3.6 Rekursion

Ein Algorithmus heißt **rekursiv**, wenn er sich selbst aufruft.

Meist werden nur einzelne Module eines Gesamtalgorithmus rekursiv verwendet.

Klassisches Beispiel:	Berechnung von n! (Fakultät von n)
Definition:	n! = 1*2*3*4*...*n
oder:	1! = 1 n! = n*(n-1)!, n>1

Wir sehen: Die Berechnung von n! wird auf die Berechnung von (n-1)! zurück geführt, diese auf die Berechnung von (n-2)! usw.

Rekursive Berechnung der Fakultät (1)

Modul *Fakultät (n)*

// Berechne Fakultät *n* für alle Ganzzahlen *n* mit *n* > 0

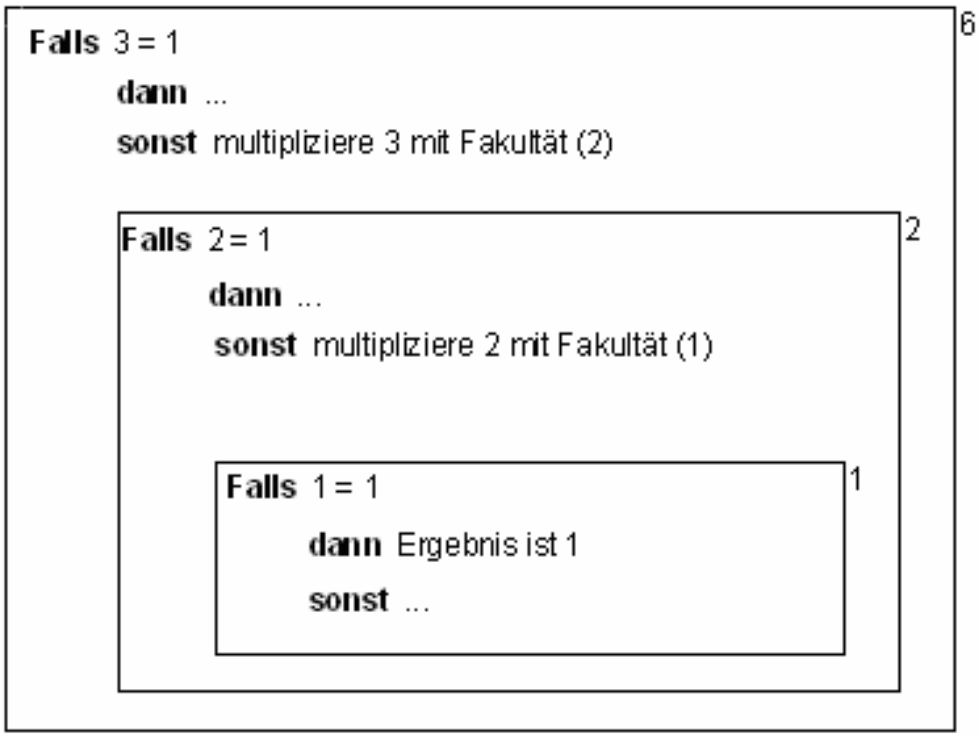
// d.h. *n*! = Fakultät (*n*) = 1 * 2 * 3 * ...* *n*; z. B. Fakultät(4) = 24

Falls *n* = 1

dann Ergebnis ist 1

sonst multipliziere *n* mit Fakultät (*n-1*)

Rekursive Berechnung der Fakultät (2)



Anmerkung: Die Berechnung von n! kann auch ohne Rekursion (iterativ) beschrieben werden. Allgemein gilt sogar: für jeden rekursiven Algorithmus gibt es einen gleichwertigen iterativen Algorithmus. Rekursive Algorithmen sind häufig knapper und leichter zu verstehen.

Vermeidung endloser Rekursion

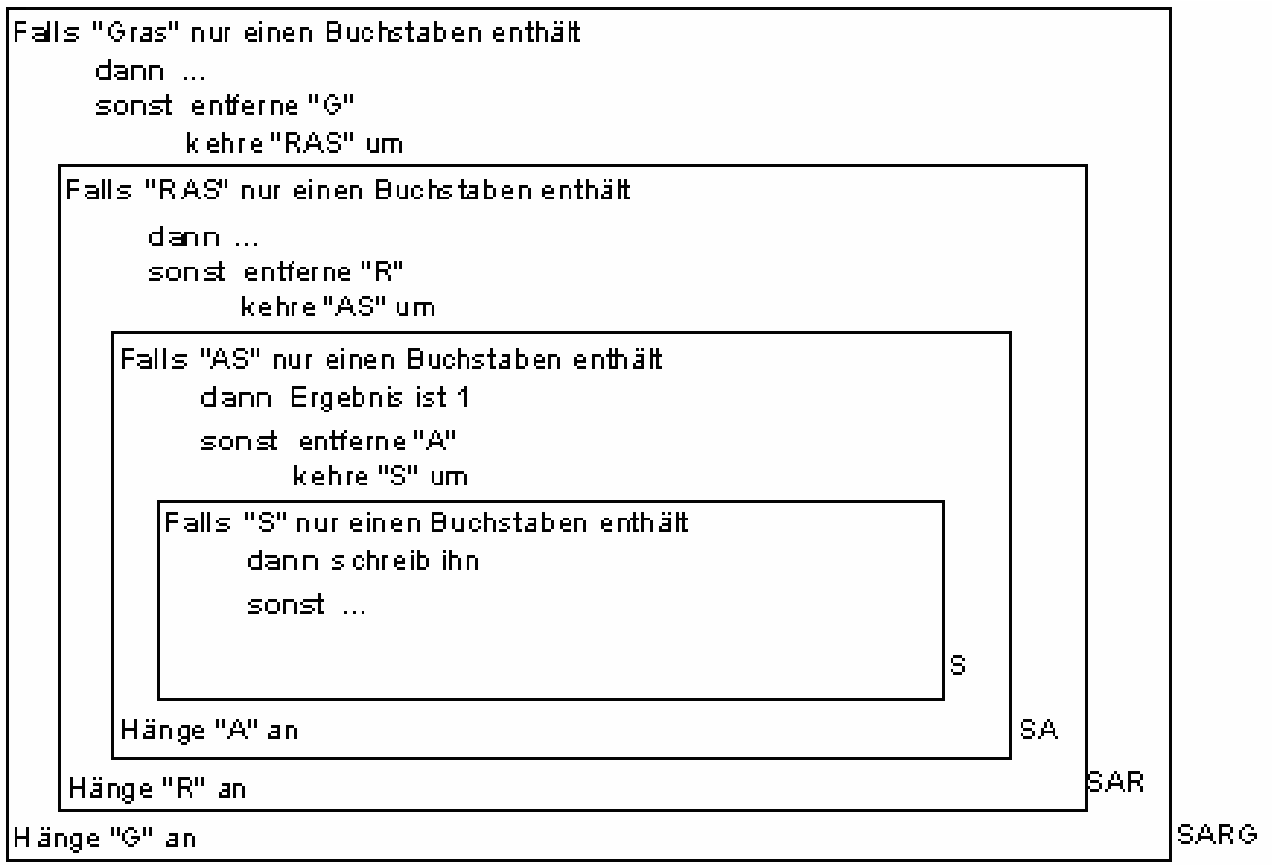
Die Definition des rekursiven Moduls muss einen Ausgang vorsehen, bei dem die Berechnung ohne erneuten Aufruf desselben Moduls erfolgt. Es muss sichergestellt sein, dass dieser Ausgang nach endlich vielen Modul-aufrufen erreicht wird.

Beispiel: Umkehrung einer Buchstabenfolge (1)

```
Entferne ersten Buchstaben

Kehre den Rest des Wortes um

Hänge den ersten Buchstaben an

Modul Umkehrung(Buchstabenfolge)

// Kehrt die Folge der Buchstaben eines gegebenen
// Strings um

Falls  Buchstabenfolge enthält nur einen Buchstaben

dann schreibe ihn nieder

sonst entferne ersten Buchstaben von Buchstabenfolge

      Umkehrung(Rest der Buchstabenfolge)

      hänge entfernten Buchstaben an
```

Beispiel: Umkehrung einer Buchstabenfolge (2)

Kehre "GRAS" um



Beispiel: Umkehrung einer Buchstabenfolge (3)

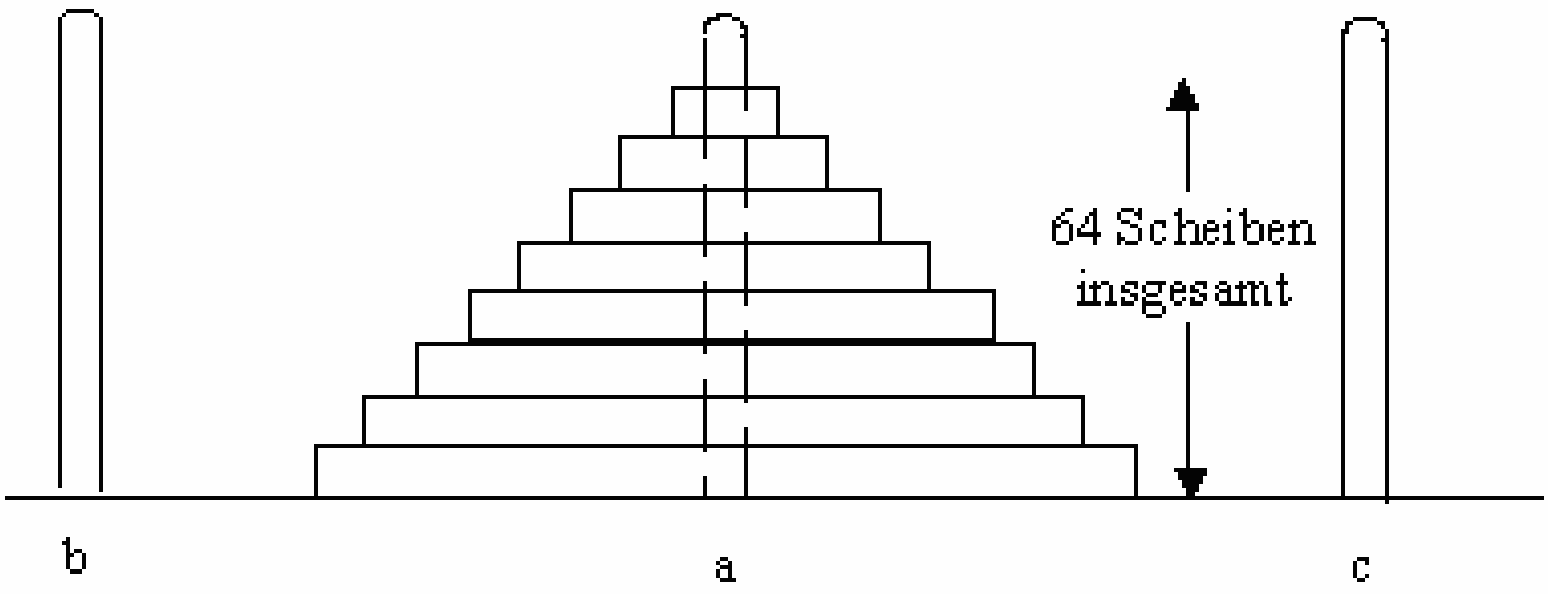
Anmerkung

Die Rückkehr aus einem rekursiven Modul erfolgt stets an die Stelle, von der aus das Modul aufgerufen wurde.

Berühmtes Beispiel: Die Türme von Hanoi (1)

Das Problem

Bringe 64 Scheiben von a (Quelle) nach b (Senke) unter Zuhilfenahme von c (Arbeitsbereich). Es darf niemals eine größere Scheibe über einer kleineren Scheibe liegen.



Beispiel: Die Türme von Hanoi (2)

Schrittweise Verfeinerung

1. *Übertrage die oberen 63 Scheiben von Stab a nach Stab c*

   Bewege die unterste Scheibe von Stab a nach Stab b

   Übertrage die oberen 63 Scheiben von Stab c nach Stab b

2. *Übertrage n-1 Scheiben von der Quelle zum Arbeitsbereich*

   Bewege 1 Scheibe von der Quelle zur Senke

   Übertrage n - 1 Scheiben vom Arbeitsbereich zur Senke

Beispiel: Die Türme von Hanoi (3)

Modul *Turmbewegung(n, Quelle, Senke, Arbeitsbereich)*

// Bewegt einen Turm mit n Scheiben von *Quelle* zu *Senke* und benutzt
// *Arbeitsbereich,* falls erforderlich.

Falls *n = 1*

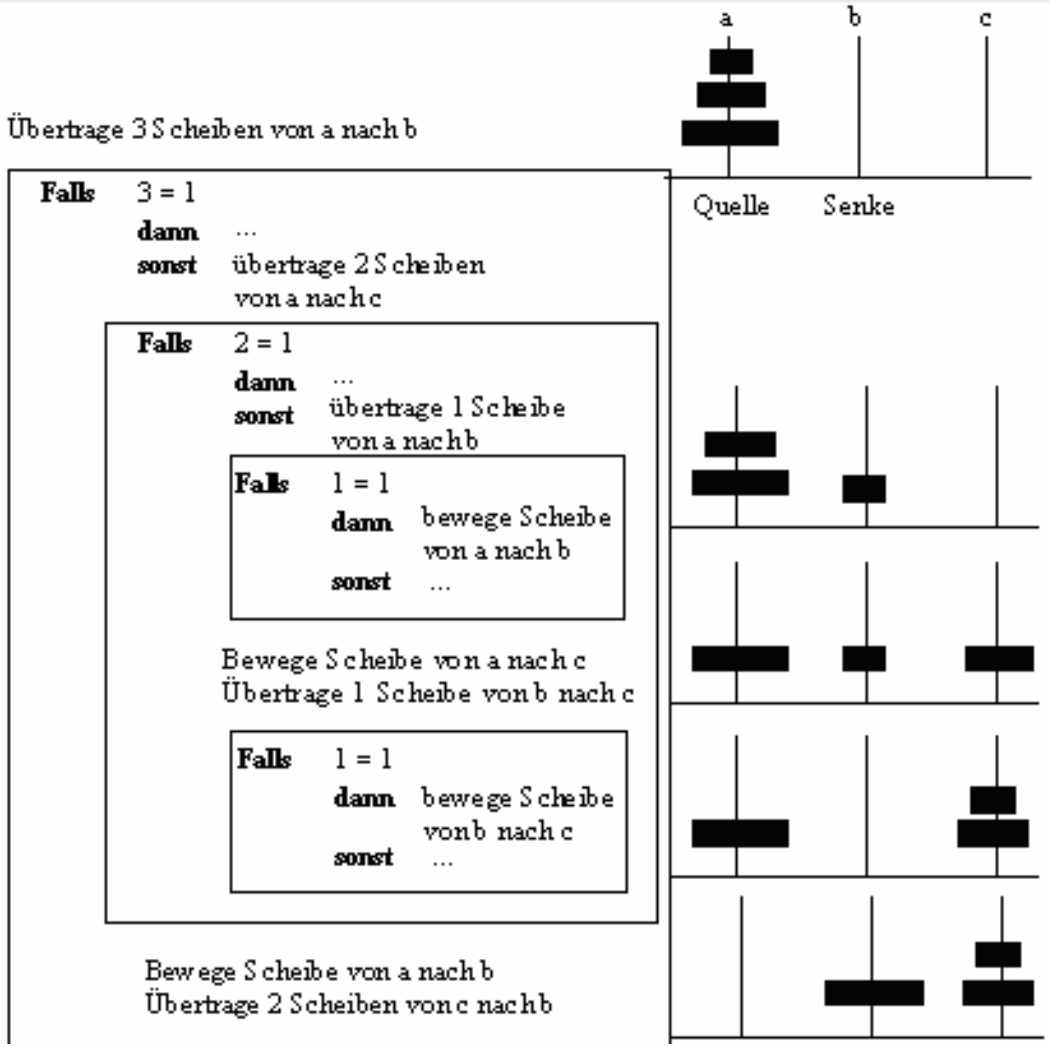
dann *bewege Scheibe von der Quelle zur Senke*

sonst *Turmbewegung(n-1, Quelle, Arbeitsbereich, Senke)*

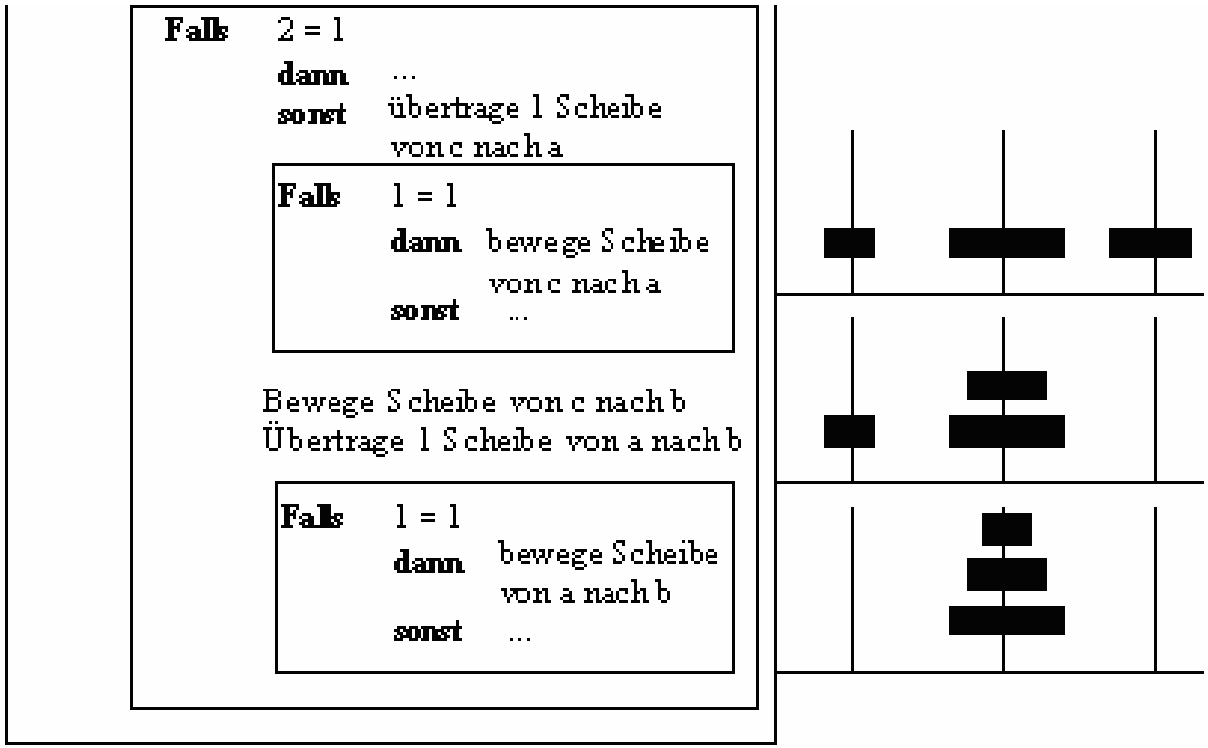
bewege 1 Scheibe von der Quelle zur Senke

Turmbewegung(n-1, Arbeitsbereich, Senke, Quelle)

Beispiel: Die Türme von Hanoi (4)



Beispiel: Die Türme von Hanoi (5)



Anmerkung

Die Lösung des Problems "Türme von Hanoi" durch einen iterativen Algorithmus ist wenig elegant und schwerer zu verstehen.

Strategie zum Entwurf rekursiver Algorithmen

1. Definiere Schritt i durch Rückgriff auf Schritt i-1, wobei Schritt i-1 ein "einfacheres" Problem löst.
2. Finde Ausstiegsbedingung zum Abbruch der Rekursion.

Beispiel: Rekursives Sortieren (1)

Sortiere erste Hälfte der Liste

Sortiere zweite Hälfte der Liste

„Mische“ beide Hälften zusammen (engl.: merge)

Modul Sortiere (Liste)

// Sortiert eine gegebene Liste von n Namen alphabetisch

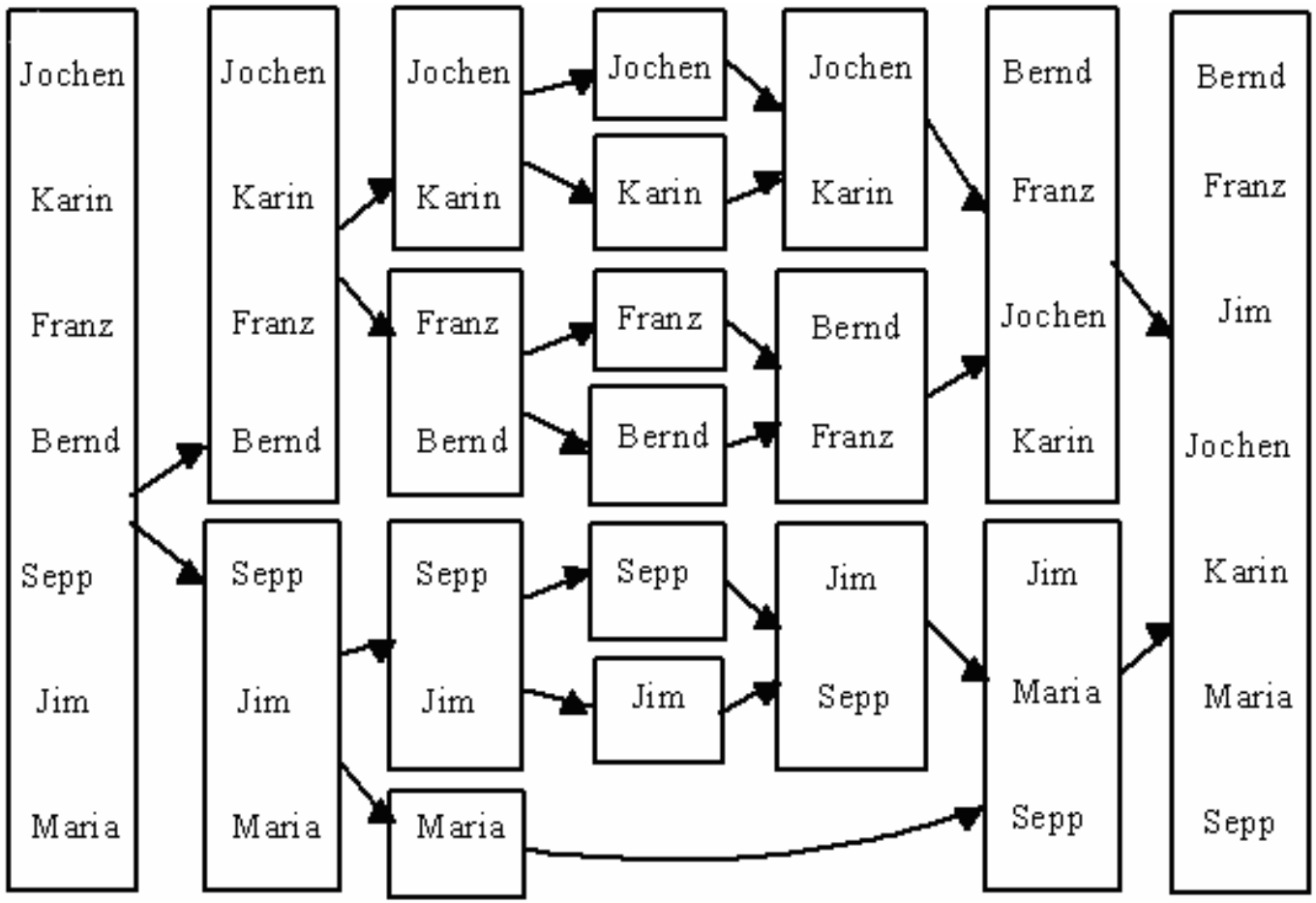
Falls *n* > 1

dann Sortiere (erste Listenhälfte)

Sortiere (zweite Listenhälfte)

Mische die zwei Hälften zusammen

Beispiel: Rekursives Sortieren (2)



Beweis von rekursiven Formeln

Mathematische Formeln, die rekursiv definiert sind, werden meist durch **Induktion** bewiesen.

Induktionsbasis:

Formel korrekt für $n = 1$, $n \in N$

Induktionsannahme:

Formel sei korrekt für ein $n \geq 1$, $n \in N$

Schluss von n auf n+1:

Berechnung des Formelwertes für n+1 unter Verwendung der Formel für n

Beispiel

Die rekursiv definierte Formel

$$s(n) = \begin{cases} 1 & , n = 1 \\ & \\ & \\ n + s(n-1), & n > 1,\ n \in \mathrm{N} \end{cases}$$

berechnet die Summe der ersten n natürlichen Zahlen.

Behauptung

$$s(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Beweis durch Induktion (1)

Induktionsbasis:

$$s(1) = 1 = \frac{1*(1+1)}{2} = 1$$

Induktionsannahme:

$$s(n) = \frac{n}{2}(n+1),\ n \in \mathrm{N}$$

Beweis durch Induktion (2)

Schluss von n auf n+1:

$$
\begin{aligned}
s(n+1) &= n+1+s(n) \\
&= n+1+\frac{n}{2}(n+1) \\
&= (1+\frac{n}{2})(n+1) \\
&= \frac{2+n}{2}(n+1) \\
&= \frac{(n+2)}{2}(n+1) \\
&= \frac{(n+1)(n+2)}{2}
\end{aligned}
$$

Komplexität

Bei rekursiver Berechnung ist die Komplexität eines Algorithmus oft wesentlich höher als bei geschlossenen Ausdrücken!

Beispiel:

$$s(n) = \begin{cases} 1 \quad , n = 1 \\ \\ n + s(n-1), n > 1, \; n \in \mathrm{N} \end{cases}$$

erfordert n - 1 Operationen.

$$\frac{n(n+1)}{2}$$ erfordert 3 Operationen für beliebig große n

Beweis der Korrektheit für die „Türme von Hanoi“ (1)

Behauptung

Der Algorithmus

Turmbewegung(*n,Quelle,Senke,Arbeitsbereich*)

ist korrekt.

Beweis

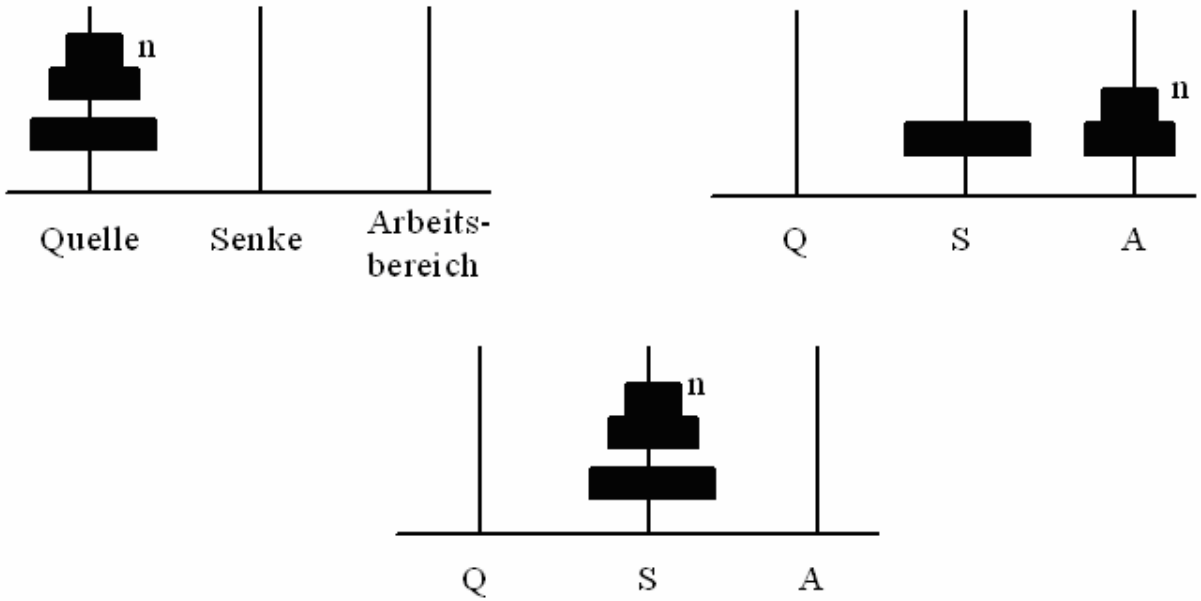
Behauptung 1: Turmbewegung tut das gewünschte

Beweis der Korrektheit für die „Türme von Hanoi“ (2)

Beweis durch Induktion über die Anzahl der Scheiben n:

Induktionsbasis:	n = 1:	klar
Induktionsannahme:	Algorithmus funktioniere für n Scheiben, $n \geq 1$	
Induktionsschluss:	- Algorithmus bewegt zunächst n Scheiben von Quelle zum Arbeitsbereich (funktioniert nach Induktionsannahme), - bewegt dann die n+1-te Scheibe zur Senke (funktioniert nach Induktionsbasis) - und bewegt schließlich die n Scheiben vom Arbeitsbereich zur Senke (funktioniert nach Induktionsannahme).	

Beweis der Korrektheit für die „Türme von Hanoi“ (3)



Behauptung 2:

Turmbewegung terminiert

Beweis:

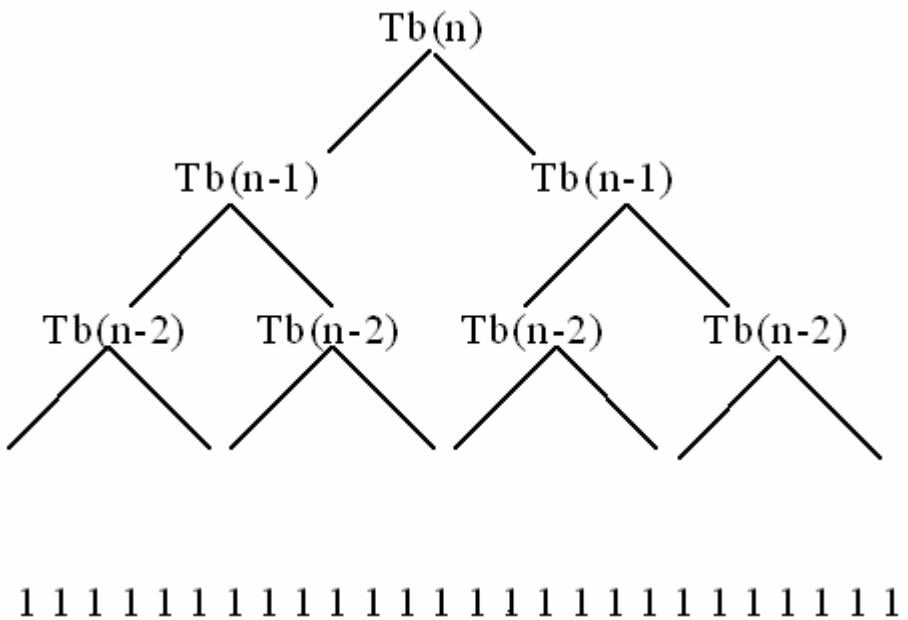
Da jeder Aufruf höchstens zwei neue Aufrufe erzeugen kann und bei jedem Aufruf weniger Scheiben bewegt werden, wird irgendwann der Algorithmus mit n = 1 aufgerufen.

Komplexität der „Türme von Hanoi“ (1)

Dazu berechne man die Anzahl *Tb* der Aufrufe von
Turmbewegung(*n, Quelle, Senke, Arbeitsbereich*)

$$Tb(n) = \begin{cases} 1 \quad , n = 1 \\ \\ 2Tb(n-1)+1, \;\; n > 1, n \in \mathrm{N} \end{cases}$$

Komplexität der „Türme von Hanoi“ (2)



$Tb(n) = 1 + 2 + 4 + ... + 2n-2 + 2n-1 = \ = 2n - 1$

3.7 Daten und Datenstrukturen

Daten

Fundamentale Objekte, die in der Rechenanlage

- erfasst
- gespeichert
- ausgegeben (angezeigt, gedruckt)
- bearbeitet
- gelöscht

werden können.

Beispiele: Zahlen, Zeichenfolgen (Strings), Dokumente, Bilder usw.

Information

Wissenszuwachs beim Menschen auf Grund von Daten. Information wird aus den Daten durch *Interpretation* gewonnen.

Daten vs. Information

Nicht eins-zu-eins abbildbar!

Beispiele

- Eine Aussage in verschiedenen Sprachen hat denselben Informationsgehalt.
- Dieselbe Aussage hat für verschiedene Personen verschiedene Bedeutung, z. B. ("es wird ein Gewitter geben").
- Dieselbe Aussage hat in verschiedenen Kontexten verschiedene Bedeutung, z. B. "ich mag Kohl".
- Verschlüsselte Daten haben für verschiedene Empfänger unterschiedlichen Informationsgehalt; nur wer den Schlüssel kennt, kann interpretieren

Datentypen

Wir kennen bereits den Unterschied zwischen **Datentypen** und **Daten-werten**.

Beispiele

Datentyp	englisch	Wert
Ganzzahl	integer	3
Gleitkomma	real	3.1415 3.7E-12
logisch	boolean	TRUE, FALSE
Zeichen	character	'A'
Zeichenfolge	string	"Egon"
Bit	bit	'0'b, '1'b
Bitfolge	bitstring	'00010100'b '000L0L00'b

Datenstrukturen

Aus den elementaren Datentypen können komplexere Datentypen konstruiert werden. Es entstehen **Datenstrukturen**.

In der Informatik wichtige Datenstrukturen sind beispielsweise:

Liste (list, file)
Eine Folge von Datenelementen gleichen Typs

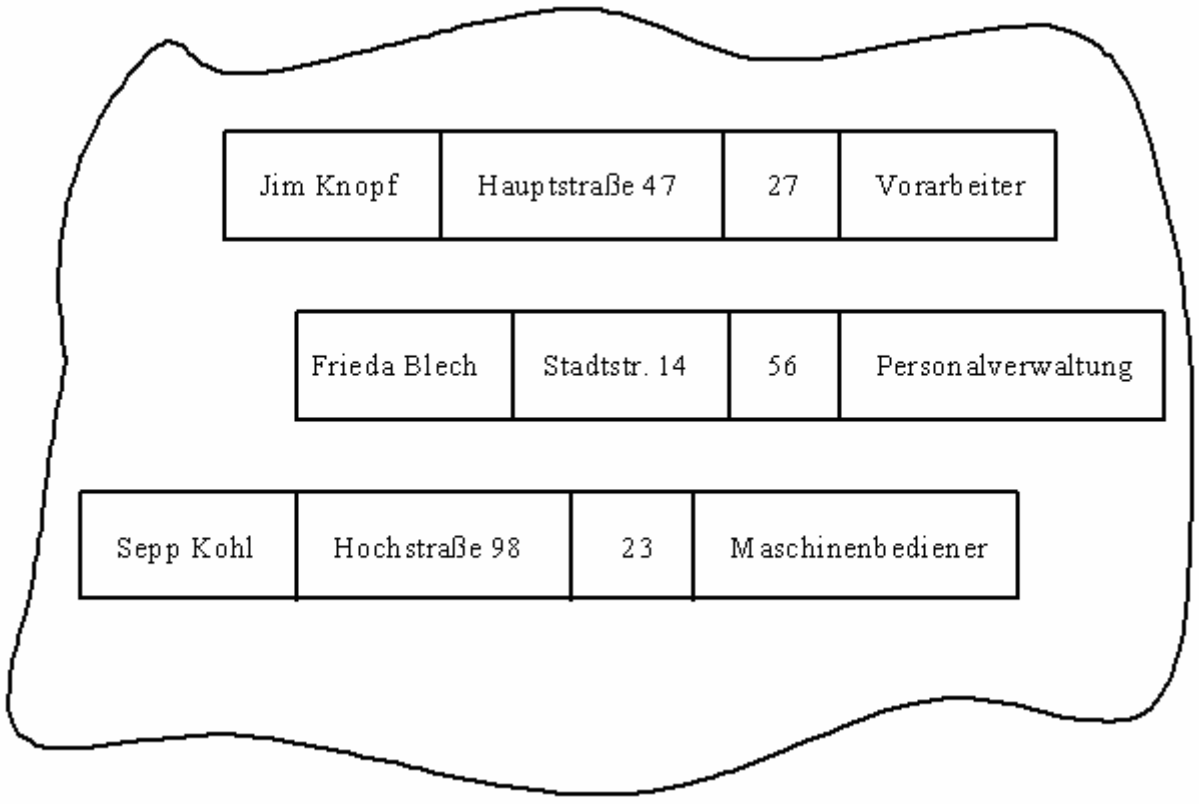
Datensatz (record)
Eine Folge von Datenelementen verschiedenen Typs

Baum (tree)
Eine hierarchische Anordnung von Datenelementen gleichen oder verschiedenen Typs

Diese Datenstrukturen können geschachtelt werden, wir können zum Beispiel Listen von Datensätzen oder Bäume von Datensätzen bilden.

Datensatz

Beispiel

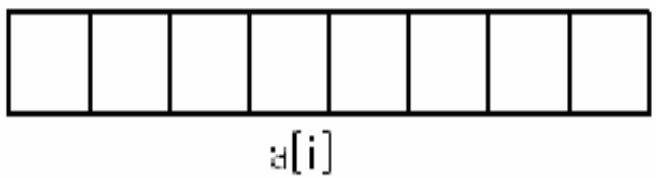


Spezialformen der Liste (1)

Drei wichtige Spezialformen der Liste sind das **Feld** (array), die **Warteschlage** (queue) und der **Stapel** oder **Kellerspeicher** (stack).

Feld (array)

Liste fester Länge, bei der jedes Datenelement durch seine Position direkt adressiert werden kann (Index).



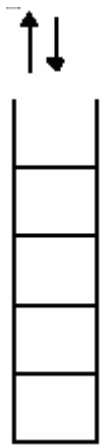
Warteschlange (queue)

Liste variabler Länge, bei der stets an *einem* Ende hinzugefügt und am *anderen* Ende entfernt wird ("first in, first out", FIFO)

Spezialformen der Liste (2)

Stapel oder Kellerspeicher (stack)

Liste variabler Länge, bei der Elemente nur an *einem* Ende hinzugefügt oder entfernt werden können ("last in, first out", LIFO)

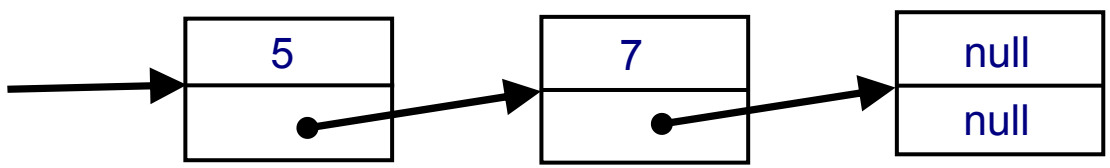


Beispiele für Listen

- Karteikasten (Liste von Datensätzen)
- Namensliste während der Sortierung im Hauptspeicher (Liste von Datensätzen)
- Seiten in einer Loseblattsammlung (Feld (array))
- mathematische Beschreibung eines Punktes im Raum
  P = (x,y,z), x, y, z ε R Wert: (3.8, 4.7, 1.9) (Feld (array))
- Lochkarten im Lochkartenleser (Warteschlange (queue))
- Prozesse, die auf die CPU warten (Warteschlange (queue))
- Tellerstapel im Restaurant (Kellerspeicher (stack))
- Güterzug auf dem Abstellgleis (Kellerspeicher (stack))

Eine verkettete Liste in Java

```
public class LinkedList {
    private Object item;
    private LinkedList next;
    public LinkedList() { }
}
```



Einfügen in die verkettete Liste

```
private LinkedList(Object i, LinkedList n) {
        item = i;
        next = n;
}
public Object insertfirst(Object elem) {
        next = new LinkedList(item, next);
        return item = elem;
}
public Object insertlast(Object elem) {
        if (item == null) {
           item = elem;
           next = new LinkedList();
        } else
           next.insertlast(elem);
        return elem;
}
```

Entfernen von Elementen aus der verketteten Liste

```
public Object removefirst() {
        if (item != null) {
            Object res = item;
            item = next.item;
            next = next.next;
            return res;
        }
        return null;
}
public Object removelast() {
        return (item == null) ?
            null : (next.item == null) ? removefirst() : next.removelast();
}
```

Suchen und Ausgeben

```
// suchen
    public Object search(Object elem) {
            return (item == null) ?
               null : (item.equals(elem)) ? item : next.search(elem);
    }

// Liste ausgeben
    public String toString() {
            return (item == null) ? "" : item.toString() + " " + next.toString();
    }
```

Algorithmen und Datenstrukturen

Erkenntnis

Häufig lassen sich Algorithmen aus einer Datenstruktur herleiten!

Beispiele:

- aus der Eingabedatenstruktur
- aus der angestrebten Ausgabedatenstruktur
- aus einer anderen als sinnvoll erkannten internen Zwischenstruktur