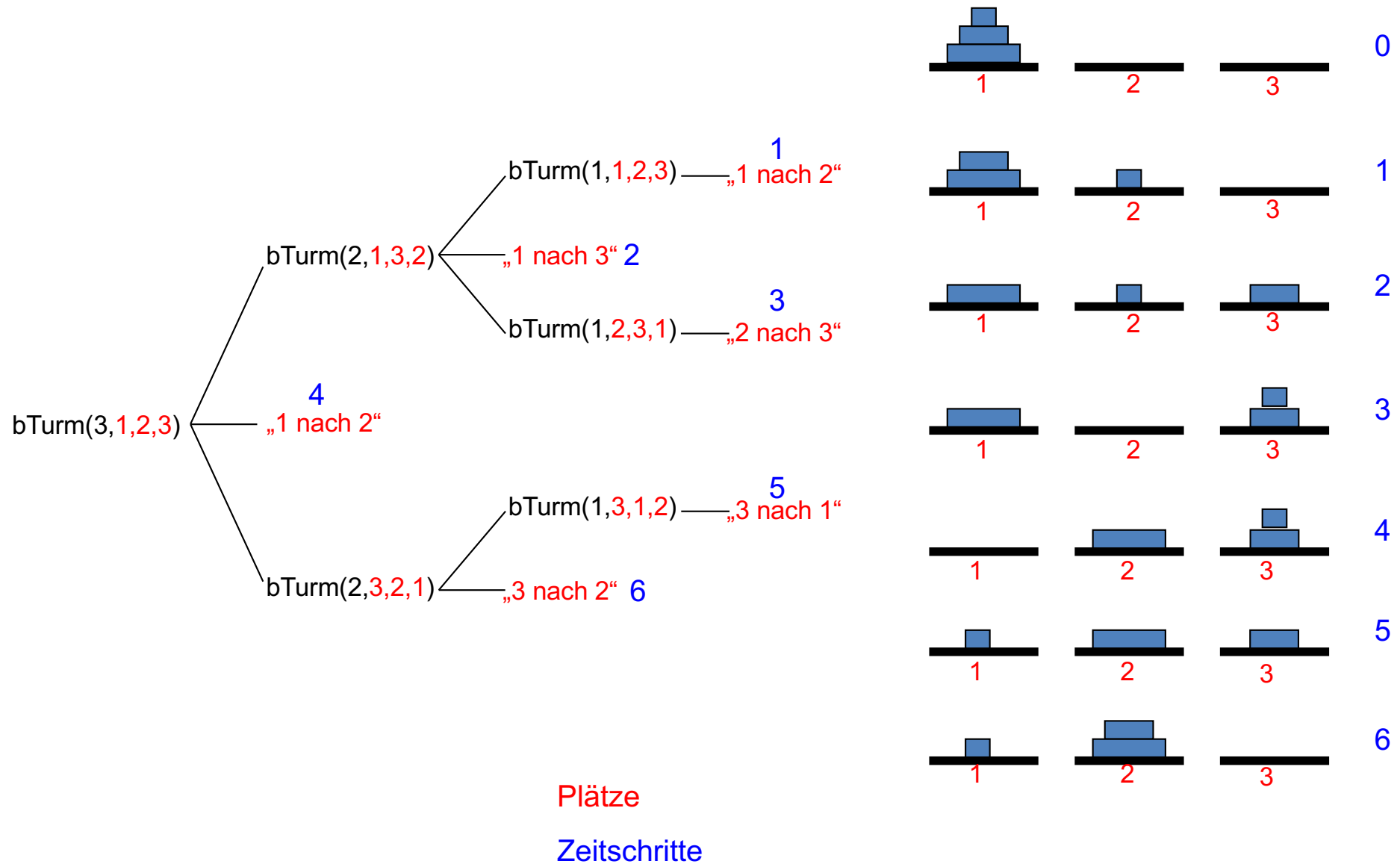
# Animation des Aufrufs `bewegeTurm(3,1,2,3)`



Plätze

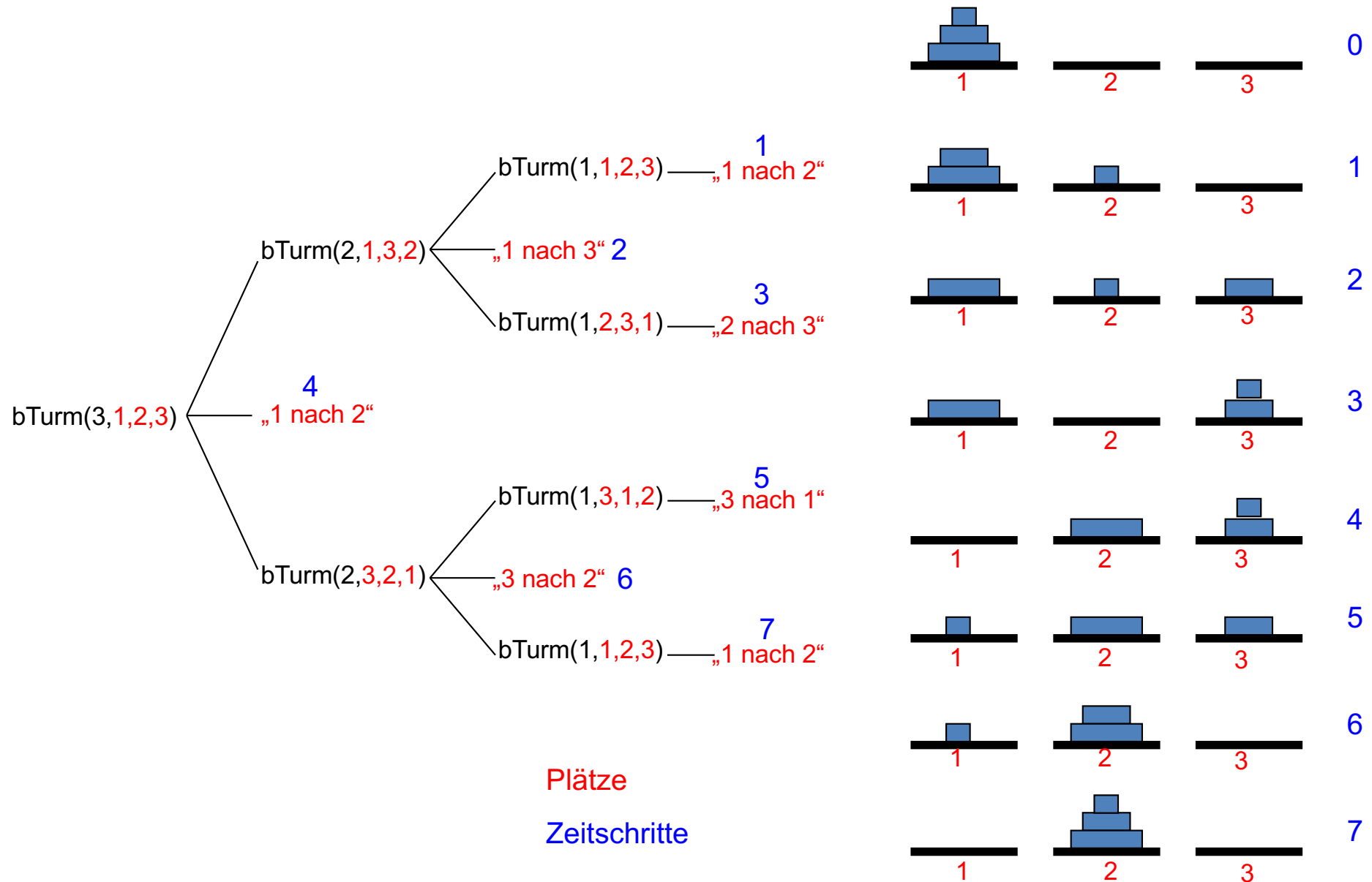
Zeitschritte

# Animation des Aufrufs `bewegeTurm(3,1,2,3)`





# Animation des Aufrufs bewegeTurm(3,1,2,3)



# Türme von Hanoi – Aufgabe 7.1

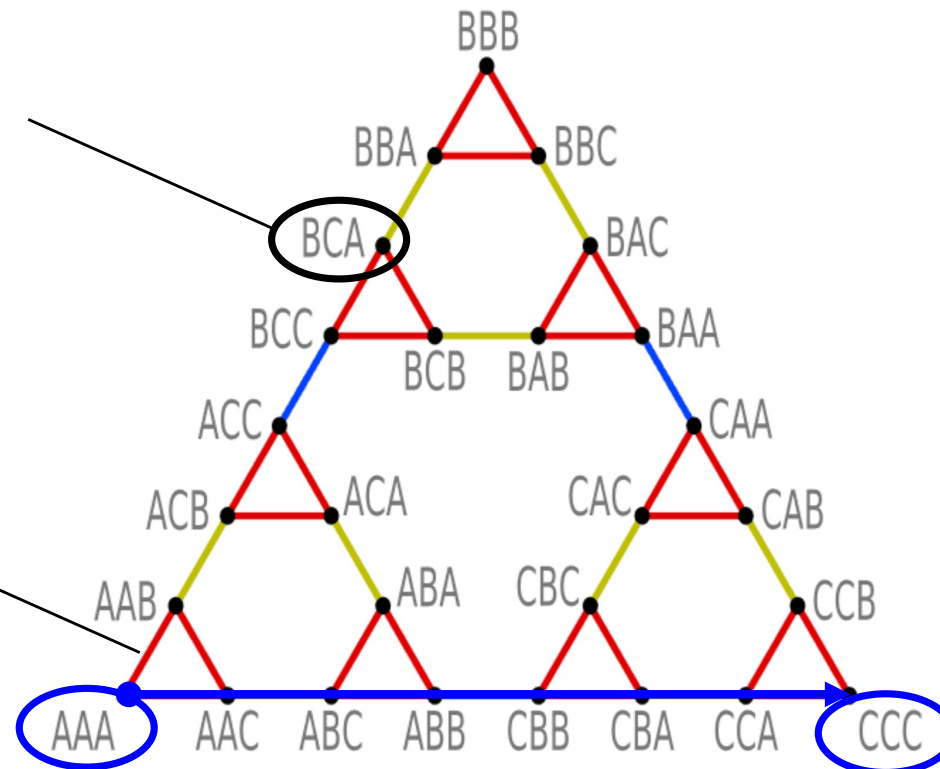
---

- a) Wie groß ist die maximale Rekursionstiefe  $R(n)$  bei Aufruf von `bewegeTurm(n,1,2,3)`?
- b) Wieviel Scheiben  $S(n)$  müssen transportiert werden, um einen Turm der Größe  $n$  vom Start- zum Zielplatz zu bewegen?

# Einschub: Suchraum bei Türme von Hanoi für n = 3 Scheiben

- Zustand BCA:
  - größte Scheibe auf Platz B
  - mittlere Scheibe auf Platz C
  - kleinste Scheibe auf Platz A
- Insgesamt:  
 $3 \cdot 3 \cdot 3 = 81$  Zustände

bewege kleinste  
Scheibe von  
A nach B



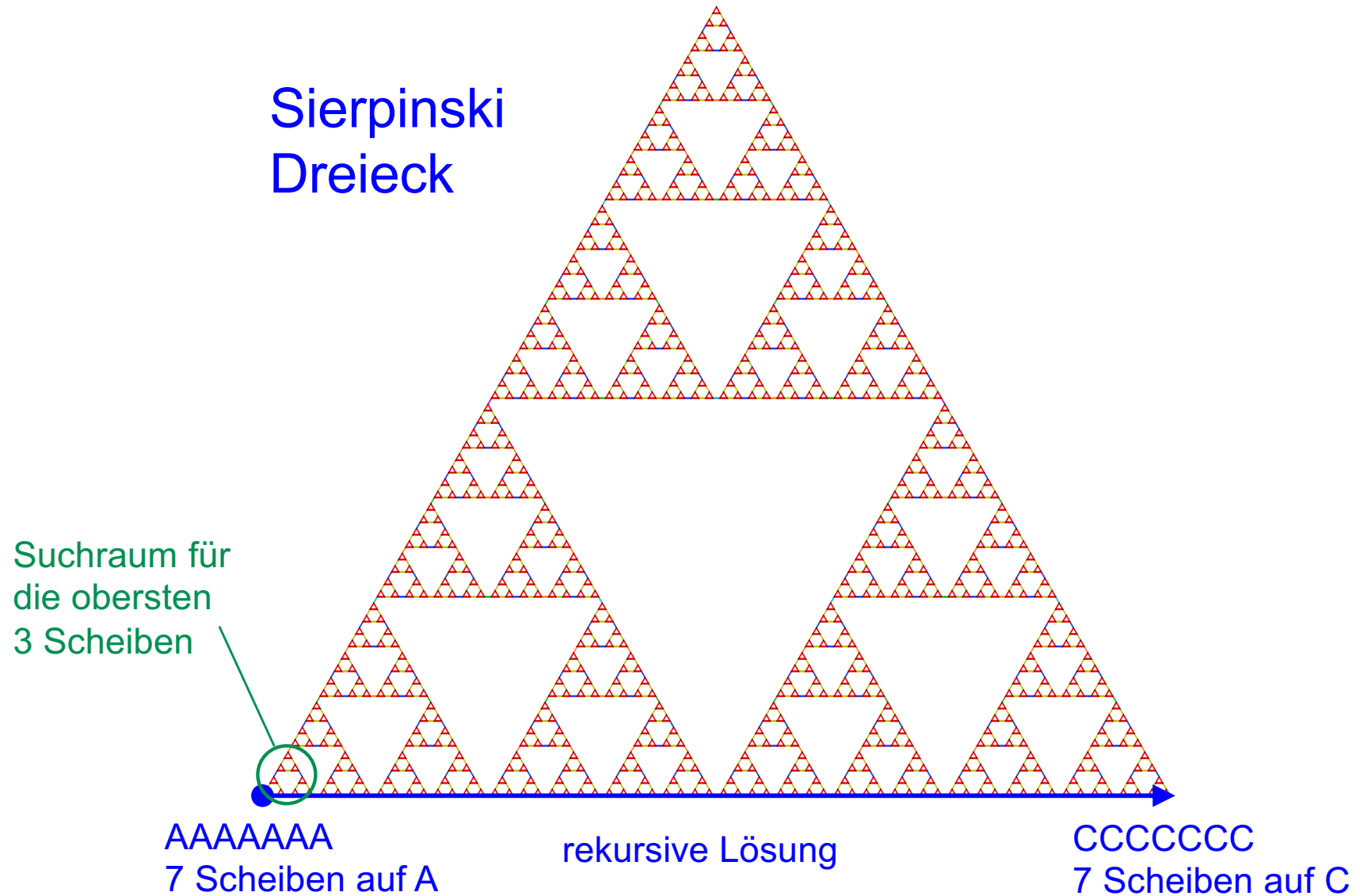
Startzustand:  
3 Scheiben  
auf Platz A

rekursive Lösung  
ist die kürzeste  
Lösung

Zielzustand:  
3 Scheiben  
auf Platz C

[https://de.wikipedia.org/wiki/Türme\\_von\\_Hanoi](https://de.wikipedia.org/wiki/Türme_von_Hanoi)

# Einschub: Suchraum bei Türme von Hanoi für n = 7 Scheiben



[https://de.wikipedia.org/wiki/Türme\\_von\\_Hanoi](https://de.wikipedia.org/wiki/Türme_von_Hanoi)

# Einschub: Meeresschnecke *Cymbiola innexa*

---



Das Gehäuse von *Cymbiola innexa* zeigt, dass in der Natur das Sierpinski-Dreieck vorkommt.

<https://www.spiegel.de/fotostrecke/muschelgehaeuse-muster-mit-formeln-erklaeren-fotostrecke-46503.html>

# Größter gemeinsamer Teiler – ggt (1)

---

## Aufgabenstellung

Gesucht ist eine rekursive Funktion  $\text{ggt}(n,m)$  zur Berechnung des größten gemeinsamen Teilers zweier ganzer Zahlen  $n, m \geq 0$ .

## Es gilt folgende Eigenschaft von ggt:

$$\text{ggt}(m, n) = \text{ggt}(n, m \bmod n) \text{ für } n > 0$$

## Begründung:

- jeder Teiler von  $m$  und  $n$  ist auch Teiler von  $n$  und  $m \bmod n$
- jeder Teiler von  $n$  und  $m \bmod n$  ist auch Teiler von  $m$  und  $n$

## Beispiel:

- 4 ist Teiler von 24 und 16 ist damit auch Teiler von 16 und  $24 \bmod 16 = 8$
- 4 ist Teiler von 16 und  $24 \bmod 16 = 8$  und ist damit auch Teiler von 24 und 16

# Größter gemeinsamer Teiler – ggt (2)

## Rekursive Lösung:

ggt(m, n) lässt sich also reduzieren auf ggt(n, m mod n)

```
static int ggt(int m, int n)
{
    if (n == 0)
        return m;
    else
        return ggt(n, m%n);
}
```

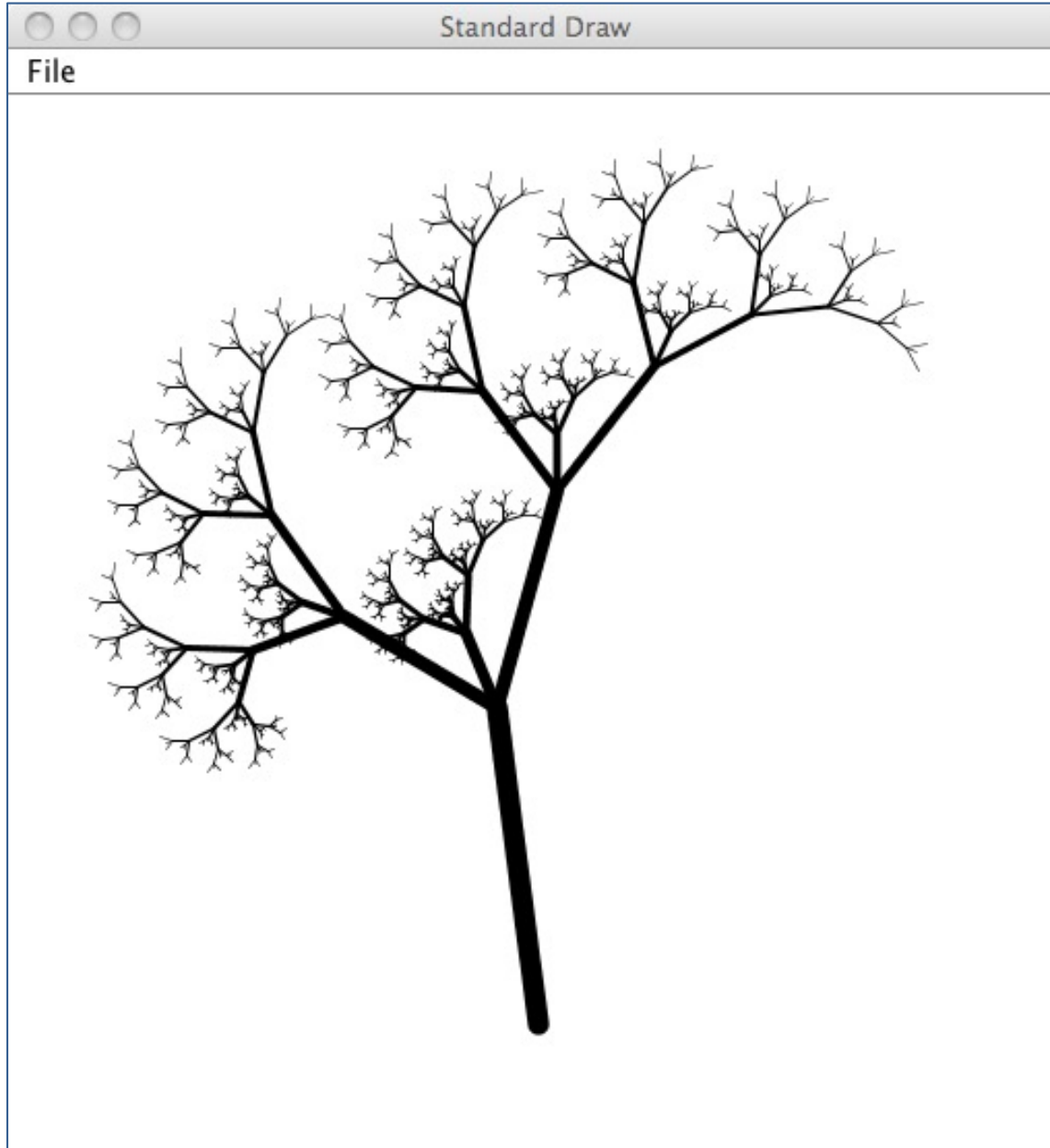
m	n	m mod n
64	24	16
24	16	8
16	8	0
8	0	

ggt(64,24) → 8

## Aufgabe 7.2:

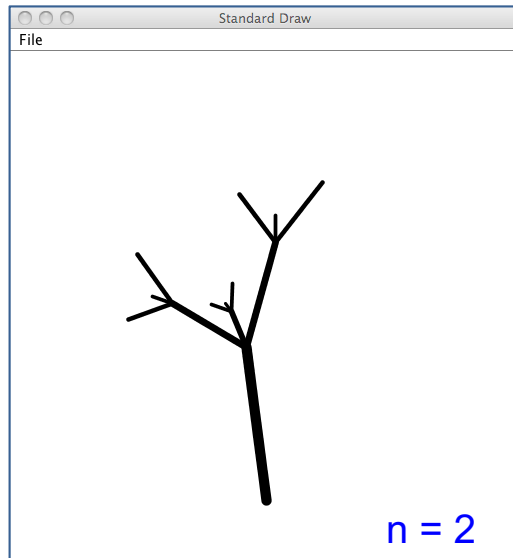
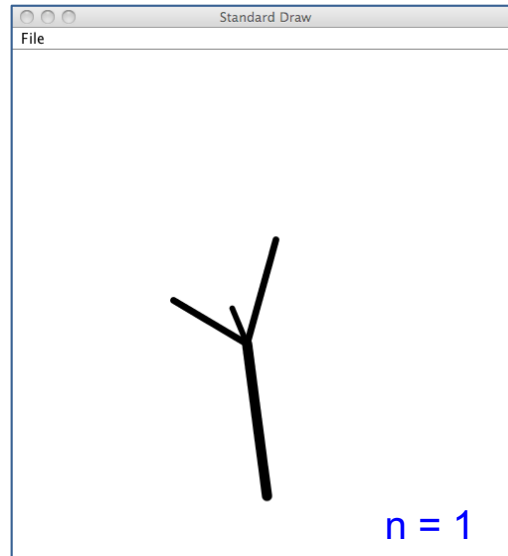
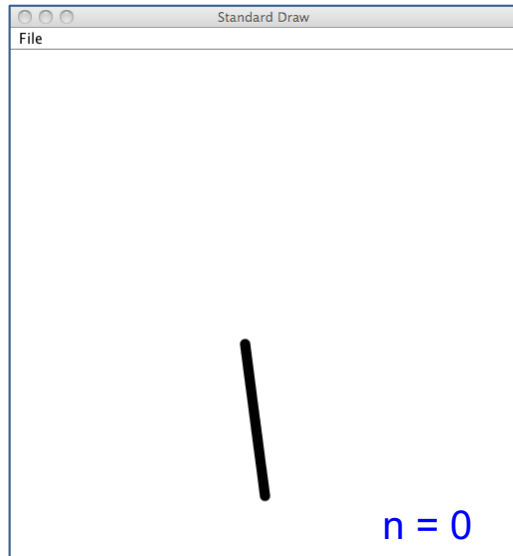
Konstruieren Sie ein Beispiel mit besonders großer Rekursionstiefe.  
Wie müssen m und n aussehen?

# Grafische Darstellung eines Baums (1)

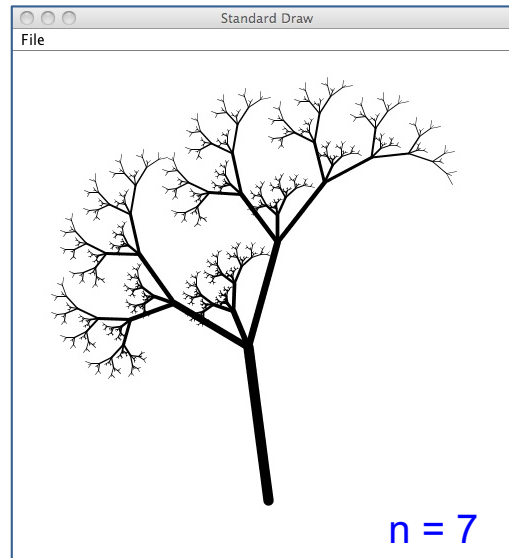




# Grafische Darstellung eines Baums (2)



...

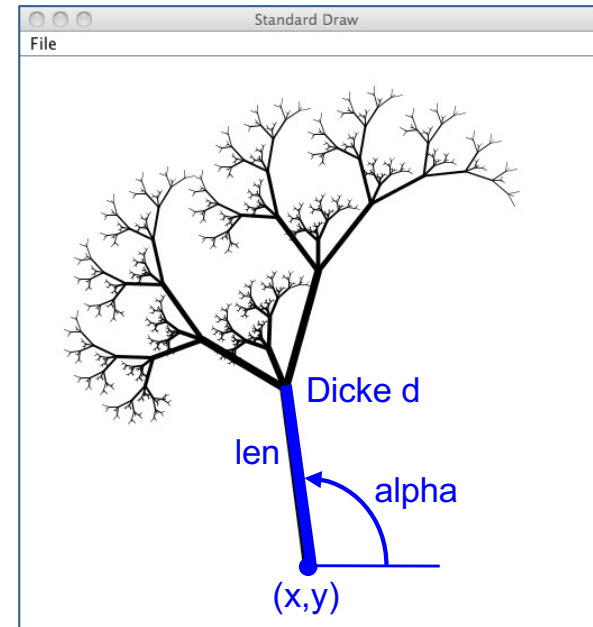


Verästeltiefe  $n$

# Grafische Darstellung eines Baums (3)

```
static void draw(double x, double y, double alpha,
                double len, double d, int n) {
    if (n >= 0) {
        double xe = x + len*Math.cos(alpha);
        double ye = y + len*Math.sin(alpha);
        StdDraw.setPenRadius(d);
        StdDraw.line(x, y, xe, ye);
        draw(xe, ye, alpha+0.90, len*0.55, d/1.5, n-1);
        draw(xe, ye, alpha+0.25, len*0.25, d/1.8, n-1);
        draw(xe, ye, alpha-0.40, len*0.70, d/1.5, n-1);
    }
}
```

```
public static void main() {
    StdDraw.setXscale(-6, +6);
    StdDraw.setYscale(-1, +11);
    draw(0, 0, 1.7, 4.0, 0.02, 7);
}
```



Die drei Äste sind gegenüber aktuellem Ast gedreht um:

$\alpha + 0.90 = \alpha + 51.6^\circ$

$\alpha + 0.25 = \alpha + 14.3^\circ$

$\alpha - 0.40 = \alpha - 22.9^\circ$

Neigung des Baumstamms ist  $1.7 = 97.4^\circ$ .

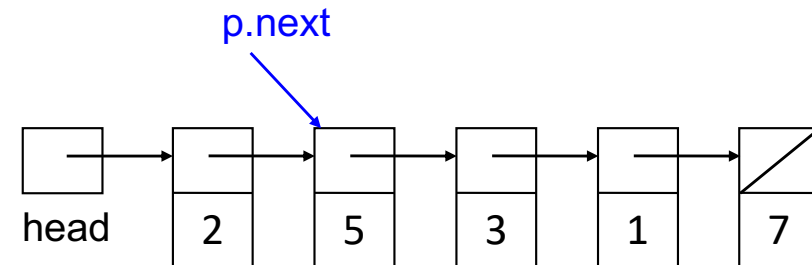
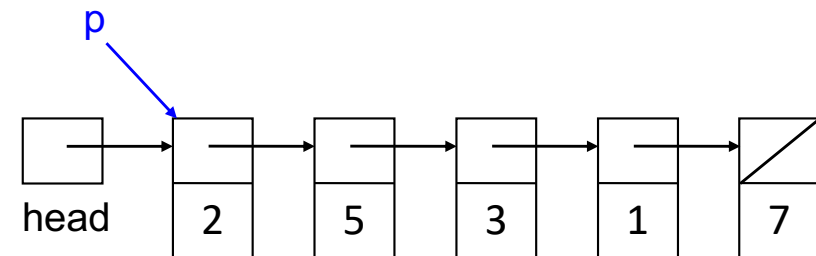
- draw zeichnet einen Ast vom Startpunkt (x,y) mit der Neigung alpha (in rad), der Länge len und der Dicke d und zeichnet am Endpunkt des Asts (xe,ye) drei weitere Äste mit jeweils Verästeltiefe n-1.
- StdDraw ist eine Klasse von <http://introcs.cs.princeton.edu/home/> und gestattet einfache Zeichenoperationen in einem Fenster.

# Rekursion über linear verkettete Listen (1)

## Ansatz

Problem für Liste `p` wird zurückgeführt auf Problem für Liste `p.next`.

Beachte, dass Liste `p.next` ein Knoten weniger enthält.



## Beispiel:

rekursives Ausgeben aller Knoten

```
void printR(Node p) {  
    if (p != null) {  
        System.out.println(p.data);  
        printR(p.next);  
    }  
}
```

# Rekursion über linear verkettete Listen (2)

---

```
public class LinkedList {  
  
    private static class Node {  
        int data;  
        Node next;  
        Node(Node p, int x) {  
            data = x;  
            next = p;  
        }  
    }  
  
    private Node head;  
  
    public LinkedList() {head = null;}  
  
    public void insert(int x) {  
        head = new Node(head,x);  
    }  
  
    // ...  
}
```

# Rekursion über linear verkettete Listen (3)

```
// ...
```

```
public void printR() {  
    printR(head);  
}
```

**printR:**

Rekursive Ausgabe der linear verketteten Liste.

```
private void printR(Node p) {  
    if (p != null) {  
        System.out.println(p.data);  
        printR(p.next);  
    }  
}
```

**print:**

Zum Vergleich iterative Ausgabe der linear verketteten Liste.

```
public void print() {  
    for (Node p = head; p != null; p = p.next)  
        System.out.println(p.data);  
}
```

```
// ...
```

# Rekursion über linear verkettete Listen (4)

```
public void eraseR(int x) {
    head = eraseR(head, x);
}

private Node eraseR(Node p, int x) {
    if (p == null)
        return null;
    else if (p.data == x)
        return p.next;
    else {
        p.next = eraseR(p.next, x);
        return p;
    }
}
```

## eraseR:

rekursives Löschen des ersten Vorkommens von x in der Liste

```
public void erase(int x) {
    if (head == null)
        return;
    else if (head.data == x)
        head = head.next;
    else {
        Node p = head;
        while (p.next != null && x != p.next.data)
            p = p.next;
        if (p.next != null)
            p.next = p.next.next;
    }
}
```

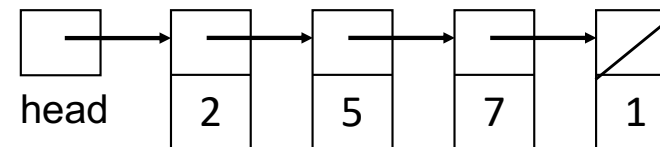
## erase:

Zum Vergleich iteratives Löschen des ersten Vorkommens von x in der Liste

# Animation der rekursiven Methode eraseR

```
public void eraseR(int x) {  
    head = eraseR(head, x);  
}  
  
private Node eraseR(Node p, int x) {  
    if (p == null)  
        return null;  
    else if (p.data == x)  
        return p.next;  
    else {  
        p.next = eraseR(p.next, x);  
        return p;  
    }  
}
```

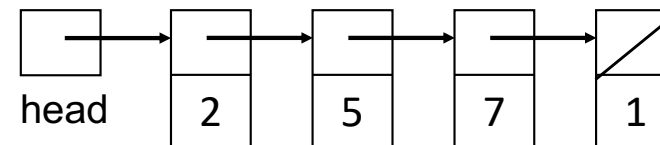
eraseR(7)



# Animation der rekursiven Methode eraseR

```
public void eraseR(int x) {  
    head = eraseR(head, x);  
}  
  
private Node eraseR(Node p, int x) {  
    if (p == null)  
        return null;  
    else if (p.data == x)  
        return p.next;  
    else {  
        p.next = eraseR(p.next, x);  
        return p;  
    }  
}
```

eraseR(7)





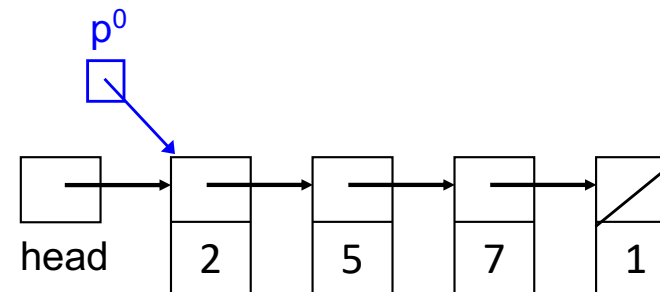
# Animation der rekursiven Methode eraseR

```
public void eraseR(int x) {
    head = eraseR(head, x);
}

private Node eraseR(Node p, int x) {
    if (p == null)
        return null;
    else if (p.data == x)
        return p.next;
    else {
        p.next = eraseR(p.next, x);
        return p;
    }
}
```

$p^i$  = Parameter  $p$  in der Rekursionstiefe  $i$

eraseR(7)      Rekursionstiefe  
↓  
eraseR(head, 7)      0



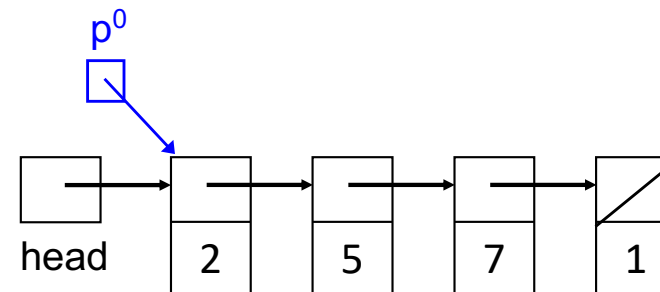
# Animation der rekursiven Methode eraseR

```
public void eraseR(int x) {
    head = eraseR(head, x);
}

private Node eraseR(Node p, int x) {
    if (p == null)
        return null;
    else if (p.data == x)
        return p.next;
    else {
        p.next = eraseR(p.next, x);
        return p;
    }
}
```

$p^i$  = Parameter  $p$  in der Rekursionstiefe  $i$

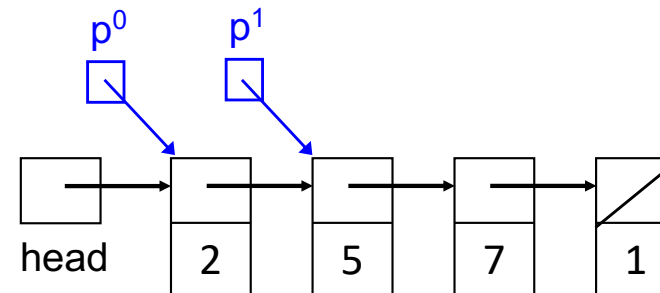
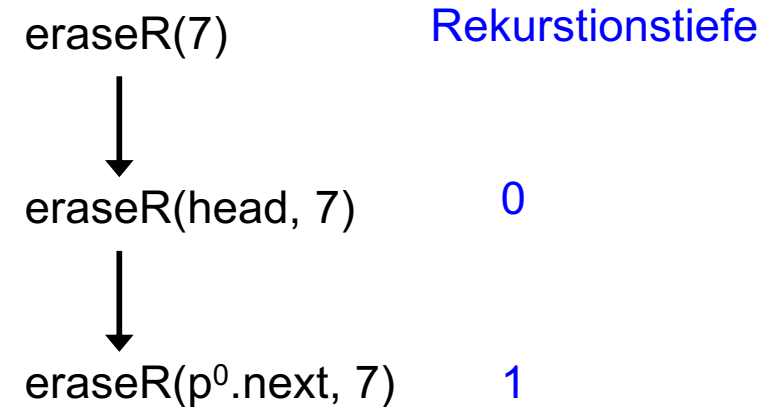
eraseR(7)      Rekursionstiefe  
↓  
eraseR(head, 7)      0



# Animation der rekursiven Methode eraseR

```
public void eraseR(int x) {  
    head = eraseR(head, x);  
}  
  
private Node eraseR(Node p, int x) {  
    if (p == null)  
        return null;  
    else if (p.data == x)  
        return p.next;  
    else {  
        p.next = eraseR(p.next, x);  
        return p;  
    }  
}
```

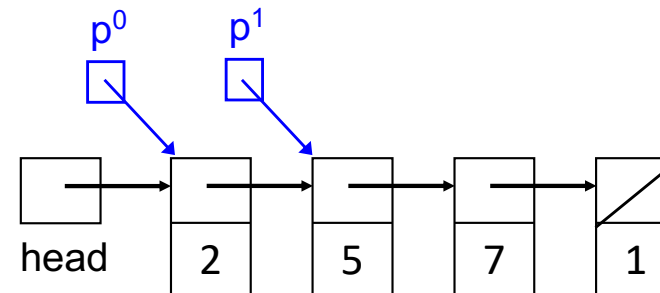
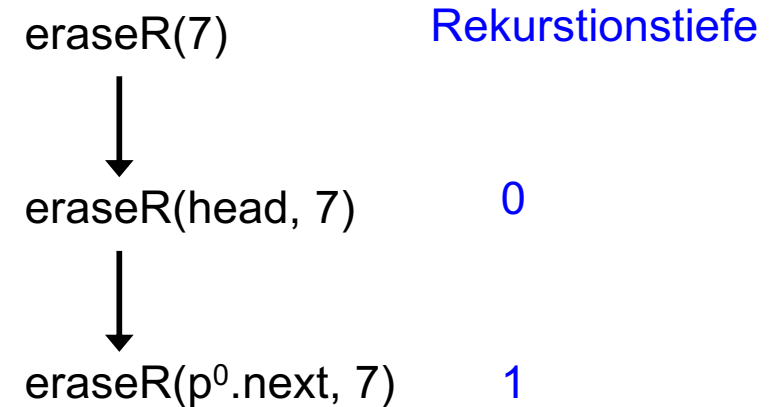
$p^i$  = Parameter  $p$  in der Rekursionstiefe  $i$



# Animation der rekursiven Methode eraseR

```
public void eraseR(int x) {  
    head = eraseR(head, x);  
}  
  
private Node eraseR(Node p, int x) {  
    if (p == null)  
        return null;  
    else if (p.data == x)  
        return p.next;  
    else {  
        p.next = eraseR(p.next, x);  
        return p;  
    }  
}
```

$p^i$  = Parameter  $p$  in der Rekursionstiefe  $i$

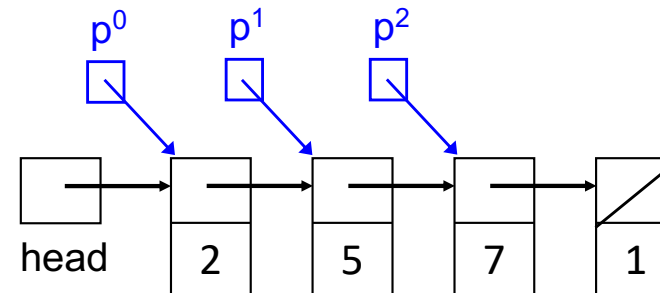
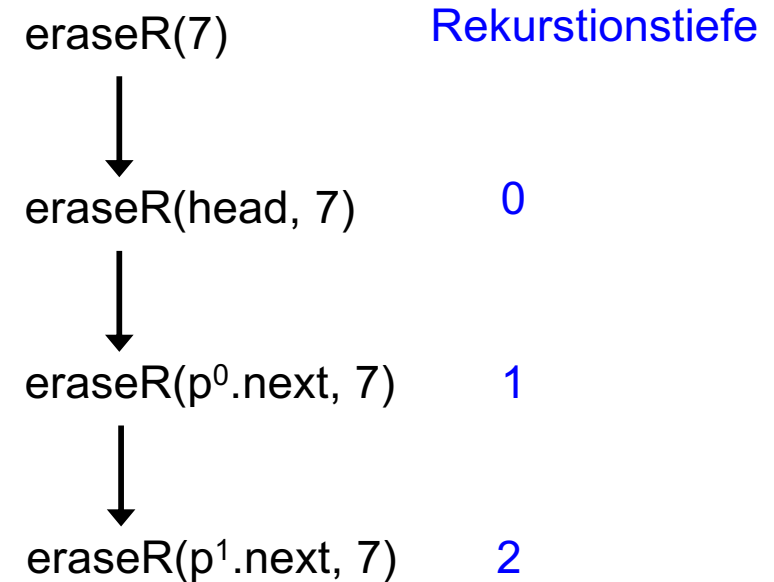


# Animation der rekursiven Methode eraseR

```
public void eraseR(int x) {
    head = eraseR(head, x);
}

private Node eraseR(Node p, int x) {
    if (p == null)
        return null;
    else if (p.data == x)
        return p.next;
    else {
        p.next = eraseR(p.next, x);
        return p;
    }
}
```

$p^i$  = Parameter  $p$  in der Rekursionstiefe  $i$

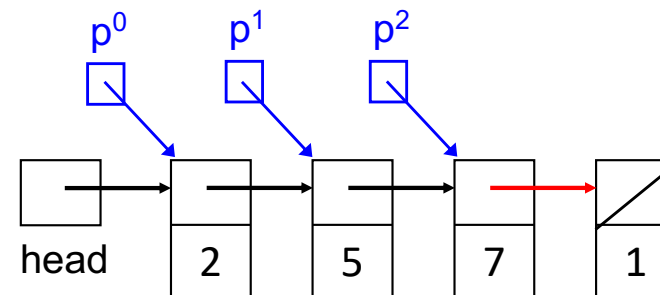
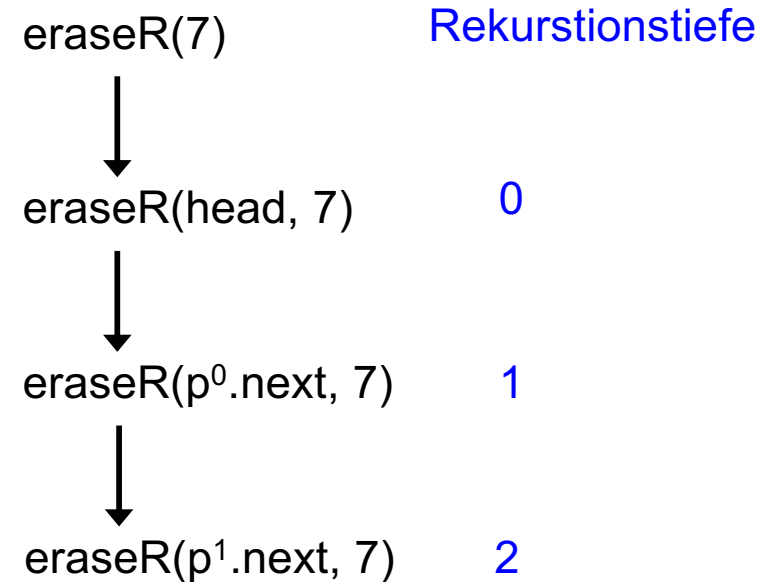


# Animation der rekursiven Methode eraseR

```
public void eraseR(int x) {
    head = eraseR(head, x);
}

private Node eraseR(Node p, int x) {
    if (p == null)
        return null;
    else if (p.data == x)
        return p.next;
    else {
        p.next = eraseR(p.next, x);
        return p;
    }
}
```

$p^i$  = Parameter  $p$  in der Rekursionstiefe  $i$

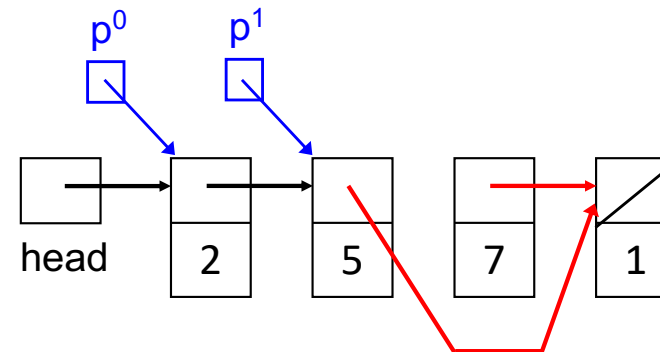
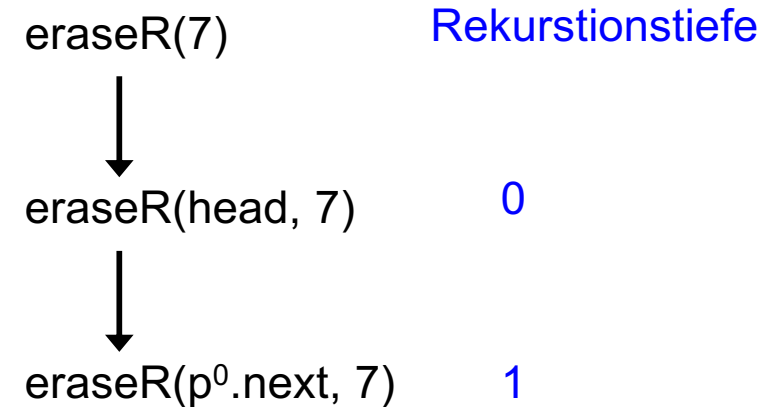


# Animation der rekursiven Methode eraseR

```
public void eraseR(int x) {
    head = eraseR(head, x);
}

private Node eraseR(Node p, int x) {
    if (p == null)
        return null;
    else if (p.data == x)
        return p.next;
    else {
        p.next = eraseR(p.next, x);
        return p;
    }
}
```

$p^i$  = Parameter  $p$  in der Rekursionstiefe  $i$



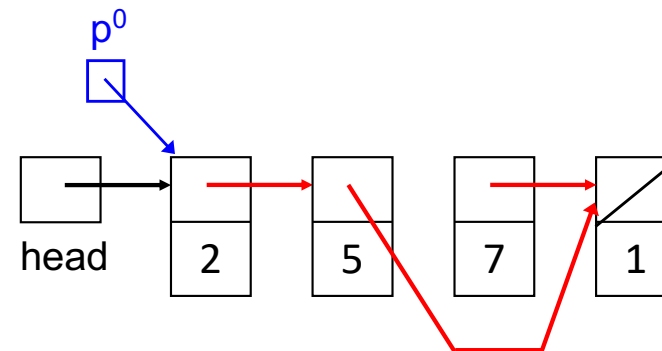
# Animation der rekursiven Methode eraseR

```
public void eraseR(int x) {
    head = eraseR(head, x);
}

private Node eraseR(Node p, int x) {
    if (p == null)
        return null;
    else if (p.data == x)
        return p.next;
    else {
        p.next = eraseR(p.next, x);
        return p;
    }
}
```

$p^i$  = Parameter  $p$  in der Rekursionstiefe  $i$

eraseR(7)      Rekursionstiefe  
↓  
eraseR(head, 7)      0

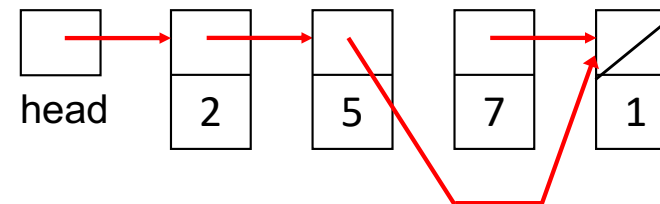




# Animation der rekursiven Methode eraseR

```
public void eraseR(int x) {  
    head = eraseR(head, x);  
}  
  
private Node eraseR(Node p, int x) {  
    if (p == null)  
        return null;  
    else if (p.data == x)  
        return p.next;  
    else {  
        p.next = eraseR(p.next, x);  
        return p;  
    }  
}
```

eraseR(7)



# Kapitel 7: Rekursion

- Grundbegriffe
- Beispiele
  - Türme von Hanoi
  - Größter gemeinsamer Teiler
  - Graphische Darstellung von Bäumen
  - Linear verkettete Listen
- Teile-und-Herrsche-Verfahren
  - Potenzfunktion
  - Binäre Suche
- Endrekursion
- Rekursion und Keller

# Teile-und-Herrsche-Verfahren

- Effiziente Lösungen bei zahlreichen Problemstellungen
- Beispiele:
  - Sortieren eines Felds mit  $n$  Zahlen
  - Suche Element  $e$  in einem sortierten Feld  $x$  mit  $n$  Elementen
  - geometrische Algorithmen

```
Funktion f(x, n) {  
    if (n == 1) {  
        // Basisfall:  
        löse Problem direkt und liefere Lösung zurück;  
    } else {  
        // Teileschritt:  
        // teile x in zwei Teilprobleme x1 und x2 (bzw. nur ein Teilproblem x1) der Größe n/2  
        // und löse sie rekursiv:  
        lös1 = f(x1, n/2);  
        lös2 = f(x2, n/2);  
        // Herrscheschritt:  
        // Setze Teillösungen lös1 und lös2 und zu einer Lösung für x zusammen  
        // und liefere sie zurück;  
    }  
}
```

Schematische Darstellung einer rekursiven Teile-und-Herrsche-Funktion  $f$ .  
 $f$  bearbeitet die Eingabe  $x$  der Größe  $n$ .

# Potenzfunktion (1)

## Aufgabenstellung

Gesucht ist eine Teile-und-Herrsche-Funktion

$$\text{power}(x,n) = x^n, n \in \mathbb{N}$$

## Idee für Teile- und Herrsche-Schritt:

$$x^n = x^{n/2} * x^{n/2}, \quad \text{falls } n \text{ gerade und } n \geq 2$$

$$x^n = x * x^{n/2} * x^{n/2}, \quad \text{falls } n \text{ ungerade und } n \geq 3$$

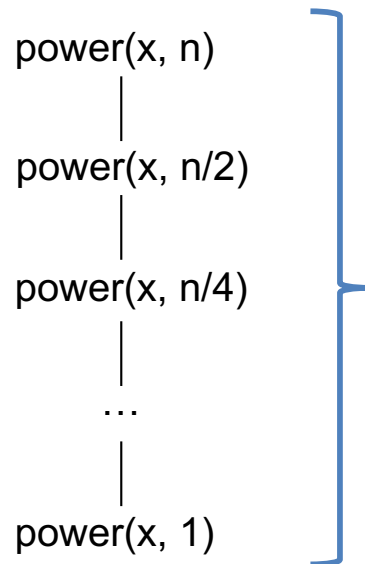
Bei  $n/2$  wird ganzzahlige Division vorausgesetzt.

## Rekursive Teile-und-Herrsche-Funktion

```
static double power(double x, int n) {
    if (n == 1)
        return x;
    else {
        double p = power(x, n/2);
        if (n%2 == 0) // n gerade
            return p*p;
        else
            return x*p*p;
    }
}
```

# Potenzfunktion (2)

## Aufrufstruktur und maximale Rekursionstiefe



Maximale Rekursionstiefe:

$$R(n) = \lfloor \log_2 n \rfloor$$

$\lfloor x \rfloor = x$  abgerundet

### Beispiel:

- Aufruf von  $\text{power}(x, 1000)$  führt zu einer maximalen Rekursionstiefe von:

$$R(1000) = \lfloor \log_2 1000 \rfloor = 9.$$

# Einschub: Potenzfunktion für große ganze Zahlen mit BigInteger

$\text{modPower}(x, y, m) = x^y \bmod m$ , für  $x, y, m \in \mathbb{N}$

Die Potenz  $x^y$  wird modular berechnet, da sonst das Ergebnis absurd groß wäre!

```
BigInteger x = new BigInteger("1273791307949826"); // x ≈ 1015
BigInteger y = new BigInteger("1340958340958234"); // y ≈ 1015
BigInteger m = new BigInteger("1000");
System.out.println(modPower(x, y, m); // 176
```

```
static BigInteger modPower(BigInteger x, BigInteger y, BigInteger m) {
    if (y.equals(BigInteger.ZERO))
        return BigInteger.ONE;
    else {
        BigInteger[] divRem = y.divideAndRemainder(BigInteger.TWO);
        BigInteger div = divRem[0]; // div = y/2
        BigInteger rem = divRem[1]; // rem = y mod 2
        BigInteger p = modPow(x, div, m);
        if (rem.equals(BigInteger.ZERO))
            return p.multiply(p).mod(m);
        else
            return x.multiply(p).multiply(p).mod(m);
    }
}
```

# Aufgabe 7.3

---

- **Wieviel Zeit** ist notwendig, um  $x^y \bmod m$  auszurechnen.
  - Dabei ist  $y \approx 10^{100}$  (100-stellige Zahl!).
  - Wir setzen einen Rechner voraus, der  $10^9$  Multiplikation/sec leistet.
1. Wieviel Zeit wäre ungefähr notwendig, um  $x^y \bmod m$  **iterativ** zu berechnen.

```
p = 1;
for (i = 1; i <= y; i++)
    p = (p*x) % m;
```

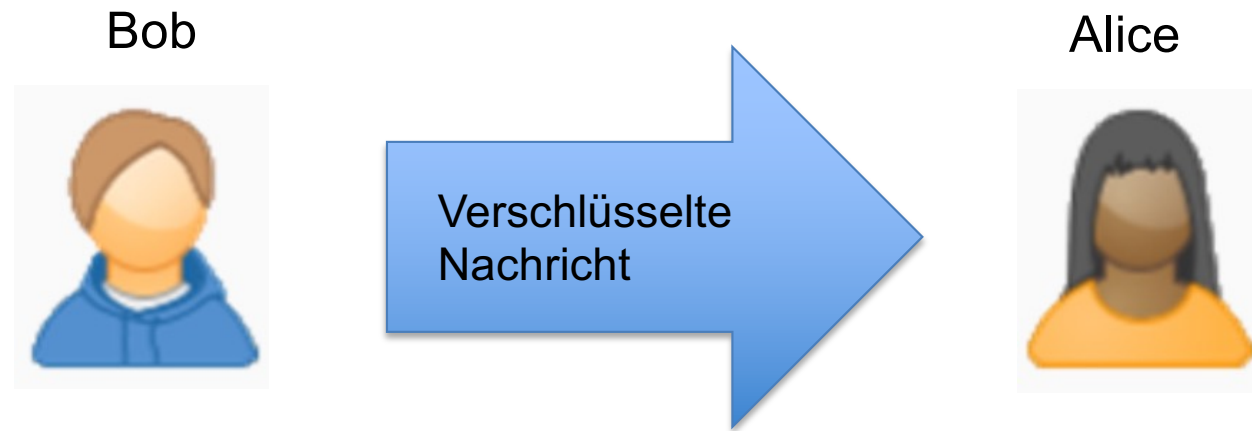
Pseudocode!

In Java müssten alle Variablen als BigInteger definiert und entsprechende Methoden benutzt werden!

2. Wieviel Zeit würde ungefähr die **Teile-und-Herrsche-Funktion** `modPower(BigInteger x, BigInteger y, BigInteger m)` benötigen?

# Einschub: Kryptographie-Verfahren RSA

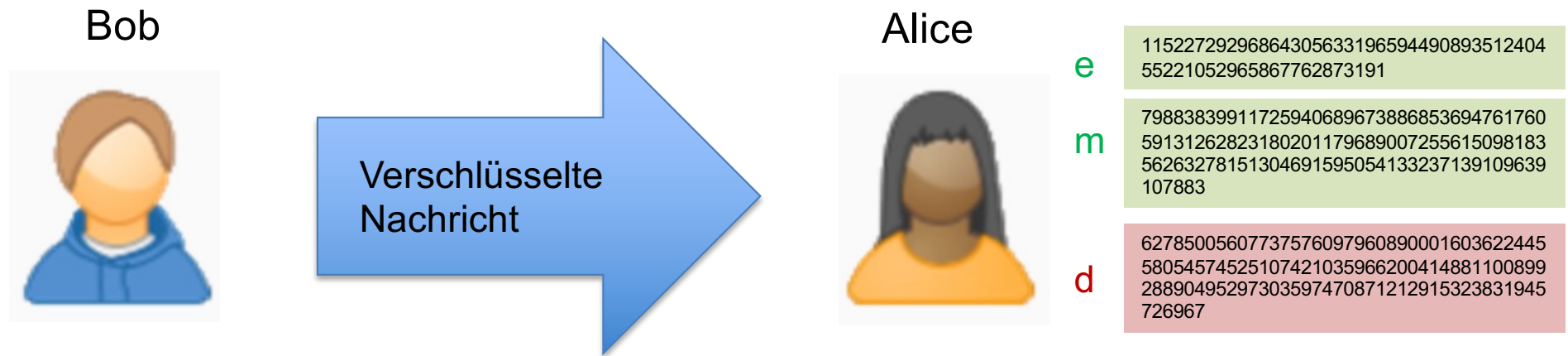
---





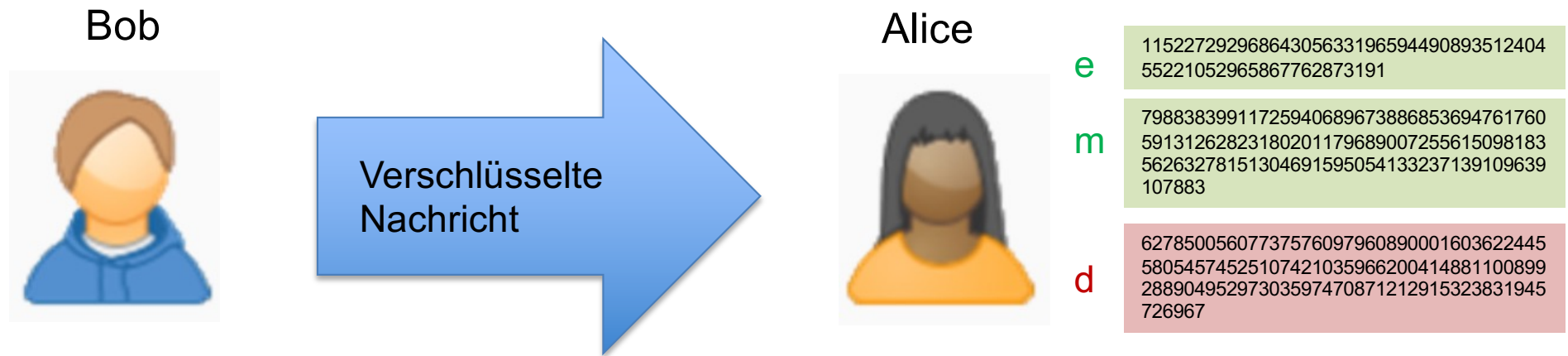
# Einschub: Kryptographie-Verfahren RSA

1. Erzeuge für Alice 3 große (am besten 100-stellige) Primzahlen  $p$ ,  $q$  und  $e < p \cdot q$ .  
Öffentliche Schlüssel sind  $e$  und  $m = p \cdot q$ .  
Berechne privaten Schlüssel  $d$  als Inverses zu  $e$ :  $e \cdot d = 1 \pmod{(p-1) \cdot (q-1)}$ .



# Einschub: Kryptographie-Verfahren RSA

1. Erzeuge für Alice 3 große (am besten 100-stellige) Primzahlen  $p$ ,  $q$  und  $e < p \cdot q$ .  
Öffentliche Schlüssel sind  $e$  und  $m = p \cdot q$ .  
Berechne privaten Schlüssel  $d$  als Inverses zu  $e$ :  $e \cdot d = 1 \pmod{(p-1) \cdot (q-1)}$ .



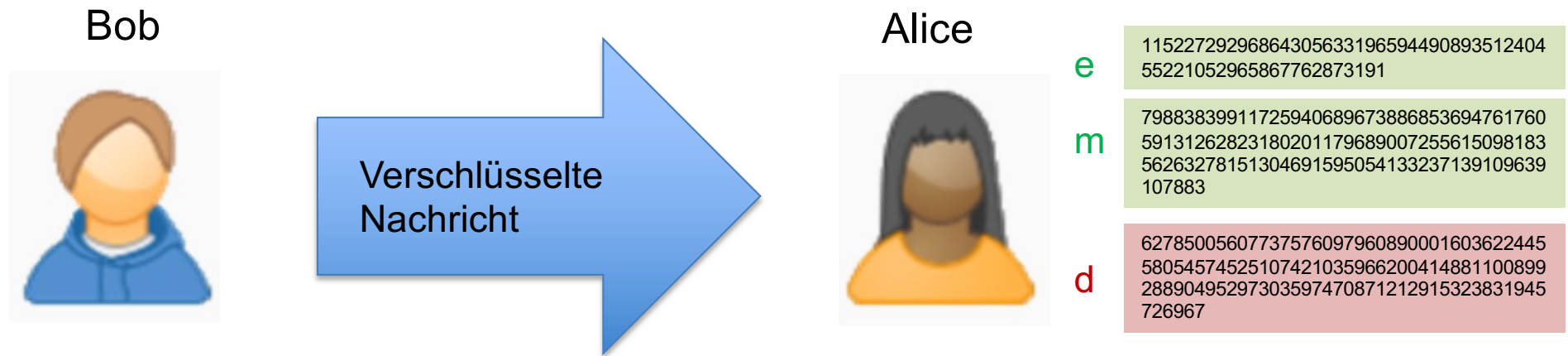
2. Bob schreibt eine Nachricht  $mes$  (String als Zahl zur Basis 256), verschlüsselt sie mit den öffentlichen Schlüsseln von Alice und schickt die verschlüsselte Nachricht an Alice:

```
mes = "Dies ist eine geheime Nachricht!";
mesEncrypted = modPower(mes, e, m);
```

```
551308618639052427531347539635316602838768255
046478614489658962245102549115898517265350673
951008654246711279058627520428
```

# Einschub: Kryptographie-Verfahren RSA

1. Erzeuge für Alice 3 große (am besten 100-stellige) Primzahlen  $p$ ,  $q$  und  $e < p * q$ .  
Öffentliche Schlüssel sind  $e$  und  $m = p * q$ .  
Berechne privaten Schlüssel  $d$  als Inverses zu  $e$ :  $e * d = 1 \text{ mod } (p-1) * (q-1)$ .



2. Bob schreibt eine Nachricht  $mes$  (String als Zahl zur Basis 256), verschlüsselt sie mit den öffentlichen Schlüssel von Alice und schickt die verschlüsselte Nachricht an Alice:

```
mes = "Dies ist eine geheime Nachricht!";
mesEncrypted = modPower(mes, e, m);
```

```
551308618639052427531347539635316602838768255
046478614489658962245102549115898517265350673
951008654246711279058627520428
```

3. Alice entschlüsselt die Nachricht mit ihrem privaten Schlüssel:

```
mesDecrypted = modPower(mesEncrypted, d, m);
println(mesDecrypted);
```

```
Dies ist eine geheime Nachricht!
```

# Kapitel 7: Rekursion

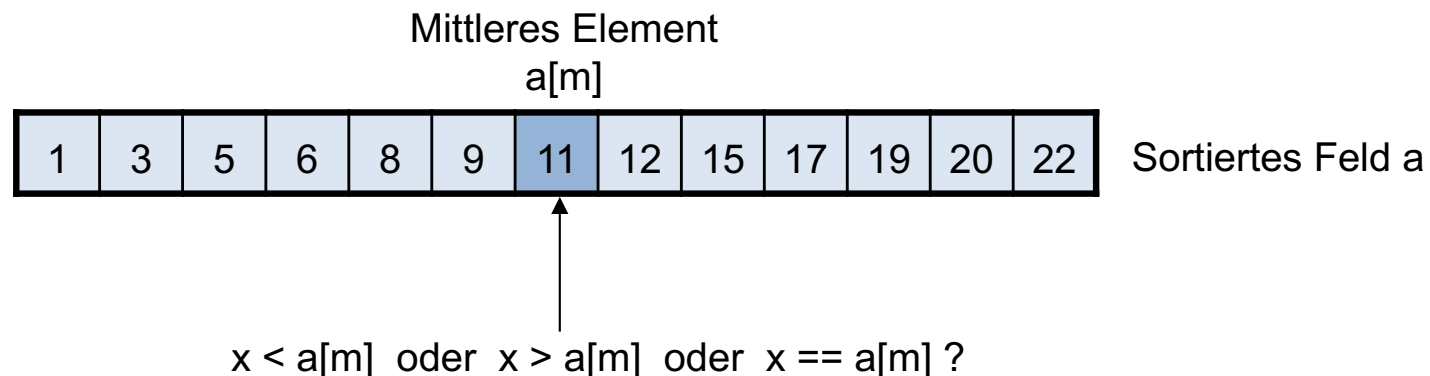
- Grundbegriffe
- Beispiele
  - Türme von Hanoi
  - Größter gemeinsamer Teiler
  - Graphische Darstellung von Bäumen
  - Linear verkettete Listen
- Teile-und-Herrsche-Verfahren
  - Potenzfunktion
  - Binäre Suche
- Endrekursion
- Rekursion und Keller

# Binäre Suche (1)

## Aufgabenstellung

- Suche  $x$  in einem **sortierten und lückenlos gefülltem Feld**  $a$ .
- Falls  $x$  gefunden wird, dann soll der Index zurückgeliefert werden und sonst  $-1$  (nicht gefunden).

## Idee für Teile-und-Herrsche-Schritt:



Falls  $x == a[m]$ , dann gefunden.

Falls  $x < a[m]$ , dann suche in linker Hälfte weiter.

Falls  $x > a[m]$ , dann suche in rechter Hälfte weiter

# Binäre Suche (2)

Beispiel: suche  $x = 8$

li=0						m=6						re=12
1	3	5	6	8	9	11	12	15	17	19	20	22

li=0		m=2				re=5						
1	3	5	6	8	9	11	12	15	17	19	20	22

Suche in linker Hälfte

			li=3	m=4	re=5							
1	3	5	6	8	9	11	12	15	17	19	20	22

Suche in rechter Hälfte;  
 $x$  wird gefunden!



Zu durchsuchender Bereich geht von  $a[li]$  bis  $a[re]$



Mittleres Element  $m = (li + re)/2$

# Binäre Suche als Java-Funktion

Das Feld a muss aufsteigend sortiert sein!

```
public static int binSuche(int[] a, int x) {  
    return binSuche(a, 0, a.length-1, x);  
}
```

```
private static int binSuche(int[] a, int li, int re, int x) {
```

```
    if (re < li)  
        return -1;
```

```
    else {
```

```
        int m = (li + re)/2;
```

```
        if (x < a[m])
```

```
            return binSuche(a, li, m-1, x);
```

```
        else if (x > a[m])
```

```
            return binSuche(a, m+1, re, x);
```

```
        else // x == a[m]
```

```
            return m;
```

```
    }
```

```
}
```

binSuche durchsucht a[li], a[li+1], ..., a[re] nach x und liefert i zurück, falls a[i] == x, sonst -1.

Basisfall: leeres Teilfeld

# Kapitel 7: Rekursion

- Grundbegriffe
- Beispiele
  - Türme von Hanoi
  - Größter gemeinsamer Teiler
  - Graphische Darstellung von Bäumen
  - Linear verkettete Listen
- Teile-und-Herrsche-Verfahren
  - Potenzfunktion
  - Binäre Suche
- Endrekursion
- Rekursion und Keller



# Endrekursion

---

## Definition

Ein **rekursiver Aufruf** heißt **endrekursiv**, falls unmittelbar nach dem Aufruf die Funktion verlassen wird.

(Endrekursion auf engl.: **tail recursion**)

## Beispiel:

```
static void print(Node p)
{
    if (p != null)
    {
        System.out.println(p.data);
        print(p.next);
    }
}
```

Aufruf ist endrekursiv.

## Aufgabe 7.4

Untersuchen Sie einige der bisher besprochenen rekursiven Funktionen auf Endrekursion.

# Eliminierung der Endrekursion (1)

---

- Ein endrekursiver Aufruf verhält sich wie eine Schleife und kann daher durch eine Schleife ersetzt werden.
- Man beachte, dass die iterative Funktion (d.h. Funktion ohne Rekursion) ressourcensparender ist. Warum?

```
static void print(Node p)
{
    if (p != null)
    {
        System.out.println(p.data);
        print(p.next);
    }
}
```

```
static void print(Node p)
{
    while (p != null)
    {
        System.out.println(p.data);
        p = p.next;
    }
}
```



# Eliminierung der Endrekursion (2)

## Allgemeines Schema

```
RT fun(T x) {  
    if (Basisfall)  
        return r;  
    else {  
        A  
        return fun(a);  
    }  
}
```



```
RT fun(T x) {  
    while (!Basisfall)  
        A  
        x = a;  
    }  
    return r;  
}
```

- RT steht für einen beliebigen Rückgabewerttyp. Der Rückgabewerttyp kann auch void sein.
- T steht für einen beliebigen Parametertyp. Im allgemeinen kann die Funktion fun auch mehrere Parameter haben.
- A steht für einen beliebigen Anweisungsblock.

# Aufgabe

---

## Aufgabe 7.5

Beseitigen Sie die Endrekursion in der binären Suche.

```
private static int binSuche(int[] a, int li, int re, int x) {
    if (re < li)
        return -1;
    else {
        int m = (li + re)/2;
        if (x < a[m])
            return binSuche(a, li, m-1, x);
        else if (x > a[m])
            return binSuche(a, m+1, re, x);
        else // x == a[m]
            return m;
    }
}
```

# Kapitel 7: Rekursion

- Grundbegriffe
- Beispiele
  - Türme von Hanoi
  - Größter gemeinsamer Teiler
  - Graphische Darstellung von Bäumen
  - Linear verkettete Listen
- Teile-und-Herrsche-Verfahren
  - Potenzfunktion
  - Binäre Suche
- Endrekursion
- Rekursion und Keller

# Keller und Rekursion (1)

---

- Endrekursive Aufrufe lassen sich einfach (d.h. schematisch) durch eine Schleife ersetzen.
- Nicht-endrekursive Aufrufe lassen sich prinzipiell mit Hilfe eines Kellers beseitigen. Manchmal kann Rekursion auch ohne Hilfe eines Kellers beseitigt werden
- Beseitigung von nicht-endrekursiven Funktionen mit Hilfe eines Kellers ist in der Regel nicht ratsam, soll aber trotzdem am Beispiel der Türme von Hanoi gezeigt werden.

```
static void bewegeTurm(int n, int s, int z, int h)
{
    if (n == 1)
        System.out.println("Bewege Scheibe von " + s + " nach " + z);
    else {
        bewegeTurm(n-1,s,h,z);
        System.out.println("Bewege Scheibe von " + s + " nach " + z);
        bewegeTurm(n-1,h,z,s);
    }
}
```

# Keller und Rekursion (2)

- Idee: Keller als Aufgabenstapel.
- Speichere im Keller zu erledigende Aufgaben als Quadrupel (n, s, z, h) ab: bewege n Scheiben von s nach z mit Hilfsplatz h.
- Beachte LIFO-Organisation des Kellers: Reihenfolge beim Auskellern eines Quadrupels ist umgekehrt zum Einkellern.

```
private static void bewegeTurm(int n, int s, int z, int h) {  
  
    Deque<Integer> stack = new LinkedList<>();  
    stack.push(n); stack.push(s); stack.push(z); stack.push(h);  
  
    while (! stack.isEmpty()) {  
        h = stack.pop(); z = stack.pop(); s = stack.pop(); n = stack.pop();  
        if (n == 1)  
            System.out.println("Bewege Scheibe von " + s + " nach " + z);  
        else {  
            stack.push(n-1); stack.push(h); stack.push(z); stack.push(s);  
            stack.push(1); stack.push(s); stack.push(z); stack.push(h);  
            stack.push(n-1); stack.push(s); stack.push(h); stack.push(z);  
        }  
    }  
}
```

Initiale Aufgabe  
einkellern

Solange Keller nicht leer  
ist, hole die oberste  
Aufgabe vom Keller und  
erledige sie

Neue Aufgaben  
einkellern.

# Einschub: Türme von Hanoi ohne Rekursion und Keller

- Statt rekursive Lösungen kann es auch einfache iterative Lösungen (ohne Keller) geben!
- Buneman und Leon haben 1980 ein einfaches iteratives Verfahren ohne Keller für Türme von Hanoi vorgeschlagen.

```
p = 0;
d = (n%2 == 0) ? -1 : +1;

while (!fertig) {
  do
    p = (p+d) mod n;
  while (p ist leer);
  nehme Scheibe S von p;
  do
    p = (p+d) mod n;
  while (S kann nicht auf p abgelegt werden);
  lege S auf p;
}
```

Starte mit Platz  $p = 0$

Drehrichtung;  
 $d = +1$  bedeutet im Uhrzeigersinn

mod math. definiert.  
Beispiel:  $-1 \bmod 3 = 2$ .

Bewege  $n = 3$  Scheiben  
von Platz 0 nach 1;  
Platz 2 ist Hilfsablage.

