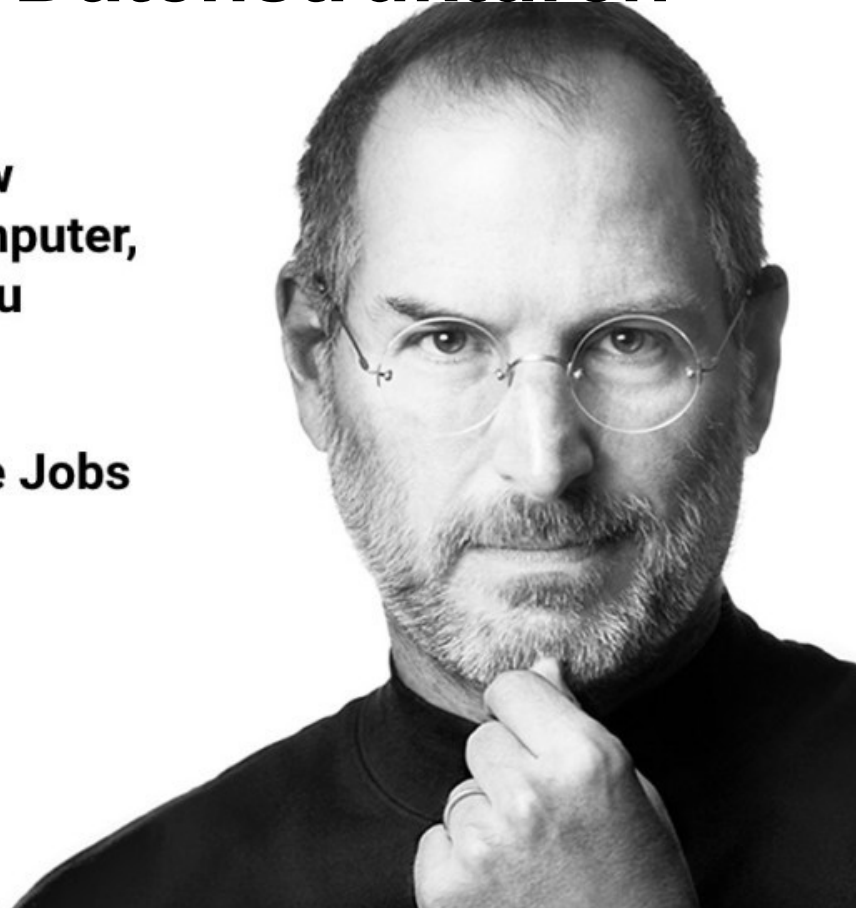


Teil 2

Algorithmen und Datenstrukturen

**"Everyone should know
how to program a computer,
because it teaches you
how to think."**

Steve Jobs



Stephan Mechler

1. Vorlesung

19.03.2024

JETZT



Algorithmen

Begriff:

- Ein Algorithmus ist ein Verfahren mit einer
 - ◆ präzisen (d.h. in einer genau festgelegten Sprache formulierten)
 - ◆ endlichen Beschreibung unter Verwendung
 - ◆ effektiver (d.h. tatsächlich ausführbarer)
 - ◆ elementarer Verarbeitungsschritte
- Zu jedem Zeitpunkt der Abarbeitung des Algorithmus benötigt dieser nur endlich viele Ressourcen

- Muhammad ibn Musa al-Chwarazmi (* ca. 783; † ca. 850)
Buch: Über das Rechnen mit indischen Ziffern (um 825)

- Erster Computeralgorithmus
(1842 von Ada Lovelace,
Notizen zu Charles Babbages Analytical Engine)



- Turingmaschinen und Algorithmusbegriff (Alan Turing 1936)

„Berechnungsvorschriften zur Lösung eines Problems heißen genau dann Algorithmus, wenn eine zu dieser Berechnungsvorschrift äquivalente Turingmaschine existiert, die für jede Eingabe, die eine Lösung besitzt, stoppt“

- Ein Algorithmus verarbeitet Daten mit Hilfe von Anweisungen
 - ◆ "Programm = Daten + Befehle"
- Daten
 - ◆ Werte: Zahlen, Zeichen, Zeichenketten, ...
 - ◆ Variablen: Benannte Behälter für Werte
- Anweisungen, Befehle
 - ◆ Zuweisung
(setze $n := 1$)
 - ◆ bedingte Anweisung
(falls $\langle \text{Bedingung} \rangle$: $\langle \text{Anweisung} \rangle$)
 - ◆ Folge von Anweisungen
($\langle \text{Anweisung } 1 \rangle$; ... ; $\langle \text{Anweisung } n \rangle$)
 - ◆ Schleifen
(solange $\langle \text{Bedingung} \rangle$: $\langle \text{Anweisung} \rangle$)

■ Aufgabe:

Berechne die Summe der Zahlen von 1 bis n

■ Algorithmus:

setze summe := 0

setze zähler := 1

solange zähler <= n:

 setze summe := summe + zähler

 erhöhe zähler um 1

gib aus: "Die Summe ist: " und summe

■ **Allgemeinheit**

Lösung einer Klasse von Problemen, nicht eines Einzelproblems, Spezialfalls

■ **Operationalität**

Einzelschritte sind wohl definiert und können auf entsprechend geeigneten Rechenanlagen ausgeführt werden

■ **Endliche Beschreibung**

Die Notation eines Algorithmus hat eine endliche Länge

■ **Funktionalität**

Ein Algorithmus reagiert auf Eingaben und produziert Ausgaben

- **Terminierung**
Algorithmus läuft für jede Eingabe nur endlich lange
- **Vollständigkeit**
Algorithmus liefert alle gewünschten Ergebnisse
- **Korrektheit**
Algorithmus liefert nur richtige Ergebnisse
- **Determinismus**
Ablauf ist für dieselbe Eingabe immer gleich
- **Determiniertheit**
Ergebnis ist festgelegt für jede Eingabe
- **Effizienz**
Algorithmus ist sparsam im Ressourcenverbrauch
- **Robustheit**
Algorithmus ist robust gegen Fehler aller Art
- **Änderbarkeit**
Algorithmus ist anpassbar an modifizierte Anforderungen

- Ein terminierender Algorithmus läuft für **jede (!)** beliebige Eingabe jeweils in endlicher Zeit ab

- **Gegenbeispiel:**

- ◆ fak(n): falls $n = 0$, liefere 1
 sonst liefere $n * \text{fak}(n-1)$

- ◆ Ergebnis von fak(2) ?

- ◆ Ergebnis von fak(-3) ?

- Ein vollständiger Algorithmus gibt alle gewünschten Ergebnisse aus

- **Gegenbeispiel:**

- ◆ Teilmengen(n):

```
setze i := 1;
```

```
solange i <  $\sqrt{n}$  :
```

```
    falls n/i ganzzahlig, gib i und n/i aus
```

```
    erhöhe i um 1
```

- ◆ Ausgabe von Teilmengen(12) ?
- ◆ Ausgabe von Teilmengen(9) ?

- Ein korrekter Algorithmus liefert nur richtige Ergebnisse

- **Gegenbeispiel:**

- ◆ Teilmengen(n):

- setze $i := 1$

- solange $i \leq \sqrt{n}$:

- falls n/i ganzzahlig, gib i und n/i aus

- erhöhe i um 1

- ◆ Ausgabe von Teilmengen(12) ?

- ◆ Ausgabe von Teilmengen(9) ?

- Ein deterministischer Algorithmus läuft für ein- und dieselbe Eingabe immer auf dieselbe Art und Weise ab
- Ein nicht-deterministischer Algorithmus kann für ein- und dieselbe Eingabe unterschiedlich ablaufen
- **Beispiel:**
 - ◆ **Absolutbetrag(x):**
wähle einen Fall:
falls $(x \leq 0)$, liefere $-x$
falls $(x > 0)$, liefere x
 - ◆ Ergebnis von Absolutbetrag(3) ?
 - ◆ Ergebnis von Absolutbetrag(-5) ?
 - ◆ Ergebnis von Absolutbetrag(0) ?

- Ein determinierter Algorithmus liefert für ein- und dieselbe Eingabe immer dasselbe Ergebnis
- **Gegenbeispiel:**
 - ◆ `wecker(uhrzeit):`
 - `wenn Wochentag="So" und uhrzeit>9`
 - `liefere "Aufstehen!"`
 - `wenn Wochentag="Sa" und uhrzeit>8`
 - `liefere "Aufstehen!"`
 - `sonst wenn uhrzeit>7 liefere "Aufstehen!"`
 - ◆ Ergebnis von `wecker(8)` heute ?
 - ◆ Ergebnis von `wecker(8)` am Sonntag?
- Wichtiges nicht-determiniertes Beispiel:
Ein Zufallszahlengenerator

- Für eine gegebene Eingabe sollen die benötigten Ressourcen möglichst gering (oder sogar minimal) sein
 - ◆ Betrachtung von Speicherplatz und Rechenzeit (Anzahl der Einzelschritte)
 - ◆ Ggf. auch Analyse von Plattenzugriffen (I/Os) oder Netzzugriffen
- **Beispiel:**
 - ◆ Die iterative Berechnung der Summe von 1 bis n benötigt n Schritte
 - ◆ Durch Verwendung der Summenformel $n * (n+1)/2$ erhalten wir einen effizienteren Algorithmus
- Unterschied Effizienz vs. Effektivität
 - ◆ Effektivität ist "Grad der Zielerreichung" (Vollständigkeit, Korrektheit)
 - ◆ Effizienz ist die "Wirtschaftlichkeit der Zielerreichung"

- Der Ablauf soll auch für fehlerhafte Eingaben der Daten wohl definiert sein
- **Beispiele:**
 - ◆ Bedienfehler
 - Leere Eingaben in (grafische) Eingabefelder
 - Eingabe von Strings in Eingabefelder für Zahlen
 - ◆ Systemfehler
 - Zugriff auf Ressourcen (Dateien, Netz etc.) nicht möglich

■ Einfache Anpassung an veränderte Aufgabenstellungen

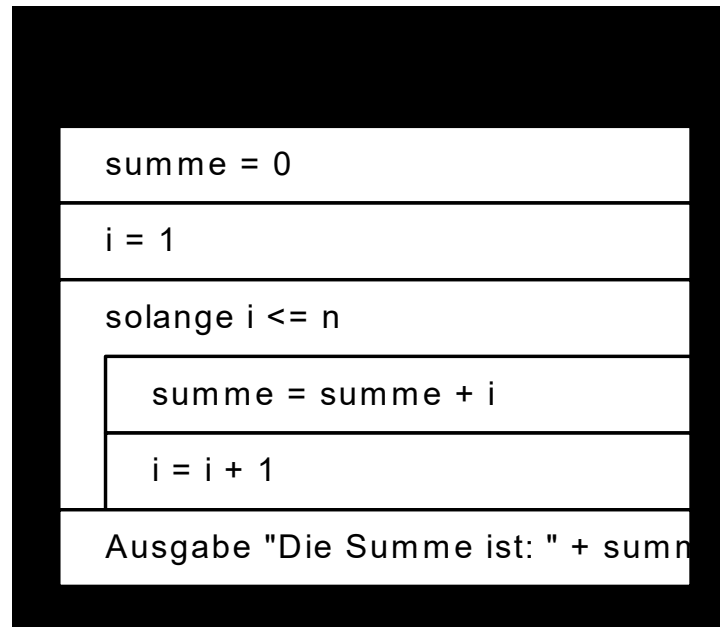
■ **Beispiele:**

- ◆ Erweiterung einer Adressenverwaltung für inländische Kunden auf Kunden im Ausland
- ◆ Umstellung der Postleitzahl von vier auf fünf Stellen
- ◆ Einführung des Euro
- ◆ "Jahr 2000" Problem (vier- statt zweistellige Jahreszahlen)
- ◆ "Jahr 2010" Problem

- Haben wir bisher mit Struktogrammen kennen gelernt
- Genauso üblich:
 - ◆ Natürliche Sprache
 - ◆ Pseudocode (halbformell)
 - ◆ Programmablaufplan
(wie Struktogramm grafisch)
 - ◆ Formale Sprache
(Programmiersprache, formaler Text)

Beispiel: SummeBis(n)

- **Aufgabe:** Berechne die Summe der Zahlen von 1 bis n
- **Natürliche Sprache:** Initialisiere eine Variable summe mit 0. Durchlaufe die Zahlen von 1 bis n mit einer Variable zähler und addiere zähler jeweils zu summe. Gib nach dem Durchlauf den Text "Die Summe ist: " und den Wert von summe aus
- **Struktogramm:**



Schreibtischtest für Algorithmen

- Veranschaulichung der Funktionsweise eines Algorithmus durch ein Ablaufprotokoll der Variableninhalte
- **Beispiel:** Vertausche die Inhalte der Variablen x und y

- 1. Versuch:

$$x = y$$

$$y = x$$

x	y
3	5
5	
	5

- Lösung: mit Hilfsvariable

$$h = x$$

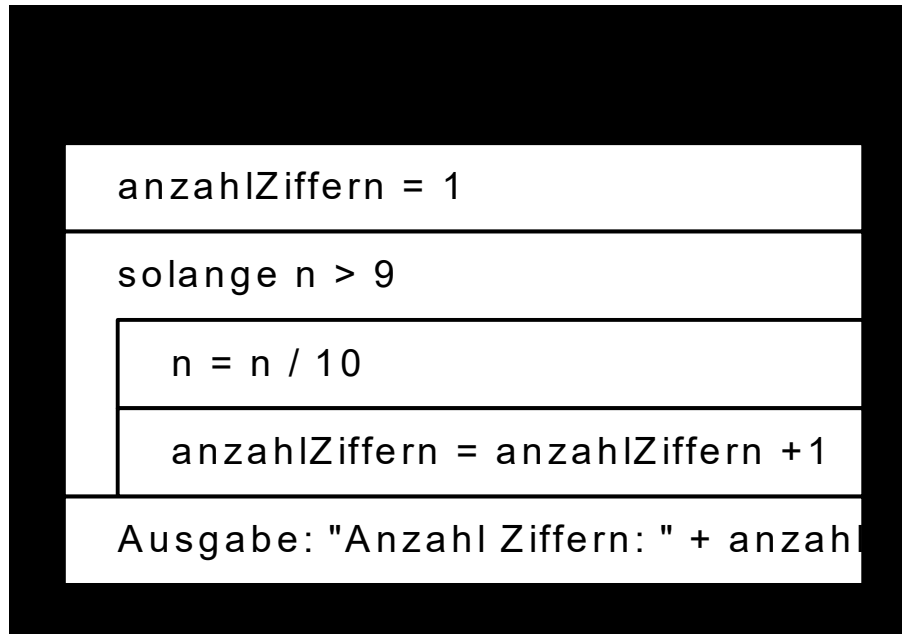
$$x = y$$

$$y = h$$

x	y	h
3	5	
		3
5		
	3	

- Für kleine Algorithmen, Teillösungen ist der Schreibtischtest ein geeignetes Mittel zur Prüfung der Korrektheit

- Algorithmus NumDigits(n)
Liefere die Anzahl Dezimalziffern einer Zahl n zurück



- Schreibtischtest

n	anzahlZiffern
437	1
43	2
4	3

Euklidischer Algorithmus – 1



Euklid
(um 300 v. Chr)



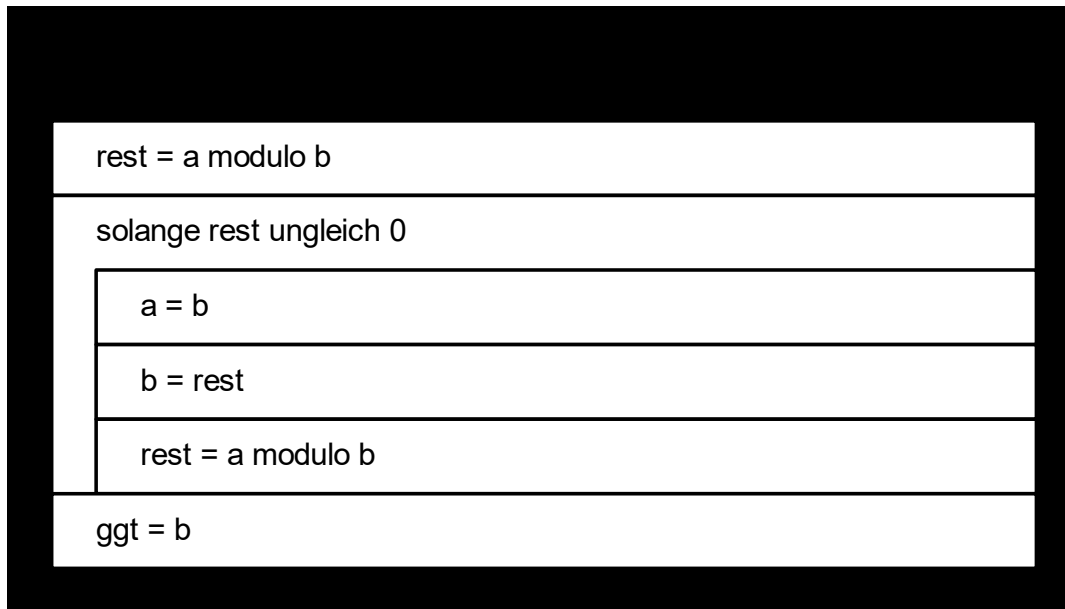
■ Aufgabe:

Finde den größten gemeinsamen Teiler zweier Zahlen a und b

■ Natürliche Sprache:

- ◆ Bilde den Rest gleich a modulo b
- ◆ Solange der Rest ungleich 0 ist:
 - setze a gleich b , b gleich dem Rest und
 - bilde den Rest gleich a modulo b
- ◆ Der größte gemeinsame Teiler ist der Wert von b

Euklidischer Algorithmus – 2



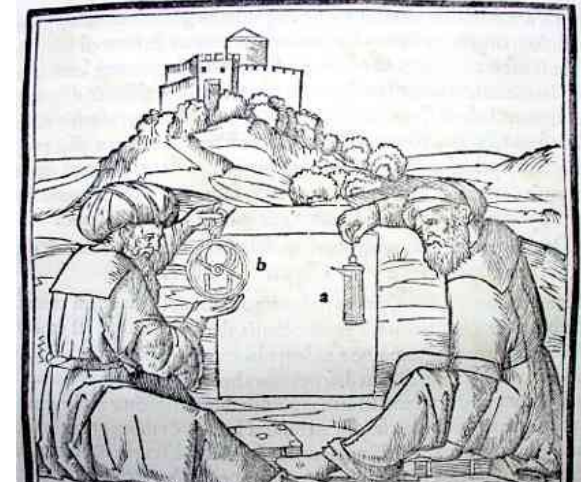
■ Frage:
Was "passiert" bei
 $b > a$?

- Nachweis der Korrektheit:
Aus $(\text{ggt teilt } a) \wedge (\text{ggt teilt } b)$
 - \Rightarrow ggt teilt $(a-b)$
 - \Rightarrow ggt teilt $(a - k*b)$
 - \Rightarrow ggt teilt rest
 - \Rightarrow größter gemeinsamer Teiler $(a,b) ==$ größter gemeinsamer Teiler (b, rest)
- und damit die Aufgabenstellung auf den nächsten Iterationsschritt zurückgeführt

Sieb des Eratosthenes – 1



Eratosthenes von Kyrene,
hellenistischer Gelehrter, um 276-
195 v. Chr.



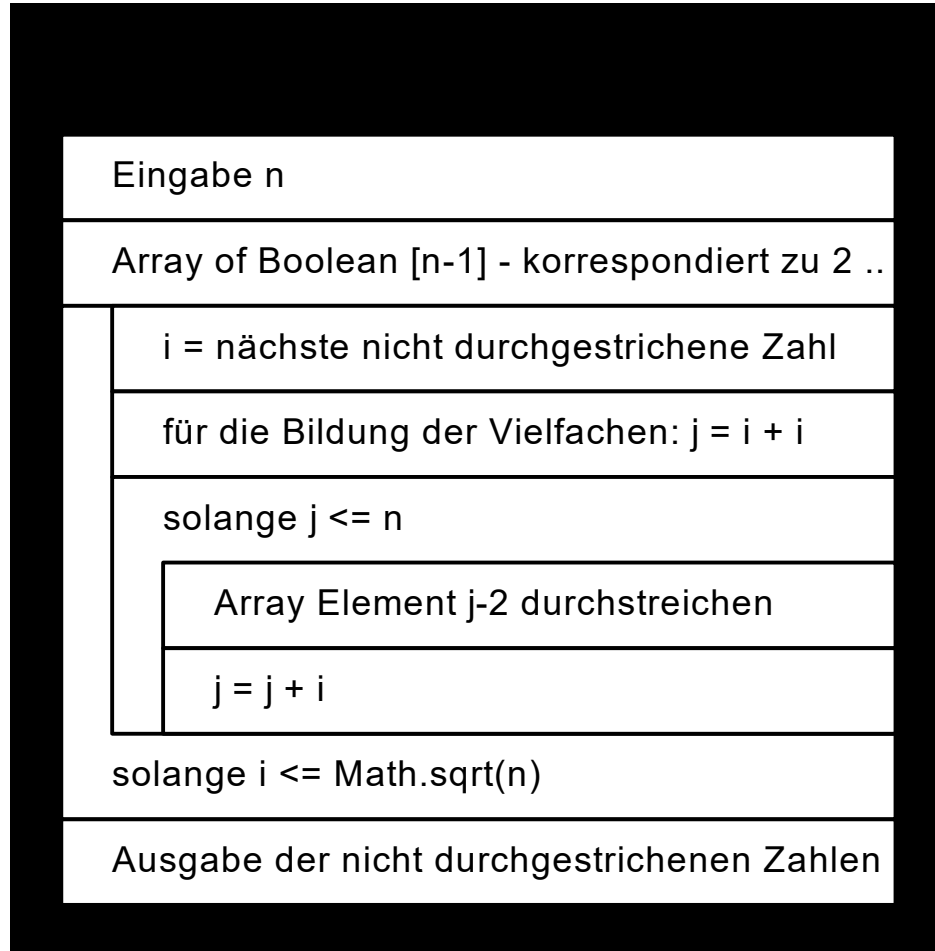
Aufgabe: Finde alle Primzahlen zwischen 2 und n

Das Sieb des Eratosthenes funktioniert im klassischen Sinne folgendermaßen:

- Schreibe alle natürlichen Zahlen von 2 bis zu einer beliebigen Zahl n auf.
- Streiche alle Vielfachen von 2 heraus.
- Gehe zur nächst größeren nicht gestrichenen Zahl und streiche deren Vielfache heraus.
- Wiederhole 3. sooft es geht.
- Die übrig gebliebenen Zahlen sind Primzahlen.

Effizienz ?!

- Welchen Datentyp wählen Sie zur Speicherung der Zahlen?
Array of Boolean, n-1 Elemente
- Bis zu welcher Grenze bilden wir Vielfache?
Durchlauf bis $\text{Math.sqrt}(n)$
- **Offensichtlich kann fast jeder Algorithmus verbessert werden, umso mehr Informationen man einfließen lässt**



- Was aber tun, wenn man nicht die geringste Idee für das "Herangehen" an einen Algorithmus hat?
- Man betrachte das Gesamtproblem als eine (ggf. verschachtelte) Folge von kleineren Teilproblemen, die einzeln gelöst werden
- Man teilt (engl. to divide) das große Problem in beherrschbare (engl. to conquer) Teilprobleme, daher der Name: Teile und herrsche bzw. Divide and Conquer
Manchmal auch scherzhaft: Salami taktik (Scheibe für Scheibe)
- Diese Zerlegung bringt gleichzeitig eine Verringerung der Komplexität des Gesamt-Algorithmus

Vorgehen:

- **Divide:** Teile das Problem in (wenigstens) zwei annähernd gleich große Teilprobleme
- **Conquer:** Löse dieses Problem oder zerlege es nach Schritt 1, wenn es noch zu komplex ist
- **Merge:** Füge die Teillösungen sinnvoll zu einer Lösung des Gesamtproblems zusammen

Beispiel: Multiplikation zweier vierstelliger Zahlen (Schulverfahren)

$$\begin{array}{r} 5432 * 1995 \\ \hline 5432 \\ 48888 \\ 48888 \\ 27160 \\ \hline 10836840 \end{array}$$

Komplexität:

■ 4 Multiplikation

■ 3 Additionen

- Verbesserung durch Rückführung auf Multiplikation von 2-stelligen Zahlen:

$$\begin{array}{|c|c|} \hline \mathbf{A} & \mathbf{B} \\ \hline \mathbf{54} & \mathbf{32} \\ \hline \end{array} * \begin{array}{|c|c|} \hline \mathbf{C} & \mathbf{D} \\ \hline \mathbf{19} & \mathbf{95} \\ \hline \end{array}$$

$$\begin{array}{rcl} A * C & = & 54 * 19 & = & 1026 \\ (A + B) * (C + D) - A * C - B * D & = & 86 * 114 - 1026 - 3040 & = & 5738 \\ B * D & = & 32 * 95 & = & 3040 \\ & & & & \hline & & & & \mathbf{10836840} \end{array}$$

- Prinzip ist auf n-stellige Zahlen verallgemeinbar
- Komplexität
 - ◆ 3 Multiplikationen 2-stelliger Zahlen: $A * C$, $B * D$, $(A + B) * (C + D)$
 - ◆ 6 Additionen (wesentlich schneller als Multiplikationen)

??? **Fragen**



***Welche Fragen
gibt es?***

JETZT!



Rekursion

- Bisher haben wir **iterative** Algorithmen betrachtet
 - ◆ leitet sich vom lateinischen "iteratio" ab und bedeutet "Wiederholung"
 - ◆ In der Informatik überträgt sich dieser Begriff auf die Wiederholung von Schleifendurchläufen innerhalb eines Programms
 - ◆ Ein iterativer Algorithmus ist demnach eine Folge von Anweisungen, die ein Problem durch mehrfaches Durchlaufen einer oder mehrerer (ggf. verschachtelter) Schleifen löst

- Betrachten wir jetzt einen anderen Algorithmustyp, den **rekursiven** Algorithmus

- Im Duden ist definiert:
rekursiv = "zurückgehend bis zu bekannten Werten"
- Die Idee:
Ein Algorithmus ruft sich immer wieder selbst auf bis durch ein Abbruchkriterium die Aufrufkette abgebrochen wird
- Das Abbruchkriterium besitzt damit eine gewisse Analogie zur Schleifenbedingung
- Ein endloser rekursiver Algorithmus (das Abbruchkriterium wird nie erreicht) wird **unterminierte Rekursion** genannt



- Viele mathematische Funktionen sind **rekursiv** definiert
- Dabei wird zum Beispiel der Wert $f(n)$ einer einstelligen Funktion f auf dem Bereich der natürlichen Zahlen für $n > 1$ auf einem oder mehreren Werten von $f(1), \dots, f(n-1)$ basierend berechnet
- Zumindest ein Startwert $f(1)$ wird dabei als Start für die Rekursion fest vorgegeben
- **Beispiel:** Fakultätsfunktion

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)! & \text{falls } n \geq 1 \end{cases}$$

- Die Definition der Fakultätsfunktion in Java kann zum Beispiel so aussehen:

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

- Aufgrund der Sichtbarkeitsregeln in Java kann der Name einer Methode im Rumpf derselben Methode benutzt werden

- Der Aufruf der Fakultätsfunktion mit dem Wert 4 führt zu folgender Aufrufkette

faculty(4)

→ faculty(4 * faculty(3))

→ faculty(4 * faculty(3 * faculty(2)))

→ faculty(4 * faculty(3 * faculty(2 * faculty(1))))

→ faculty(4 * faculty(3 * faculty(2 * faculty(1 * faculty(0)))))

= 24

Berechnung der Fakultätsfunktion



- Beim rekursiven Aufruf einer (Fakultäts-) Funktion wird (wie bereits bekannt) für jeden Aufruf ein Aktivierungssatz angelegt
- Daher gibt es keine Konflikte, obwohl (im Beispiel) 5 Integer-Variablen namens n gleichzeitig aktiv sind
- Die Aktivierungssätze werden wieder entfernt, wenn die entsprechenden Funktionsaufrufe beendet sind und der Ergebniswert vorliegt

Berechnung der Fakultätsfunktion

- Beim rekursiven Aufruf einer (Fakultäts-) Funktion wird (wie bereits bekannt) für jeden Aufruf ein Aktivierungssatz angelegt
- Daher gibt es keine Konflikte, obwohl (im Beispiel) 5 Integer-Variablen namens n gleichzeitig aktiv sind
- Die Aktivierungssätze werden wieder entfernt, wenn die entsprechenden Funktionsaufrufe beendet sind und der Ergebniswert vorliegt



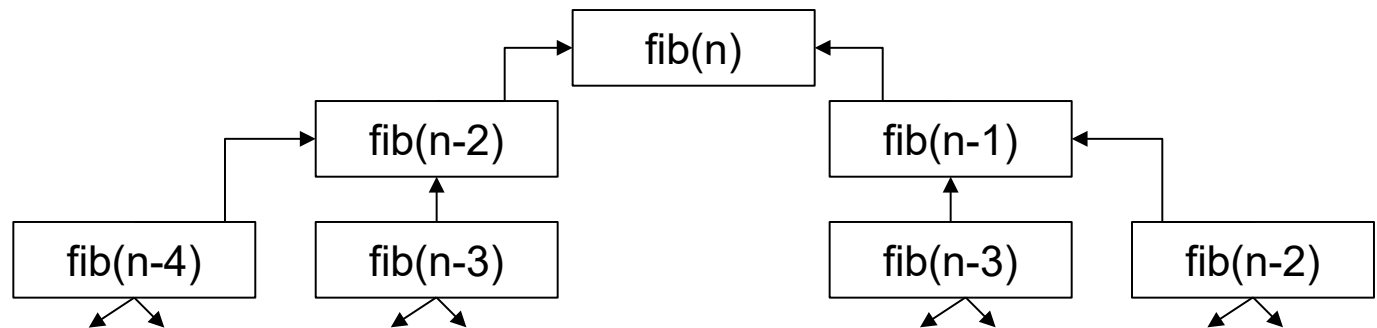
- Die Fibonacci-Funktion ist mathematisch wie folgt definiert:



Leonardo Pisano
Fibonacci

1170 – 1250, Pisa

$$\text{fib}(n) = \begin{cases} 0 & \text{falls } n == 0 \\ 1 & \text{falls } n == 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{falls } n \geq 2 \end{cases}$$



- Liefert die Folge: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- Die Vermehrung von Kaninchen sollte modelliert werden
- Problem: Explosion der Aufrufe = 2^n

Rekursion vs. Iteration

- Rekursion ist ein mächtiges, allgemeines Programmierprinzip
- **Jede** rekursive Lösung lässt sich (zumindest theoretisch) auch iterativ lösen

im Vergleich:

- Die rekursive Formulierung ist oft eleganter
- Die iterative Lösung ist oft effizienter aber komplizierter

Kriterien für "gute" Rekursion – 2

■ Schema der Rekursion

```
if ( problemKleinGenug )
    nichtRekursiverZweig; // Terminierungsfall
else
    rekursiverZweigMitKleineremProblem;
```

■ Nachweis der Terminierung

- ◆ Es gibt einen Terminierungszweig
d.h. für mindestens ein Argument ist das Ergebnis bekannt
- ◆ Das Problem wird bei jedem rekursiven Aufruf signifikant "kleiner", d.h. der "Abstand" zum Terminierungsfall wird kleiner

Wann Rekursion, wann Iteration?

- Rekursion ist vorzuziehen, wenn
 - ◆ die Rekursionstiefe gering
 - ◆ Formulierung des iterativen Vorgehens nicht trivial
 - zu langer und
 - unübersichtlicher Quellcode entsteht
 - ◆ die beanspruchten Ressourcen (z.B. Speicherplatz) vernachlässigbar klein sind
 - Aufrufstapel
 - Lokale Variablen

- Wir haben rekursive Funktionen kennengelernt
- Diese haben in der Mathematik ihre Entsprechung, sind also auch in Programmiersprechen wie Java vertreten
- Es gibt aber auch Fälle, in denen rekursive Prozeduren sinnvoll eingesetzt werden können
- Java unterstützt rekursive Prozeduren genau so wie rekursive Funktionen
 - ◆ Für jeden Aufruf einer Prozedur wird ein eigener Aktivierungssatz angelegt
 - ◆ Diese Aktivierungssätze werden nach Ablauf der Prozedur automatisch wieder entfernt
 - ◆ Üblicherweise vollbringen Prozeduren ihre Aufgaben durch Seiteneffekte
(die sich auf die Parameter oder auf externe Dateien beschränken sollten)

Die Grundschrirte dieses Algorithmenmusters:

- Eine Menge von Eingabewerten ist gegeben
- Es ist eine Menge von Lösungen, die aus den Eingabewerten aufgebaut sind, prinzipiell bekannt
- Die Lösungen lassen sich schrittweise aus partiellen Lösungen beginnend mit der leeren Lösung, durch Hinzunahme von Eingabewerten aufbauen
- Es wird eine Bewertungsfunktion für die partiellen und schließlich vollständigen Lösungen benötigt und verwendet
- Gesucht wird dann die bzw. eine optimale Lösung

■ **Aufgabenstellung:**

Ein bestimmter Betrag soll aus einer minimalen Anzahl von Münzen bezahlt werden.

Beispiel: 28 Cent = 20 Cent + 5 Cent + 2 Cent + 1 Cent

■ **Algorithmus:**

- Wähle die größtmögliche Münze aus dem Zielwert (Betrag) aus
- Ziehe den Wert vom Zielwert ab
- Gehe zu Schritt 1 solange Differenz vom Zielwert und Münzwert größer als 0

Greedy-Algorithmen – 3

so werden
Kommandozeilen-
Parameter abge-
fragt. Hier wird in
args[0] der zu
zahlende Betrag
übergeben.
Vgl. auch nächste
Folie

```
public class Greedy {
    public static void boerse(int betrag) {
        int[] muenzen = {1,2,5,10,20,50};
        int i = muenzen.length - 1;
        if (betrag != 0) {
            while (muenzen[i] > betrag) {
                i--;
            }
            System.out.println(muenzen[i] + " Cent");
            boerse(betrag -= muenzen[i]);
        }
    }
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Beim Aufruf Cent-Betrag angeben.");
            System.exit(0);
        }
        int zahlBetrag = Integer.parseInt(args[0]);
        System.out.println(zahlBetrag + " Cent bezahlt man mit :");
        boerse(zahlBetrag);
    }
}
```

```
Command Prompt

C:\>java Greedy 28
28 Cent bezahlt man mit :
20 Cent
5 Cent
2 Cent
1 Cent

C:\>java Greedy 99
99 Cent bezahlt man mit :
50 Cent
20 Cent
20 Cent
5 Cent
2 Cent
2 Cent

C:\>java Greedy
Beim Aufruf Cent-Betrag angeben.

C:\>
```



Es gibt eine alte Sage:

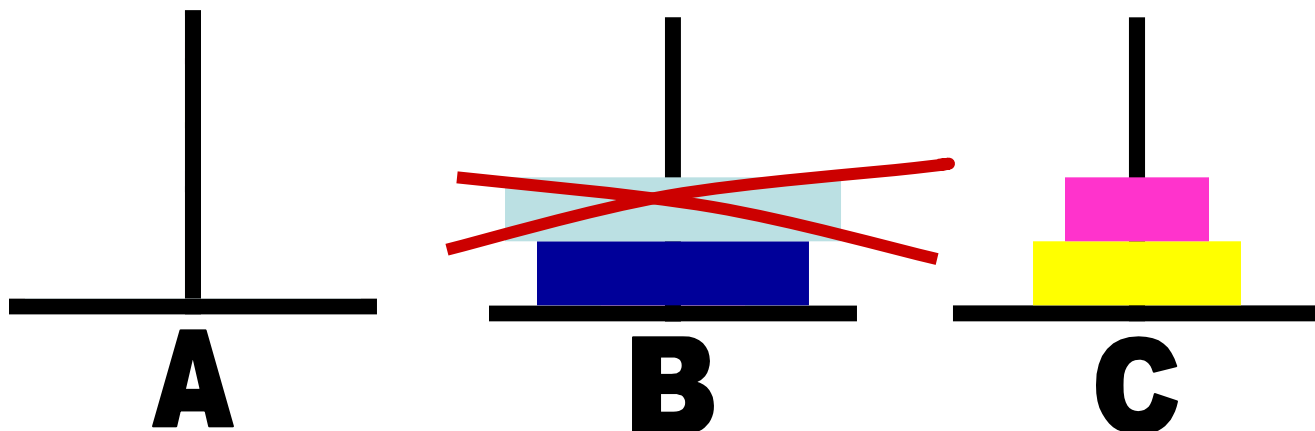
Die Mönche eines Brahma-Tempels in der Nähe von Hanoi arbeiten an der Lösung eines Problems. 64 Scheiben unterschiedlichen, nach oben verjüngenden Durchmessers lagern auf einem Stab a

Sie sollen verlagert werden auf einen zweiten Stab b, aber niemals darf eine Scheibe mit einem größeren Durchmesser auf eine mit einem kleineren platziert werden. Zur temporären Auslagerung dient ein dritter Stab c

Wenn dieses Puzzle gelöst ist, ist das **Ende aller Dinge** da und die Welt geht unter ...

Die Türme von Hanoi

- Eines der prominentesten Beispiele für Rekursion ist das Problem der Türme von Hanoi
- Die Aufgabe besteht darin, n Scheiben verschiedener Durchmesser von einem Platz A zu einem Platz B zu transportieren
- Dabei dürfen Scheiben auf einem Platz C zwischengelagert werden
- Es ist verboten, eine Scheibe auf eine andere Scheibe kleineren Durchmessers zu legen

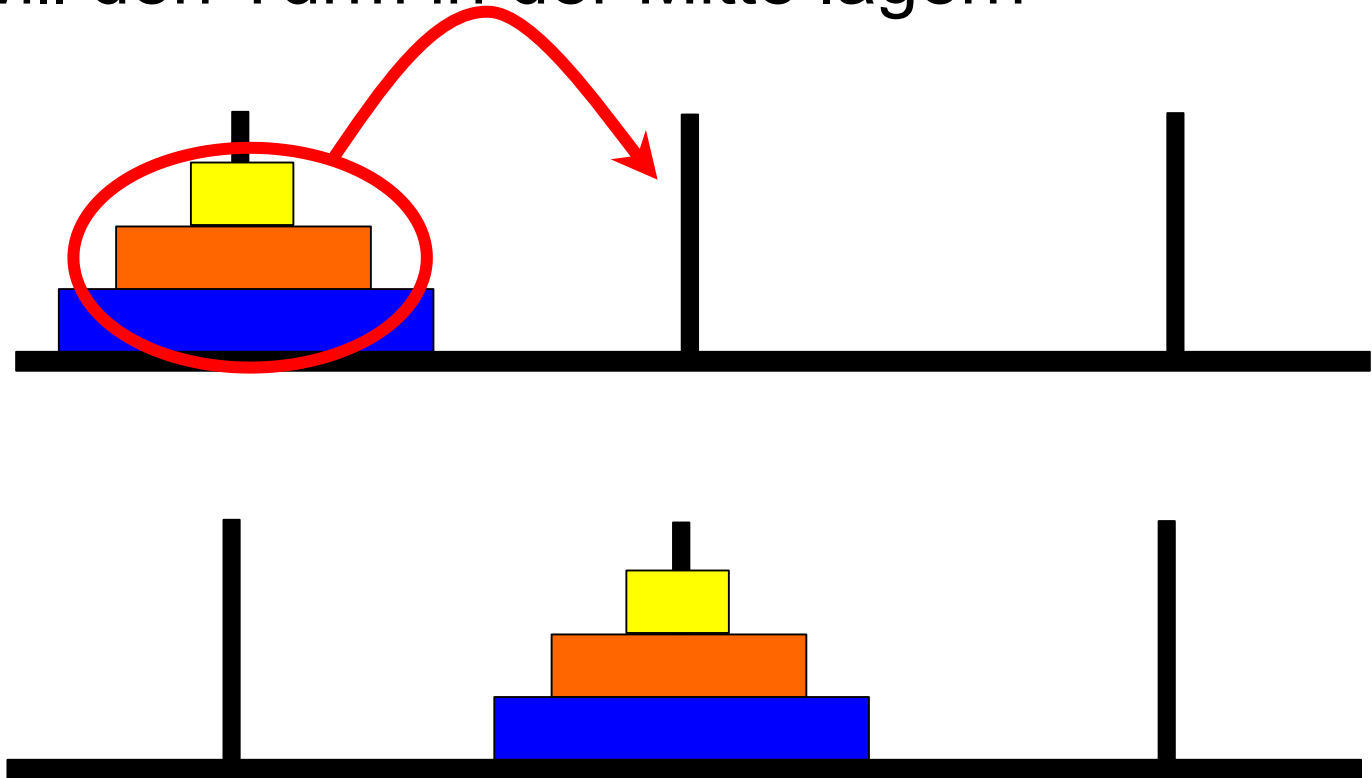


Die Türme von Hanoi – Rekursion

- Das Problem lässt sich mit folgender Strategie lösen
 - Schaffe N-1 Scheiben von Platz A nach Platz C
 - Schaffe 1 Scheibe von Platz A nach Platz B
 - Schaffe N-1 Scheiben von Platz C nach Platz B
- Das Problem ist kleiner geworden:
Schafft man die Lösung für N-1 Scheiben, dann hat man eine Anleitung, wie das Problem für N Scheiben zu lösen ist
- Es handelt sich um eine rekursive Lösung:
Für die Lösung des Problems in Teil 1 darf Platz C und in Teil 3 darf Platz A für die Zwischenlagerung von Scheiben benutzt werden
- Ist das Problem entsprechend weit reduziert, muss nur eine einzelne Scheibe transportiert werden (Rekursionsstart mit $N = 1$)

Die Türme von Hanoi – Detail

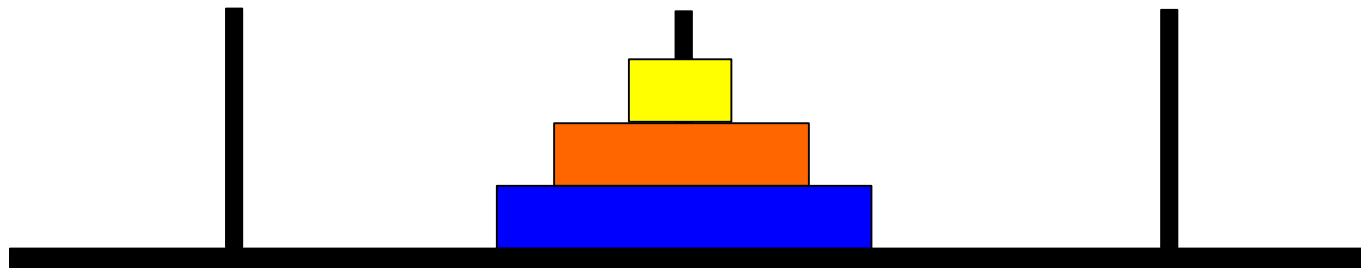
- Ich will den Turm in der Mitte lagern



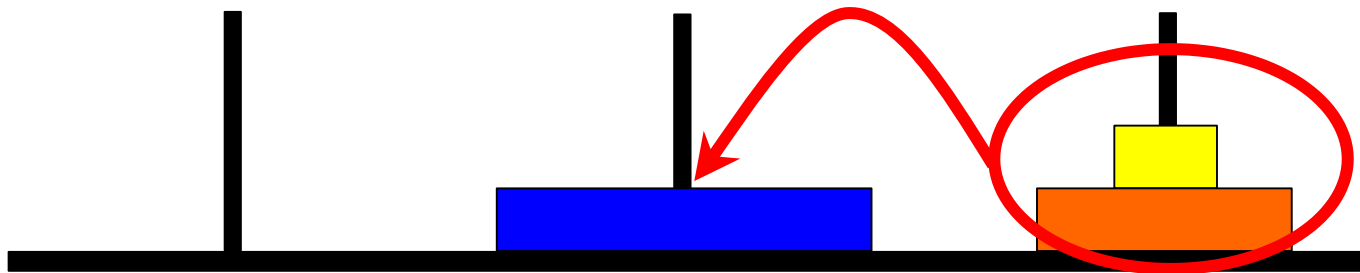
Diesen Schritt kann man wie Erreichen?

Die Türme von Hanoi – Detail

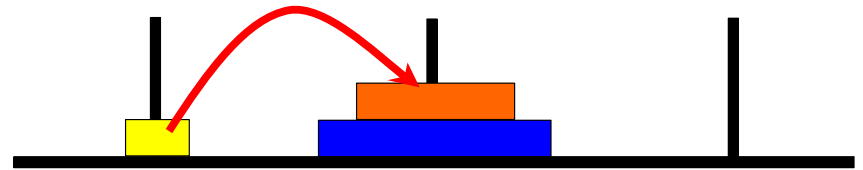
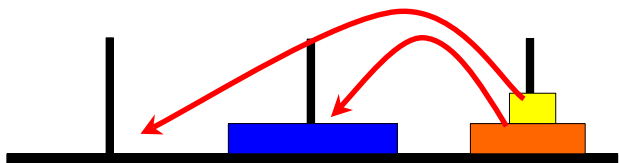
- Schritt (1) könnte erreicht man:



Durch:

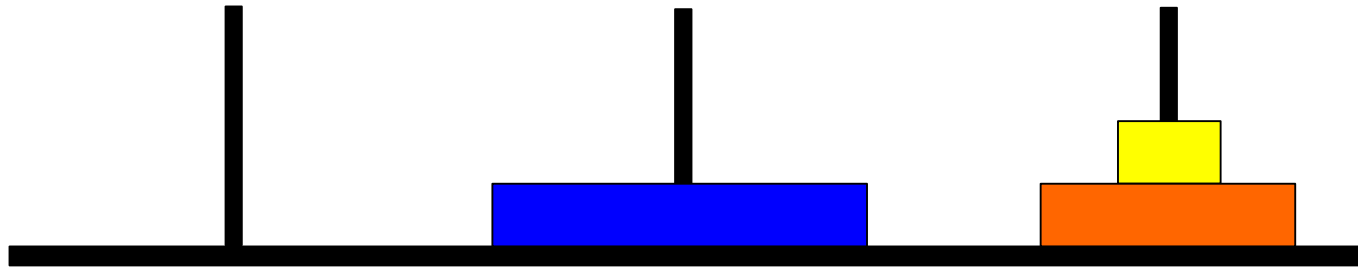


Teilschritte:

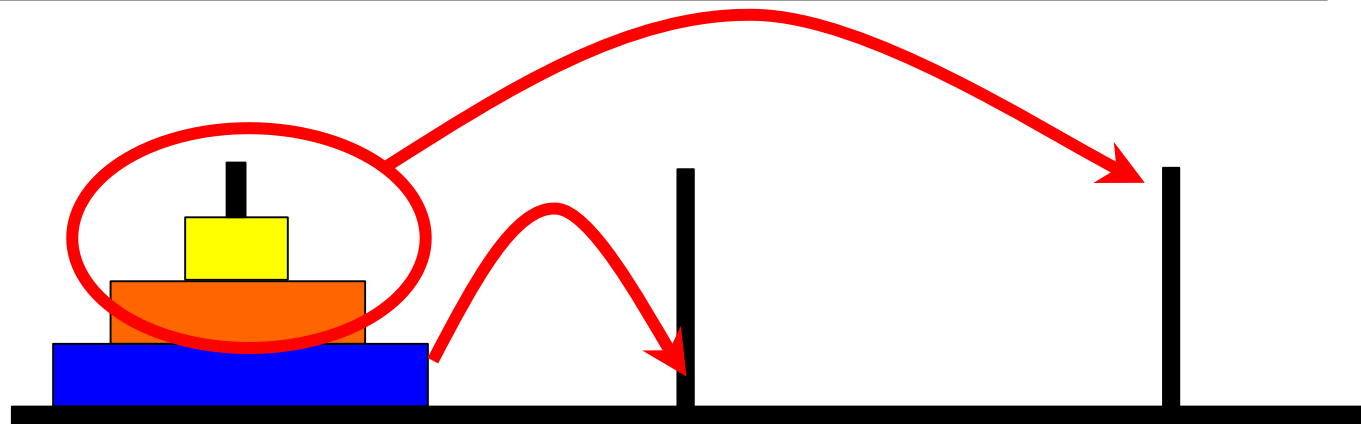


Die Türme von Hanoi – Detail

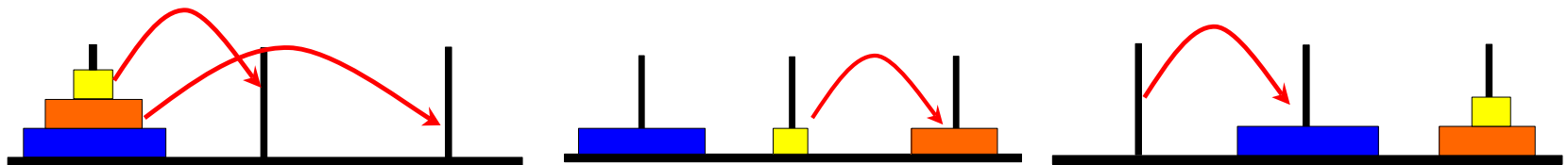
- Versuchen wir nun diesen Schritt(2) zu erreichen:



Durch:



Teilschritte:



Die Türme von Hanoi – Lösung

- In Java kann das Problem wie folgt rekursiv gelöst werden

```
static void hanoi ( int n, char von, char nach, char tmp ) {  
    if ( n == 1 )  
        System.out.println( "M1:Schaffe Scheibe von "+von+" nach "+nach );  
  
    else {  
        hanoi( n-1, von, tmp, nach );  
        System.out.println( "M2:Schaffe Scheibe von "+von+" nach "+nach );  
        hanoi( n-1, tmp, nach, von );  
    }  
}
```

Aufruf: hanoi(4, 'a', 'b', 'c'); oder hanoi(3, 'a', 'b', 'c');

Der Seiteneffekt der Prozedur besteht darin, die Anweisungen zum Transportieren der Scheiben auf dem Bildschirm auszugeben

Die Türme von Hanoi – Detail

Das passiert mit 3 Scheiben:



Aufruf1 von Ebene 0 (main)n: 3 von:a nach:b tmp:c

Aufruf2 von Ebene 1 n: 2 von:a nach:c tmp:b

Aufruf3 von Ebene 2 n: 1 von:a nach:b tmp:c

M1:Schaffe Scheibe von a nach b Ebene (3)

M2:Schaffe Scheibe von a nach c Ebene (1)

Aufruf4 von Ebene 2 n: 1 von:b nach:c tmp:a

M1:Schaffe Scheibe von b nach c Ebene (4)

M2:Schaffe Scheibe von a nach b Ebene (0)

Aufruf5 von Ebene 1 n: 2 von:c nach:b tmp:a

Aufruf6 von Ebene 2 n: 1 von:c nach:a tmp:b

M1:Schaffe Scheibe von c nach a Ebene (6)

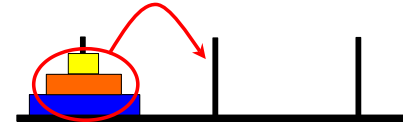
M2:Schaffe Scheibe von c nach b Ebene (1)

Aufruf7 von Ebene 2 n: 1 von:a nach:b tmp:c

M1:Schaffe Scheibe von a nach b Ebene (7)

Die Türme von Hanoi – Das Programm

Aufruf1 von Ebene 0 (main)n: 3 von:a nach:b tmp:c

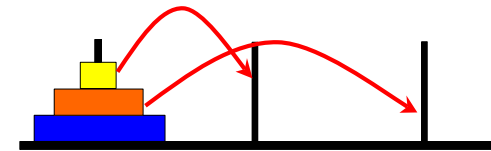


Aufruf2 von Ebene 1 n: 2 von:a nach:c tmp:b

Aufruf3 von Ebene 2 n: 1 von:a nach:b tmp:c

M1:Schaffe Scheibe von a nach b Ebene (3)

M2:Schaffe Scheibe von a nach c Ebene (1)



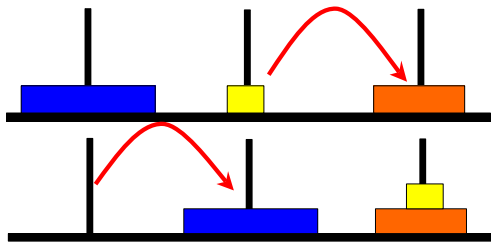
Aufruf4 von Ebene 2 n: 1 von:b nach:c tmp:a

M1:Schaffe Scheibe von b nach c Ebene (4)

M2:Schaffe Scheibe von a nach b Ebene (0)

Aufruf5 von Ebene 1 n: 2 von:c nach:b tmp:a

Aufruf6 von Ebene 2 n: 1 von:c nach:a tmp:b

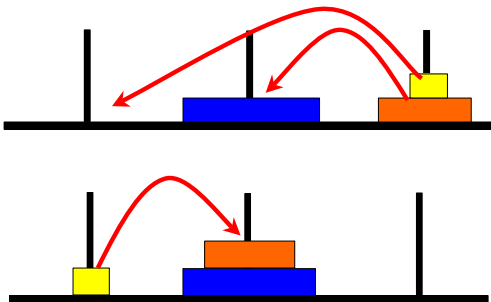


M1:Schaffe Scheibe von c nach a Ebene (6)

M2:Schaffe Scheibe von c nach b Ebene (1)

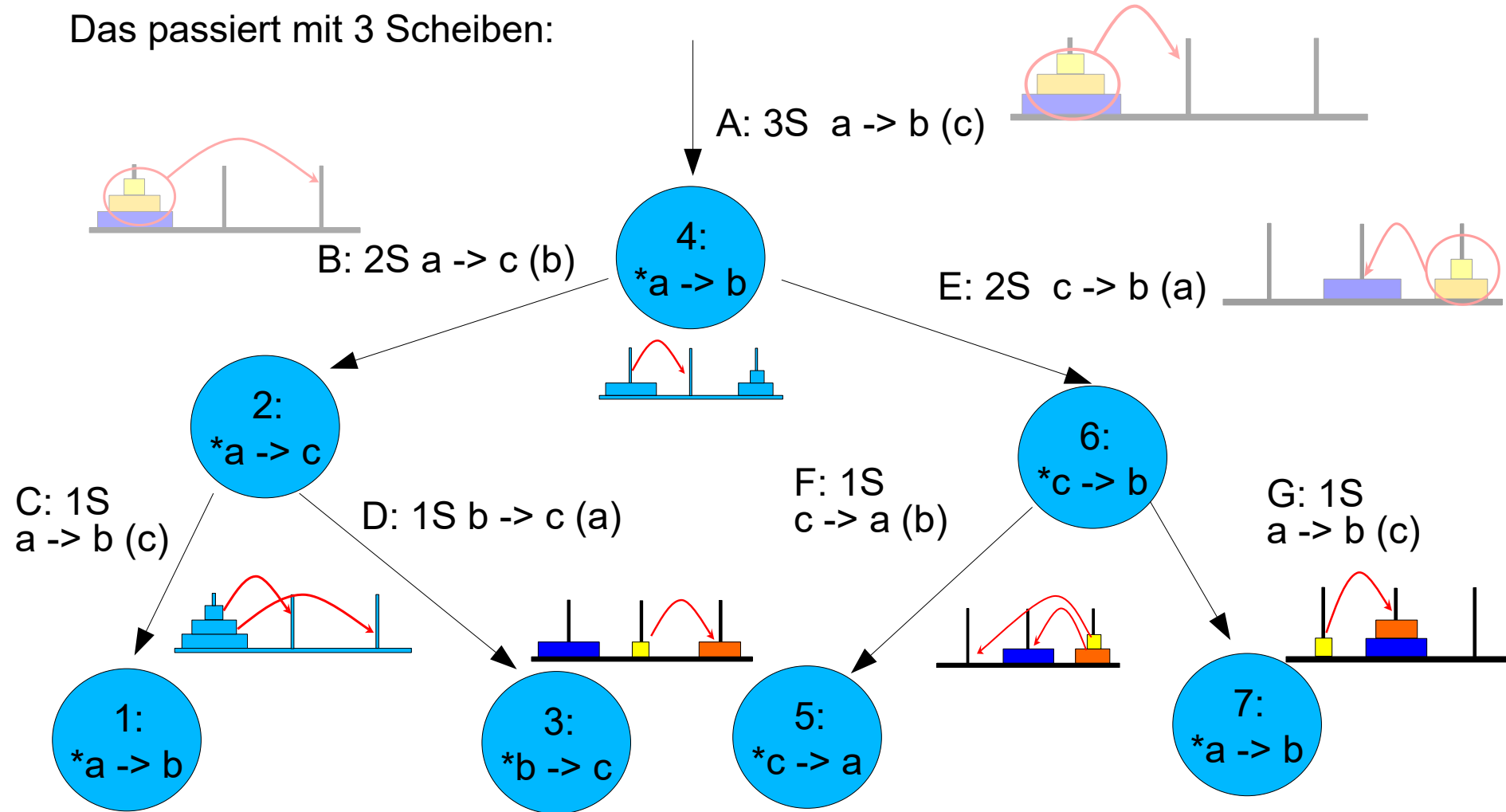
Aufruf7 von Ebene 2 n: 1 von:a nach:b tmp:c

M1:Schaffe Scheibe von a nach b Ebene (7)



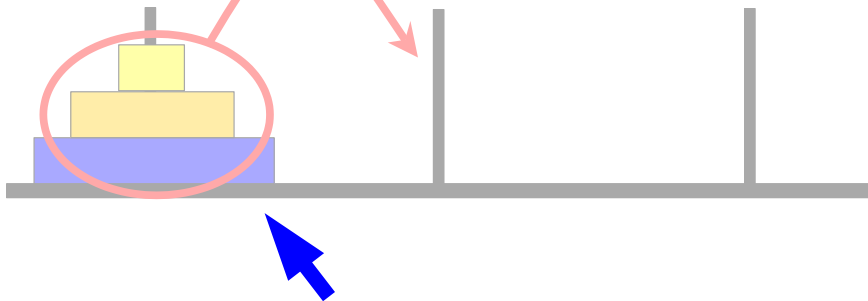
Die Türme von Hanoi – Rekursionsbaum

Das passiert mit 3 Scheiben:



Die Türme von Hanoi – Rekursionsbaum

Rekursion



Aufruf1 von Ebene 0 (main)n: 3 von:a nach:b tmp:c

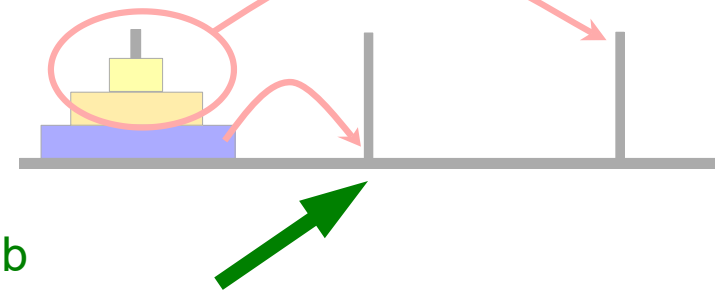
Aufruf2 von Ebene 1 n: 2 von:a nach:c tmp:b

Aufruf3 von Ebene 2 n: 1 von:a nach:b tmp:c

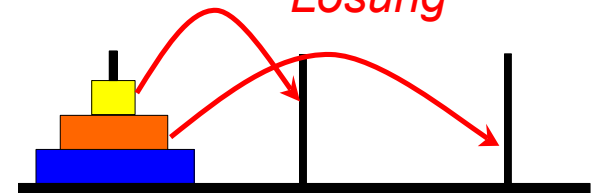
M1:Schaffe Scheibe von a nach b Ebene (3)

M2:Schaffe Scheibe von a nach c Ebene (1)

Rekursion

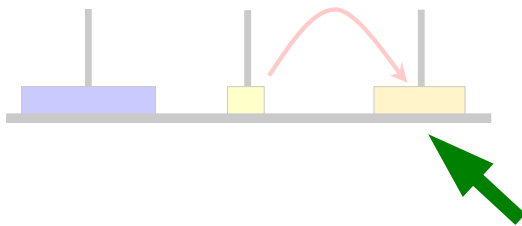


Lösung



Die Türme von Hanoi – Rekursionsbaum

Rekursion

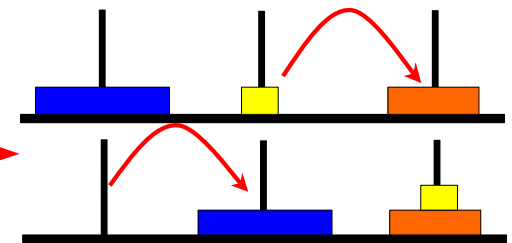


Aufruf4 von Ebene 2 n: 1 von:b nach:c tmp:a

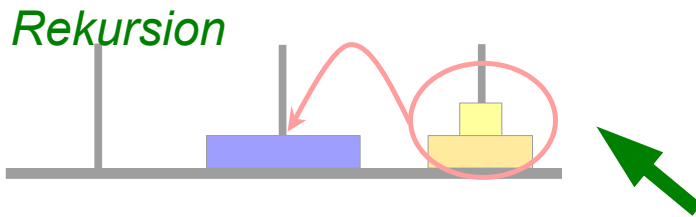
M1:Schaffe Scheibe von b nach c Ebene (4)

M2:Schaffe Scheibe von a nach b Ebene (0)

Lösung



Die Türme von Hanoi – Rekursionsbaum



Aufruf5 von Ebene 1 n: 2 von:c nach:b tmp:a

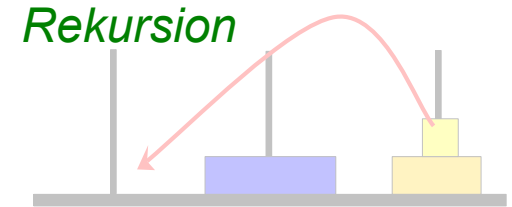
Aufruf6 von Ebene 2 n: 1 von:c nach:a tmp:b

M1:Schaffe Scheibe von c nach a Ebene (6)

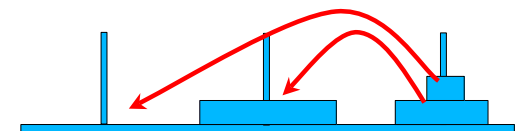
M2:Schaffe Scheibe von c nach b Ebene (1)

Aufruf7 von Ebene 2 n: 1 von:a nach:b tmp:c

M1:Schaffe Scheibe von a nach b Ebene (7)



Lösung



Lösung/Rekursion

Die Türme von Hanoi – Was macht das Programm

```
static void hanoi ( int n, char von, char nach, char tmp ) {
```

```
    if ( n == 1 )
```

```
        System.out.println( "M1:Schaffe Scheibe von "+von+" nach "+nach );
```

```
    else {
```

```
        hanoi( n-1, von, tmp, nach );
```

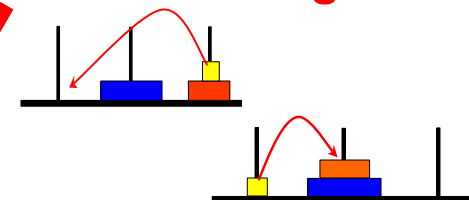
```
        System.out.println( "M2:Schaffe Scheibe von "+von+" nach "+nach );
```

```
        hanoi( n-1, tmp, nach, von );
```

```
    }
```

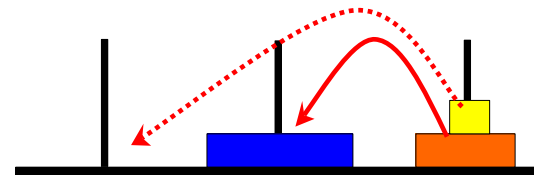
```
}
```

„Obere Scheibe bewegen“



„Umschichten auf den Hilfsstab“

„nach Rekursion umschichten“



„Umschichten vom Hilfstab auf Zielposition“

Die Türme von Hanoi – Komplexität

- Die Anzahl der Züge bewegt sich in Zweierpotenzen proportional zur Anzahl der Scheiben

Scheiben	Züge	Zweierpotenz
1	1	2^1-1
2	3	2^2-1
3	7	2^3-1
4	15	2^4-1
5	31	2^5-1
6	63	2^6-1
7	127	2^7-1
...

Vorsicht *FALLE*



- Sollten Sie jetzt das Ende aller Dinge vor dem Ende Ihres Studiums herannahen fürchten:
- Betrachten wir die **Komplexität**: 64 Scheiben, angenommener Zeitbedarf für das Umschichten einer Scheibe: 1 Sekunde – ergibt:
- $2^{64} - 1$ Umschichtungen bzw. Sekunden
 - ⇒ ca. $1,8447 * 10^{19}$ Sekunden
 - ⇒ ca. $5,8494 * 10^{11}$ Jahre
 - ⇒ mehr als **585** Milliarden Jahre

Mittlerweile!



- Der schnellste Großrechner (202.752 Cores + 27.648 GPUs / 13 MW) schafft 146.000.000.000.000.000 Operationen / Sek.

- Erstellung des Lösungsweg:

Grob: ~ 1 Stunden 10 Min

→ Ohne Overhead!



- Unter Umständen ist es notwendig, dass sich zwei oder mehr Unterprogramme gegenseitig wiederholt (rekursiv) aufrufen
- Man spricht dann von **Ko-Rekursion**
- **Beispiel:**

```
static void a ( ... ) {  
    ...  
    b( ... );  
    ...  
}  
static void b ( ... ) {  
    ...  
    a( ... );  
    ...  
}
```

- Aufgrund der Sichtbarkeitsregeln in Java ist Ko-Rekursion überhaupt kein Problem

Ko-Rekursion - Beispiel

```
public static boolean gerade (int eingabe ) {  
    if (eingabe == 0)  
        return true;  
    else  
        return ungerade(eingabe - 1);  
}
```

```
public static boolean ungerade (int eingabe) {  
    if (eingabe == 0)  
        return false;  
    else  
        return gerade(eingabe - 1);  
}
```


??? **Fragen**



***Welche Fragen
gibt es?***

JETZT



Suchen

Erster Versuch

- In der täglichen Programmierpraxis stellt sich häufig die Aufgabe, in einem Feld ein Element mit bestimmtem Wert zu suchen
- Zu diesem Zweck existieren verschiedene Such-
Algorithmen,
 - ◆ die sich im Aufbau
 - ◆ Voraussetzungen und
 - ◆ Effizienz unterscheiden
- Für unsere Beispiele betrachten wir im Folgenden
`int[] feld1 = { 54 , 80 , 11 , 91 , 17 , 23 , 58 , 28 };`

```
public class SuchenSortieren {
    public static void ausgabeIntFeld(String bemerkung, int[] ausgabeFeld) {
        int laenge = ausgabeFeld.length;

        System.out.print("\n' + bemerkung + "\n {"");
        for (int i = 0; i < laenge-1; i++) {
            System.out.print(ausgabeFeld[i] + ", ");
        }
        System.out.println(ausgabeFeld[laenge-1] + "}");
    }

    public static void main(String[] args) {
        int[] feld1 = { 54 , 80 , 11 , 91 , 17 , 23 , 58 , 28 };

        ausgabeIntFeld("Ergebnis:", feld1);
    }
}
```

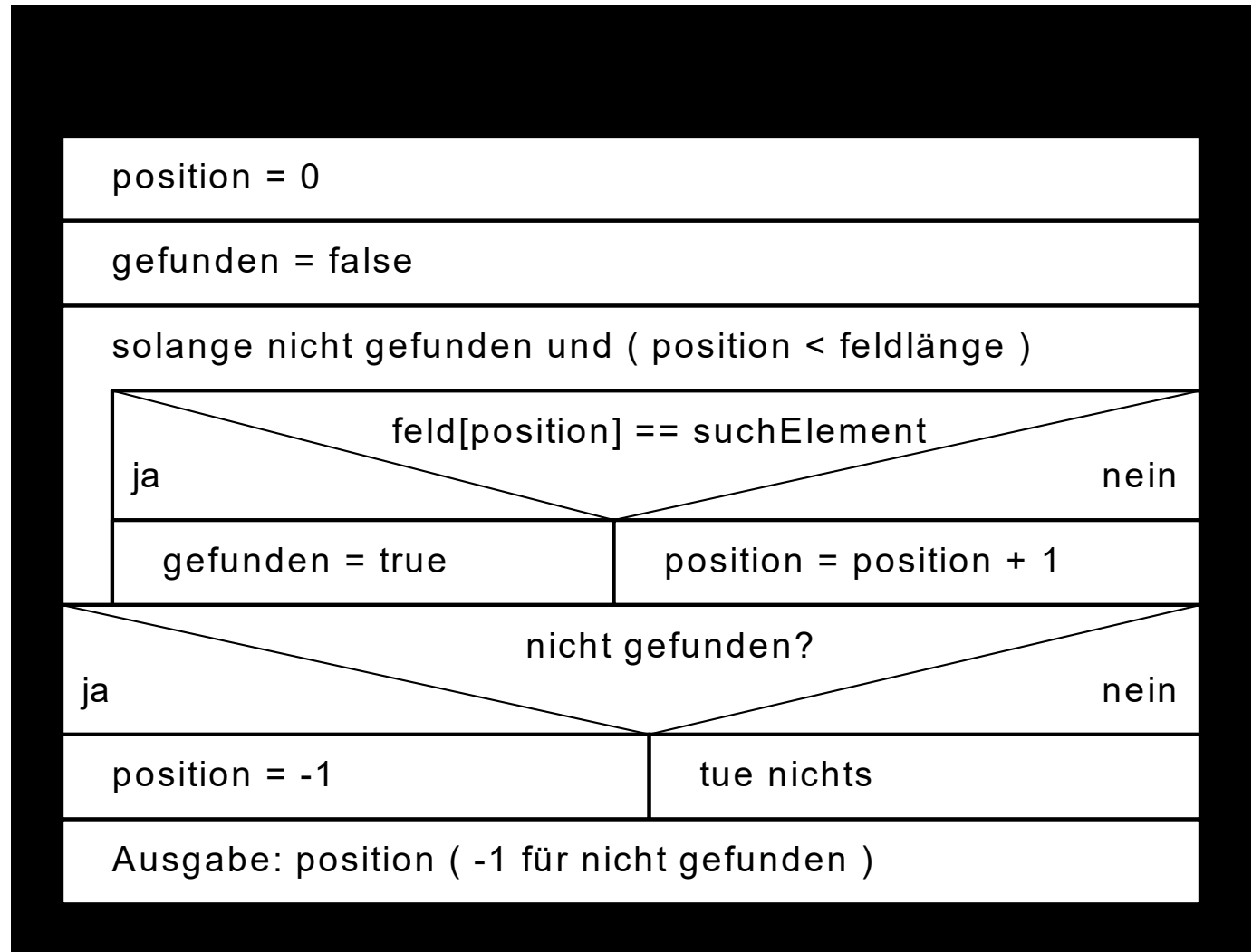
**Dieser Quellcode
dient uns als Rumpf
für die weiteren
Betrachtungen**

Idee:

- Sofern über die zu untersuchende Datenmenge a_1, \dots, a_n (in unserem Fall das Array `feld1`) keine weiteren Informationen vorliegen
- wird die Datenmenge sequentiell durchlaufen
- Dabei wird Schritt für Schritt die Menge vergrößert, in der sich das gesuchte Element x mit Sicherheit nicht befindet
- Der Algorithmus terminiert unter zwei Bedingungen:
 - ◆ Das Element ist gefunden, d.h. $a_i == x$
 - ◆ Das gesamte Array wurde durchlaufen, aber kein Element hat den Schlüssel x

**feld =
unsortiertes Feld
der Länge n**

**suchElement =
Element, dessen
Position bestimmt
werden soll**



Gegeben:

54	80	11	91	17	23	58	28
----	----	----	----	----	----	----	----

Gesucht: 58

54	80	11	91	17	23	58	28
----	----	----	----	----	----	----	----



⇒ gefunden an Position 6 (Rückgabewert)

Gesucht: 27

54	80	11	91	17	23	58	28
----	----	----	----	----	----	----	----



⇒ nicht gefunden, Rückgabewert -1

```
public static int linSuch (int[] feld, int suchElement) {
    int position = 0;
    boolean gefunden = false;
    int feldlaenge = feld.length;

    while (!gefunden && (position < feldlaenge)){
        if (feld[position] == suchElement)
            gefunden = true;
        else
            position ++;
    }
    if (!gefunden)
        position = -1;
    return (position);
}
```



```
int[] feld1 = { 54 , 80 , 11 , 91 , 17 , 23 , 58 , 28 }
```

Aufruf:

- `System.out.println(linSuch(feld1, 58));`
→ Ausgabe: 6
- `System.out.println(linSuch(feld1, 91));`
→ Ausgabe: 3
- `System.out.println(linSuch(feld1, 27));`
→ Ausgabe: -1

- Betrachten wir die Komplexität:
 - ◆ günstigster Fall: 1 Vergleich
 - ◆ ungünstigster Vergleich: n Vergleiche
 - ◆ im Mittel: $n/2$ Vergleiche
- Spielen wir Zahlenraten: Ich denke mir eine Zahl zwischen 1 und 30 – Bitte raten Sie!
- Worin besteht der Unterschied zwischen unserem Beispiel und der Zahlenfolge 1...30?
- Deswegen betrachten wir, bevor wir noch einmal zum Suchen zurückkehren, Sortieralgorithmen

JETZT



Sortieren

- Sortieralgorithmen sollen die Elemente einer Menge (z.B. eines Arrays) nach einem bestimmten Kriterium sortieren
- In einem sortierten Feld können die Elemente leichter gesucht werden
- Die sortierte Ausgabe einer Ergebnisliste (etwa einer Adressverwaltung) ist benutzerfreundlicher als die unsortierte
- Voraussetzung hierfür:
 - ◆ Die Elemente müssen nach irgendeinem Kriterium vergleichbar sein
 - ◆ z.B. $<$ oder $>$ für Zahlen etc.
 - ◆ nur Gleichheit bzw. Ungleichheit reichen nicht
 - ◆ möglich sind aber auch komplizierte Vergleichsoperationen für ADTs

- Dabei ist die Art der Vergleichsoperation selbst für den Sortieralgorithmus absolut unerheblich
- Es wurden sehr unterschiedliche Sortieralgorithmen entwickelt.
- Sie unterscheiden sich nicht nur in ihrer zentralen Idee sondern auch noch in der notwendigen Anzahl der
 - ◆ Vergleiche und
 - ◆ Vertauschungen von Elementen
- Selbst bei elementaren Datentypen, aber erst recht bei ADTs bestimmen die notwendige Anzahl der Vergleiche und Vertauschungen maßgeblich die Ausführungsgeschwindigkeit des Algorithmus

- Im realen Leben wird so häufig eine Spielkartenhand sortiert: es wird von links nach rechts eine sortierte Teilfolge gebildet durch Einsortieren bisher unsortierter Karten (der zweiten, bisher unsortierten Teilfolge)

Idee:

- In eine bereits sortierte Menge (zu Anfang ggf. leer) werden weitere Elemente sofort an der richtigen Position eingefügt
- Es werden Teilfelder sortiert, begonnen wird mit einem Teilfeld der Länge 2
- Es wird immer das letzte Element des Teilfeldes in die richtige Position gebracht

1. Genauer: Das letzte Element des Teilfeldes wird mit allen Elementen verglichen, die sich vor ihm im Feld befinden
2. Ist das letzte Element kleiner als sein Vorgänger, so rutscht dieser Vorgänger um eine Position nach hinten
3. So rutschen nach und nach alle Elemente des Feldes um eine Position nach hinten, die größer sind als das letzte Element
4. Wird ein Element gefunden, das kleiner ist als das letzte, dann wird das letzte Element an der Position hinter dem kleineren Element eingefügt (daher der Name)
5. Das Teilfeld wird um das nächste Element erweitert, Wiederholung der Schritte 3. – 6.
6. Ist das Teilfeld genauso groß wie das gesamte zu sortierende Feld terminiert der Algorithmus

Unsortiert:

54	80	11	91	17	23	58	28
----	----	----	----	----	----	----	----

1. Durchlauf:

Abgrenzung zur noch unsortierten Teilfolge

54	80	11	91	17	23	58	28
----	----	----	----	----	----	----	----

kein Tausch

2. Durchlauf:

54	80	11	91	17	23	58	28
----	----	----	----	----	----	----	----

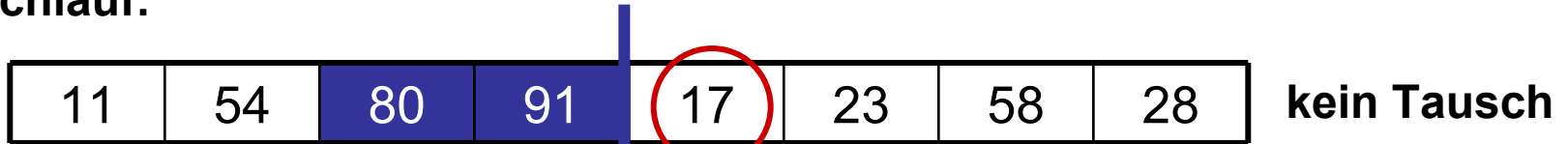
Tausch!

54	11	80	91	17	23	58	28
----	----	----	----	----	----	----	----

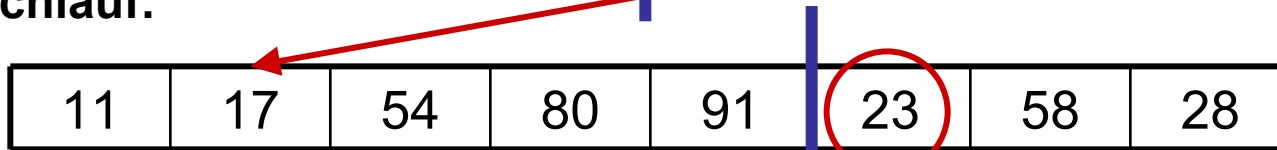
Tausch!

11	54	80	91	17	23	58	28
----	----	----	----	----	----	----	----

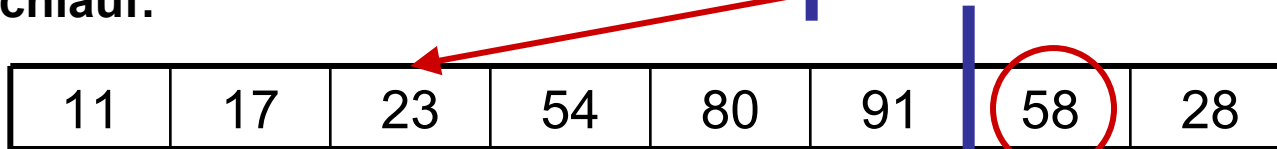
3. Durchlauf:



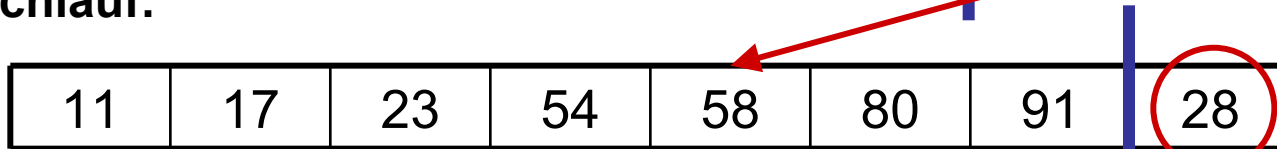
4. Durchlauf:



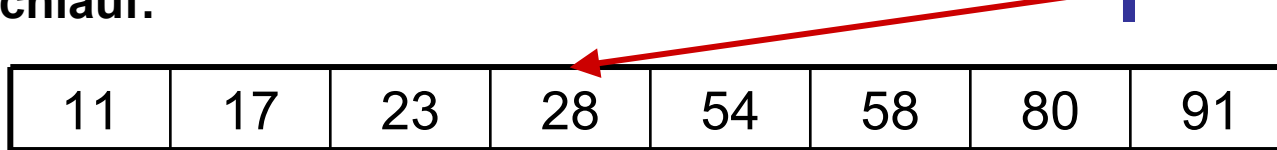
5. Durchlauf:



6. Durchlauf:



7. Durchlauf:



sortiertes Feld = gesamtes Feld \Rightarrow Fertig!

feld =
unsortiertes Feld
der Länge n

n = Länge des Feldes feld

wiederhole für alle i von 1 bis n-1

temp = feld [i]

j = i

solange j > 0 und feld[j - 1] > temp

feld[j] = feld[j - 1]

j = j - 1

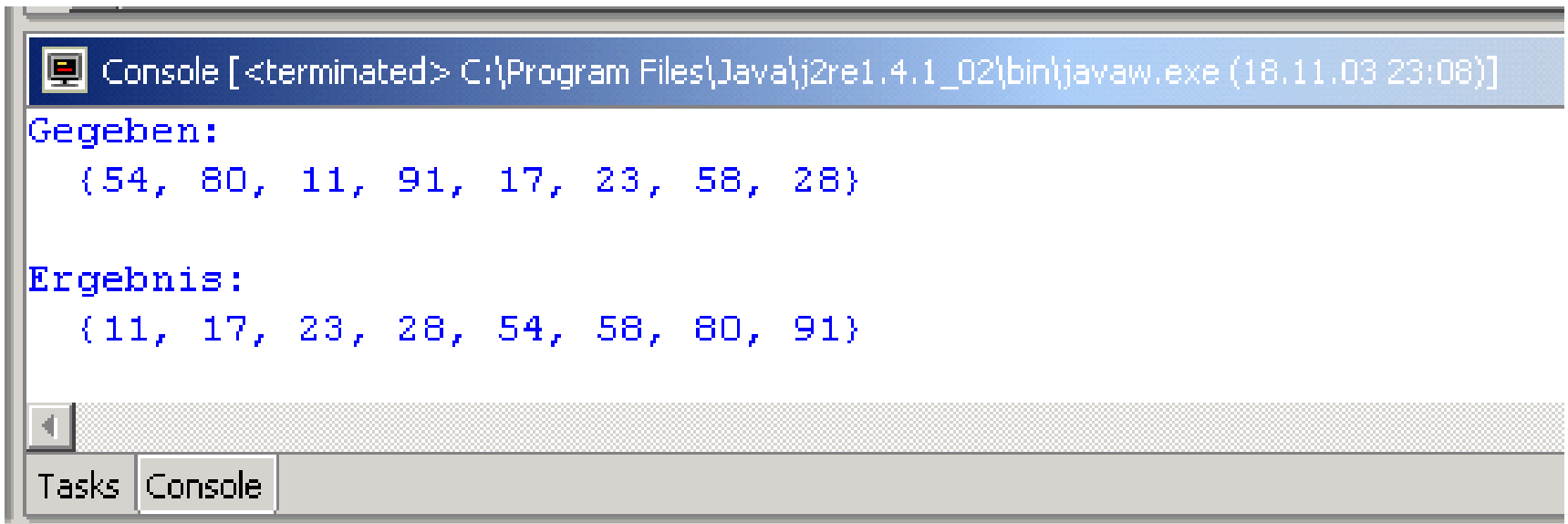
feld[j] = temp

```
public static void insertionSort(int feld[]) {  
    int n = feld.length;  
    int temp, j;  
  
    for (int i=1; i<n; i++) {  
        temp = feld[i]; // einzusortierendes Element merken  
        j = i;  
        // dieses Element an die richtige Position rutschen lassen  
        while ((j > 0) && (feld[j-1] > temp)) {  
            feld[j] = feld[j-1];  
            j--;  
        }  
        // einzusortierendes Element an richtige Position schreiben  
        feld[j] = temp;  
    }  
}
```

**Durchlaufen der zu
sortierenden Teilfolgen,
beginnend bei
2 Elementen**



```
public static void main(String[] args) {  
    int[] feld1 = { 54 , 80 , 11 , 91 , 17 , 23 , 58 , 28 };  
  
    ausgabeIntFeld("Gegeben:", feld1);  
    insertionSort(feld1);  
    ausgabeIntFeld("Ergebnis:", feld1);  
}
```



```
Console [<terminated> C:\Program Files\Java\j2re1.4.1_02\bin\javaw.exe (18.11.03 23:08)]  
Gegeben:  
    {54, 80, 11, 91, 17, 23, 58, 28}  
  
Ergebnis:  
    {11, 17, 23, 28, 54, 58, 80, 91}
```

- Der Name Bubble-Sort – Sortieren mit Blasen entspringt der bildlichen Vorstellung, dass die großen Blasen (= großen Elemente) wie Luftblasen in Sprudelwasser (= dem zu sortierenden Feld) nach oben steigen

Idee:

- Es werden jeweils zwei benachbarte Elemente verglichen. Wenn das linke Element größer ist als das rechte, werden sie vertauscht
- Begonnen wird der Vergleich mit dem Element 0 und 1. Dann wird mit dem Element 1 und 2 fortgesetzt, und danach werden die Elemente 2 und 3 verglichen bis an das Ende des Feldes

1. So wandert das größte Element an das Ende des Feldes
2. Die Schritte 1. – 3. werden solange wiederholt, bis keine Vertauschung von Elementen mehr stattgefunden hat
3. War keine Vertauschung mehr notwendig, dann ist das Feld sortiert und der Algorithmus terminiert

Unsortiert:

54	80	11	91	17	23	58	28
----	----	----	----	----	----	----	----

1. Durchlauf:

54	80	11	91	17	23	58	28
----	----	----	----	----	----	----	----

kein Tausch

54	80	11	91	17	23	58	28
----	----	----	----	----	----	----	----

Tausch!

54	11	80	91	17	23	58	28
----	----	----	----	----	----	----	----

54	11	80	91	17	23	58	28
----	----	----	----	----	----	----	----

kein Tausch

54	11	80	91	17	23	58	28
----	----	----	----	----	----	----	----

Tausch!

54	11	80	17	91	23	58	28
----	----	----	----	----	----	----	----

Tausch!

54	11	80	17	23	91	58	28
----	----	----	----	----	----	----	----

Tausch!

54	11	80	17	23	58	91	28
----	----	----	----	----	----	----	----

Tausch!

54	11	80	17	23	58	28	91
----	----	----	----	----	----	----	----

2. Durchlauf:

11	54	17	23	58	28	80	91
----	----	----	----	----	----	----	----

3. Durchlauf:

11	17	23	54	28	58	80	91
----	----	----	----	----	----	----	----

4. Durchlauf:

11	17	23	28	54	58	80	91
----	----	----	----	----	----	----	----

5. Durchlauf: keine Vertauschungen mehr, Feld ist sortiert und Algorithmus terminiert somit

Merke!



- Die Performance des Bubble-Sort Algorithmus kann noch verbessert werden
- Der verbesserte Bubble-Sort berücksichtigt die Tatsache, dass nach dem ersten Durchlauf das größte Element bereits an der richtigen Stelle steht, nach dem zweiten Durchlauf die beiden größten etc.
- Statt sich zu merken ob überhaupt noch getauscht wurde, laufen zwei ineinander geschachtelte Schleifen:
 - ◆ Die äußere Schleife durchläuft die "untere", bisher unsortierte Teilfolge, mit jedem Durchlauf wird die unsortierte Teilfolge um ein Element kürzer
 - ◆ Die innere Schleife vergleicht – wie beschrieben – jeweils zwei benachbarte Elemente

- Der Shell-Sort-Algorithmus ist eine Erweiterung des Insertion-Sort Algorithmus
- Benannt ist er nach seinem Entwickler D. L. Shell

Idee:

- Es werden Elemente vertauscht, die weiter voneinander entfernt stehen
- Das gesamte Feld wird als zusammengesetzt aus Teilfolgen betrachtet
- Diese Teilfolgen werden jeweils vorsortiert
- Die Elemente einer Teilfolge haben im gesamten zu sortierenden Feld jeweils einen festen Abstand voneinander, die Schrittweite

- Schrittweite = 1 entspricht dem Insertion Sort
- Die Idee ist Teilfolgen mit jeweils **großer** Schrittweite vorzusortieren
- Die Schrittweite wird nach jeder erfolgten Sortierung einer Teilfolge reduziert, bis sie den Wert 1 annimmt
- Durch die Vorsortierung werden die Elemente des Feldes schneller in die Nähe ihrer endgültigen Position gebracht
- Dadurch sind beim letzten Sortierdurchgang bei Schrittweite 1 nur noch Verschiebungen um wenige Positionen nötig

- Durch weniger Verschiebeoperationen ist die Geschwindigkeit des Shell-Sorts höher als die des Insertion-Sorts

Geeignete Schrittweiten:

- Von der Wahl der Schrittweite hängt maßgeblich die Performance von Shell-Sort ab
- Geeignete Schrittweiten ergeben sich durch die Formel:
$$h_{i+1} = 3 \cdot h_i + 1$$

Startwert $h_1 = 1 \rightarrow 1, 4, 13, 40, 121, 364, 1093, \dots$
- Man berechne h_i bis das Ergebnis die Hälfte der zu sortierenden Elemente erreicht hat und erhält so die rückwärts abzuarbeitende Liste der Schrittweiten

Unsortiert:

54	80	11	91	17	23	58	28
----	----	----	----	----	----	----	----

Feldgröße = 8

Schrittweiten: 1, 4, 13 (Stopp, da 13 größer 8, ein Schritt zurück)

1. Durchlauf, Schrittweite = 4:

54	80	11	91	17	23	58	28
----	----	----	----	----	----	----	----

Tausch!

17	80	11	91	54	23	58	28
----	----	----	----	----	----	----	----

Tausch!

17	23	11	91	54	80	58	28
----	----	----	----	----	----	----	----

kein Tausch

17	23	11	91	54	80	58	28
----	----	----	----	----	----	----	----

Tausch!

17	23	11	28	54	80	58	91
----	----	----	----	----	----	----	----

Ergebnis

Unsortiert:

54	80	11	91	17	23	58	28
----	----	----	----	----	----	----	----

Nach 1. Durchlauf mit Schrittweite 4:

17	23	11	28	54	80	58	91
----	----	----	----	----	----	----	----

- Feld ist offensichtlich schon erheblich "sortierter"
- Es liegen 4 sortierte Teilfolgen vor:
 $\{17, 54\}, \{23, 80\}, \{11, 58\}, \{28, 91\}$
- Im zweiten Durchlauf wird jetzt die Schrittweite auf 1 gesenkt (= Insertion-Sort, aber mit erheblich weniger notwendigen Verschiebungen!)
- Danach terminiert der Algorithmus, das Feld ist sortiert

feld =
unsortiertes Feld
der Länge n

```
h = 1; n = Länge des Feldes
```

```
h = 3 * h + 1
```

```
solange h < n
```

```
h = h / 3
```

```
wiederhole für alle i von h bis n-1
```

```
temp = feld[ i ]
```

```
j = i
```

```
solange j > h und feld[ j - h ] > temp
```

```
feld[ j ] = feld[ j - h ]
```

```
j = j - h
```

```
feld[ j ] = temp
```

```
solange h ungleich 1 ist
```

```
public static void shellSort(int[] feld) {
    int n = feld.length;    // Länge des Feldes
    int h = 1;              // Schrittweite
    int j, temp;           // für Elementetausch

    do {                   // Start-Schrittweite ermitteln
        h = 3 * h + 1;
    } while (h < n);
    do {
        // neue Schrittweite berechnen
        // beim ersten Mal: einen Schritt zurück
        h /= 3;
    } while (h > 1);
}
```



```
for (int i = h; i < n; i++) {
    // Teilfolge sortieren
    temp = feld[i];
    j = i;
    // notwendige Verschiebungen durchführen
    while ((j >= h) && (feld[j-h] > temp)) {
        feld[j] = feld[j-h];
        j -= h;
    }
    feld[j] = temp;
}
} while (h != 1);    // bei Schrittweite 1 fertig
}
```

- Quick-Sort ist der wohl am häufigsten eingesetzte Sortieralgorithmus
- Er arbeitet nach dem Prinzip Divide and Conquer
- und ist daher in der Regel rekursiv programmiert

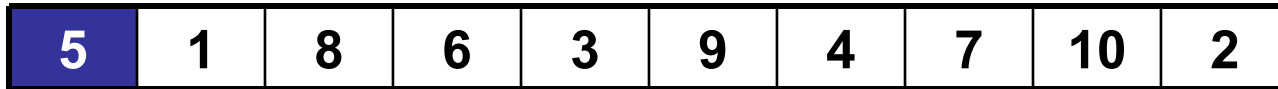
Idee:

- Ein zu sortierendes Feld wird in zwei Teilfelder zerlegt, die vorsortiert sind
- Dazu wird ein Element als Vergleichselement genutzt, das an die richtige Stelle im Feld verschoben wird
- Alle Elemente, die kleiner sind als dieses Element, werden in das linke Teilfeld verschoben

- Die größeren Elemente kommen in das rechte Teilfeld
- Das einsortierte Element steht nun an der richtigen Stelle und braucht nicht mehr beachtet zu werden
- Die beiden Teilfelder links und rechts von diesem Element werden dann nach dem gleichen Prinzip weiterverarbeitet wie das gesamte zu sortierende Feld
- Das Zerlegen des Feldes ist der wesentlich kompliziertere Teil des Algorithmus
- Betrachten wir daher erstmal den Quick-Sort Rumpf

Beispiel – 1

Unsortiert:



Vergleichselement an richtige Position und kleinere Elemente nach links, größere nach rechts



↑ Position ok

Teilfeld mit größeren Elementen

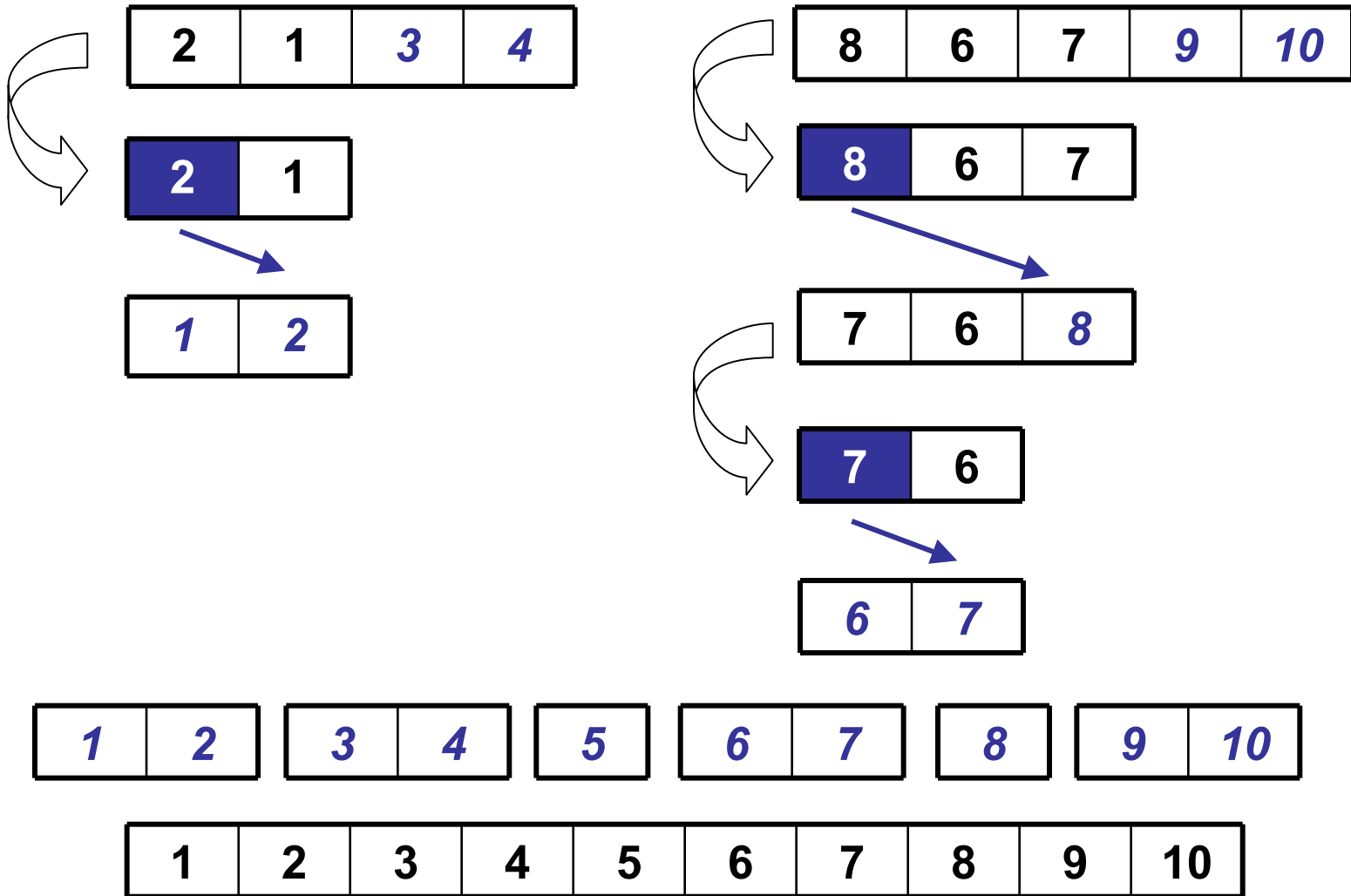
Teilfeld mit kleineren Elementen



↑ ↑
Position ok

↑ ↑
Position ok

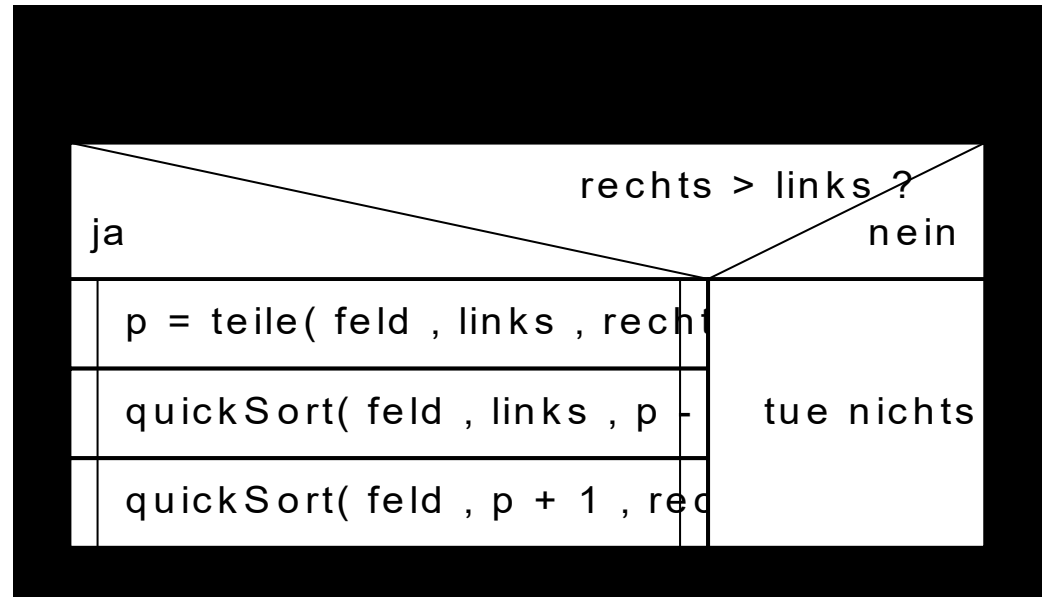
Beispiel – 2



feld =
unsortiertes Feld der Länge n

links =
linker Rand des (Teil-)Feldes

rechts =
rechter Rand des (Teil-)Feldes



```
public static void quickSort(int[] feld, int links, int rechts) {  
    if (rechts > links) {  
        int p = quickSortTeile(feld, links, rechts);  
        quickSort(feld, links, p-1);  
        quickSort(feld, p+1, rechts);  
    }  
}
```

Das Zerlegen des Feldes:

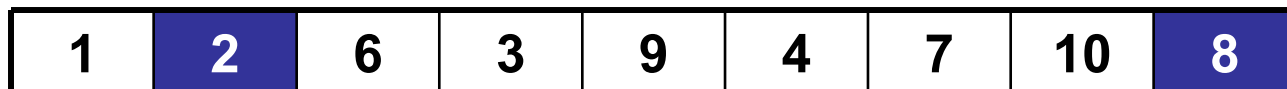
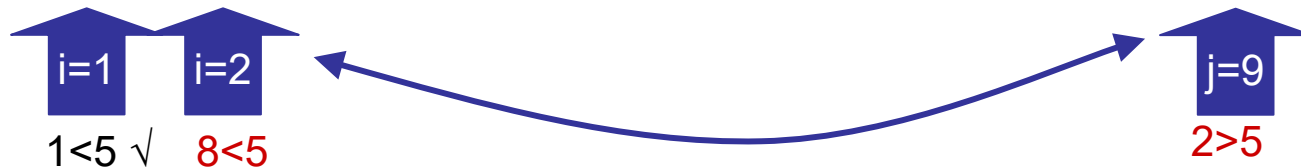
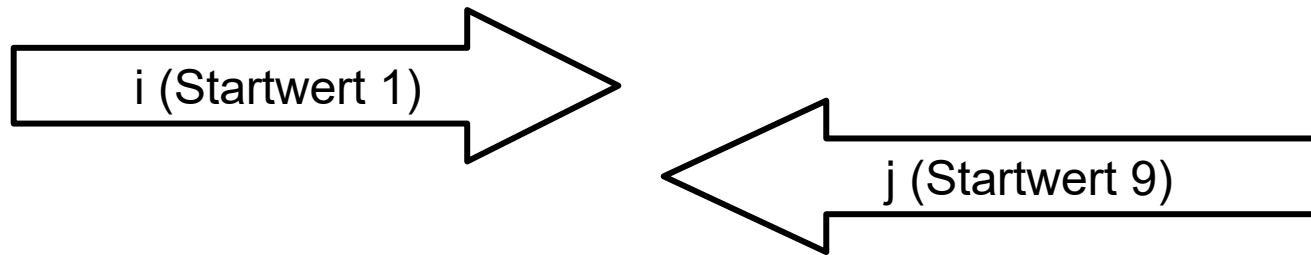
- Das erste Element des Feldes wird als das Element angenommen, welches an die richtige Position gebracht werden soll
- Bevor das geschehen kann, müssen die anderen Elemente entsprechend umgeordnet werden
- Dazu werden zwei Zeiger auf die Elemente verwendet: Der Zeiger i läuft vom ersten zum letzten Element, der Zeiger j vom letzten zum ersten
- Mit Hilfe des Zeigers i werden die Elemente so lange untersucht, bis eines gefunden wird, das größer ist als das Teilungselement

1. Mit Hilfe des Zeigers j werden von rechts beginnend die Elemente durchsucht, bis eines gefunden wird, das kleiner ist als das Teilungselement
2. Das i -te und das j -te Element werden vertauscht
3. Die Punkte 4. – 6. werden so lange wiederholt ausgeführt, bis der Zeiger i größer oder gleich dem Zeiger j ist. Dann ist die Position gefunden, an der das Teilungselement eingefügt werden muss – nämlich an der Position, auf die der Zeiger j verweist
4. Dazu tauschen das j -te und das erste Element die Plätze
5. Es entstehen zwei Teilfelder. Ein Teilfeld, welches die kleineren Elemente enthält, und eines, das die größeren Werte enthält

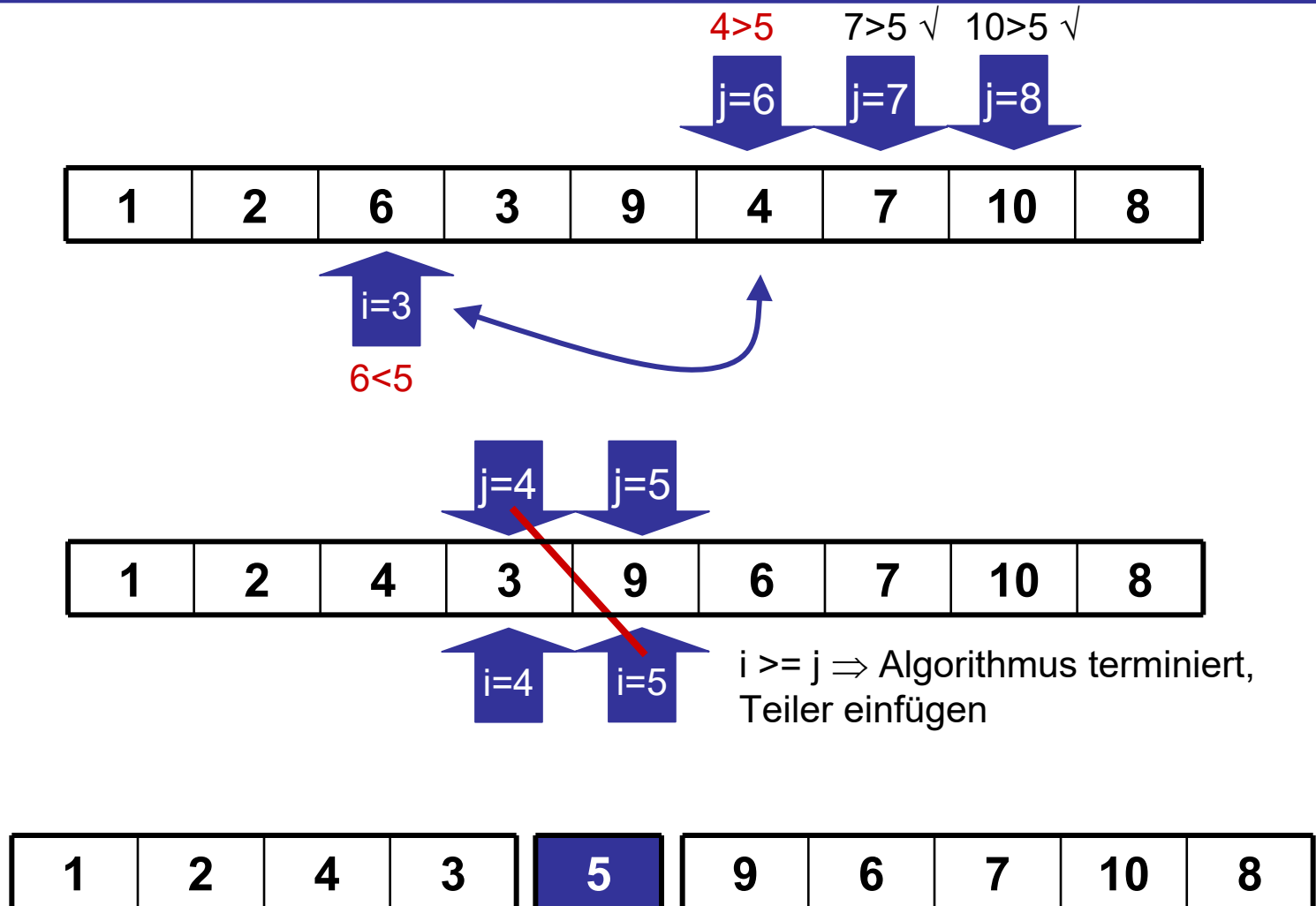
Unsortiert:



↑ Vergleichselement, Teiler



Beispiel – 2

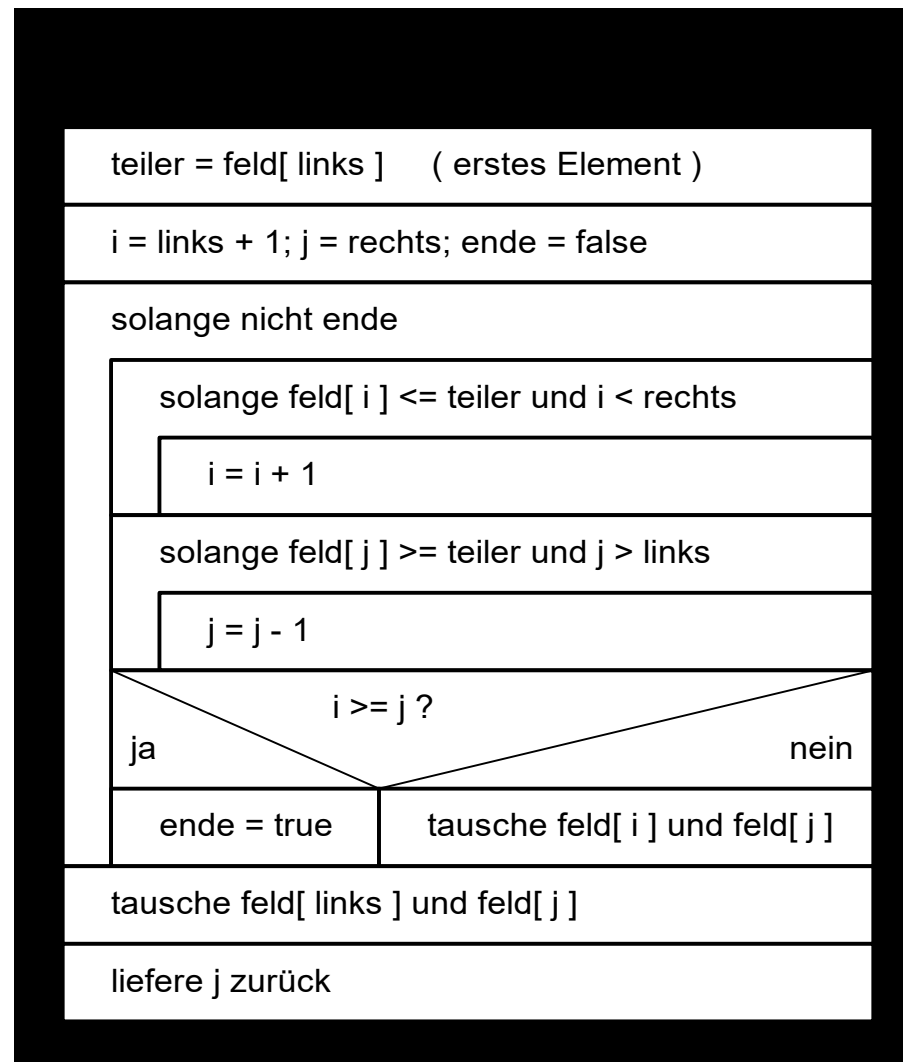


feld =
unsortiertes Feld der
Länge n

links =
linker Rand des
(Teil-)Feldes

rechts =
rechter Rand des
(Teil-)Feldes

Rückgabe:
 $j =$
Position des
einsortierten
Elements



```
public static int quickSortTeile(int[] feld, int links, int rechts) {
    int teiler = feld[links]; // Referenzelement
    int i = links + 1;        // Intervallgrenzen
    int j = rechts;
    int temp;                 // für Elementetausch
    boolean ende = false;

    while (!ende) {
        // terminiert, wenn Element größer als Teiler gefunden
        // oder obere Grenze erreicht
        while ((feld[i] <= teiler) && (i < rechts)) {
            i++;
        }
        // terminiert, wenn Element kleiner als Teiler gefunden
        // oder untere Grenze erreicht
        while ((feld[j] >= teiler) && (j > links)) {
            j--;
        }
    }
}
```

```
}  
// Richtige Position für Teiler gefunden?  
if (i >= j)  
    ende = true;  
// Ansonsten Elementetausch  
else {  
    temp = feld[i];  
    feld[i] = feld[j];  
    feld[j] = temp;  
}  
}  
temp = feld[j];  
feld[j] = feld[links];  
feld[links] = temp;  
// Rückgabewert ist die Position an der das Teilerelement  
// einsortiert wurde  
return (j);  
}
```

Vergleich der Verfahren

Sortieralgorithmus	Anzahl der Vergleiche im \emptyset	Beispiel für 1000 Elemente
Insertion-Sort	$n^2 / 4$	250.000
Bubble-Sort	$n^2 / 2$	500.000
Shell-Sort ($h_{i+1}=3*h_i+1$)	$n^{3/2}$	31.623
Quick-Sort	$2n * \ln(n)$	13.816

- Es ist leicht erkennbar, dass Quick-Sort der mit Abstand schnellste Algorithmus (von den in dieser Vorlesung betrachteten) ist
- Trotz Rekursion – ist daher, wo immer möglich, Quick-Sort der Vorzug vor anderen Verfahren zu geben
- Der Abstand wird steigender Anzahl der Elemente noch deutlicher

??? Frage



***Welche Fragen
gibt es?***

JETZT!



Suchen

Zweiter Anlauf

- Ab jetzt dürfen wir annehmen, dass unser Suchfeld sortiert vorliegt
- Kann man das intuitive Vorgehen beim Zahlenraten: Zuerst die Mitte nehmen, Intervall verkleinern, wieder die Mitte nehmen ... auf einen Suchalgorithmus verallgemeinern?

Idee:

- Es wird zuerst das mittlere Element in der sortierten Datenmenge untersucht. Ist es größer als das gesuchte Element, muss nur noch in der unteren Hälfte gesucht werden, anderenfalls in der oberen
- **Mitte** = bezogen auf die Länge des Suchfeldes, **nicht** auf die Spannbreite der Werte im Suchfeld

- Nun wird die Suche auf die neue halbierte Datenmenge angewendet. Das mittlere Element wird mit dem gesuchten Element verglichen
- Ist das mittlere Element größer, wird in der unteren Hälfte weitergesucht, sonst in der oberen und so fort
- Somit halbiert sich bei jedem Schritt die Anzahl der Elemente, die noch untersucht werden müssen
- Der Algorithmus terminiert,
 - ◆ wenn das Element gefunden wurde,
 - ◆ oder nur noch ein Element übrig bleibt – entweder ist dies das gesuchte Element, oder das Suchfeld enthält das gesuchte Element nicht

Gegeben:

54	80	11	91	17	23	58	28
----	----	----	----	----	----	----	----

Sortiert, gesucht 58:

11	17	23	28	54	58	80	91
----	----	----	----	----	----	----	----

Mitte = $(0 + 7) / 2 = 3$



$28 < 58 \Rightarrow$ rechts weitersuchen, im Intervall von 4 bis 7

Mitte = $(4 + 7) / 2 = 5$



$58 == 58 \Rightarrow$ gefunden, Rückgabewert 5

Es wurden nur 2 Vergleiche benötigt,
statt 7 bei der linearen Suche

Sortiert, gesucht 27:

11	17	23	28	54	58	80	91
----	----	----	----	----	----	----	----

$$\text{Mitte} = (0 + 7) / 2 = 3$$



$28 > 27 \Rightarrow$ links weitersuchen, im Intervall von 0 bis 2

$$\text{Mitte} = (0 + 2) / 2 = 1$$



$17 < 27 \Rightarrow$ rechts weitersuchen, im Intervall von 2 bis 2

$$\text{Mitte} = (2 + 2) / 2 = 2$$

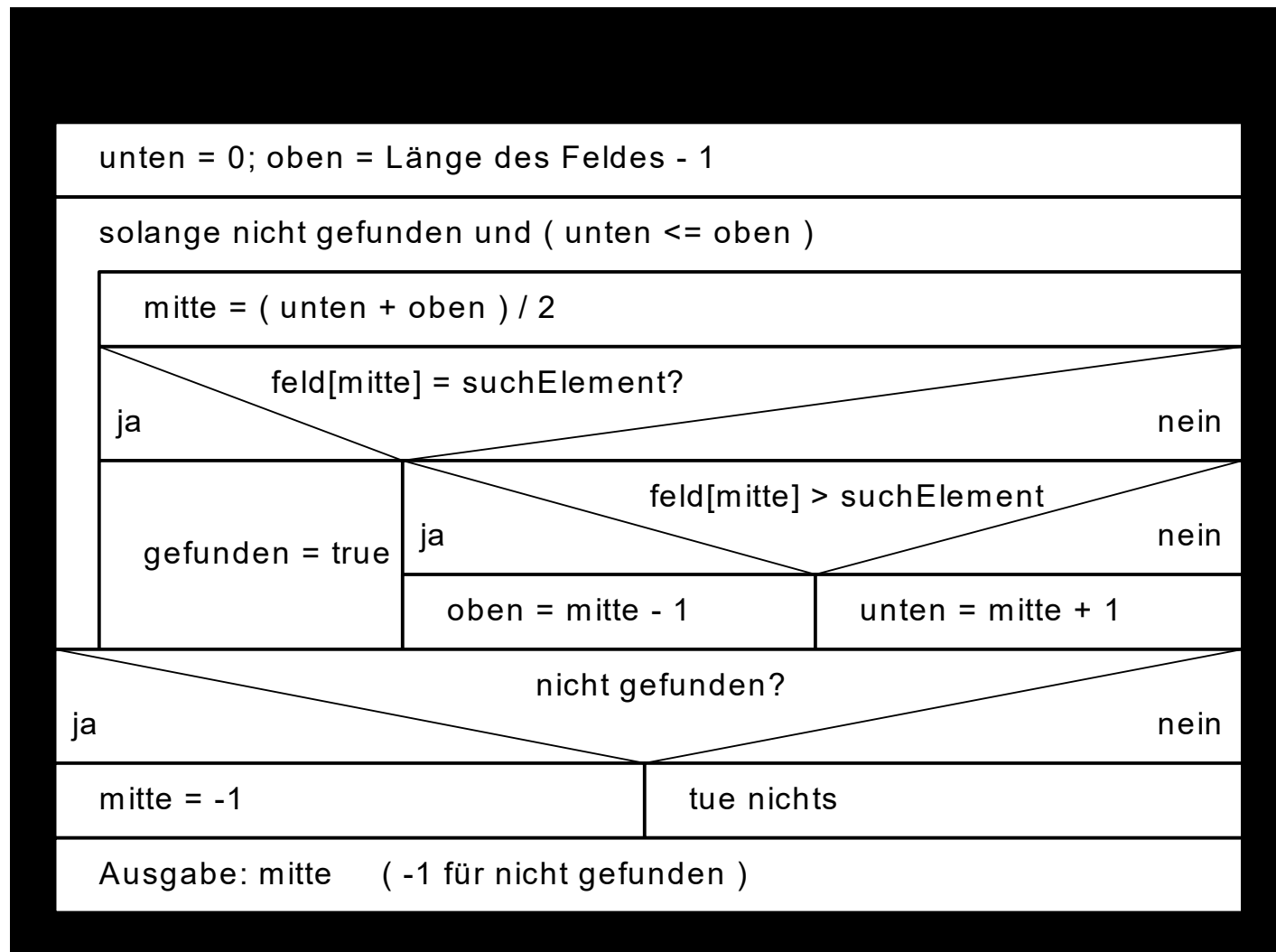


$23 < 27 \Rightarrow$ rechts weitersuchen, geht nicht, da letztes Element, nicht gefunden, Rückgabewert -1

Es wurden nur 3 Vergleiche benötigt, statt 8 bei der linearen Suche

feld =
sortiertes Feld der
Länge n

suchElement =
Element, dessen
Position bestimmt
werden soll



```
public static int binSuch(int[] feld, int suchElement) {
    int mitte = 0, unten = 0;           // Variablen für Feldgrenzen
    int oben = feld.length - 1;
    boolean gefunden = false;         // Indikator

    while (!gefunden && (unten <= oben)) {
        mitte = (unten + oben) / 2; // Mitte des Suchfelds
        if (feld[mitte] == suchElement)
            gefunden = true;
        else // Ansonsten Suchintervall halbieren
            if (feld[mitte] > suchElement)
                oben = mitte - 1;
            else
                unten = mitte + 1;
    }
    if (!gefunden) // Nicht gefunden?
        mitte = -1;
    return (mitte);
}
```

```
int[] feld1 = { 54 , 80 , 11 , 91 , 17 , 23 , 58 , 28 }
```

Aufruf:

```
Vorher:    insertionSort(feld1);    // oder anderer  
                                                    // Sortieralgorithmus!
```

```
■ System.out.println(binSuch(feld1, 58));
```

→ Ausgabe: 6

```
■ System.out.println(binSuch(feld1, 91));
```

→ Ausgabe: 8

```
■ System.out.println(binSuch(feld1, 27));
```

→ Ausgabe: -1

Merke!



- Für die Elementsuche in sortierten Feldern ist die binäre Suche der linearen Suche unbedingt vorzuziehen, da die binäre Suche wesentlich weniger Vergleiche benötigt
- Bei unsortierten Feldern muss bei der Aufwandsschätzung der Zeitbedarf für den Sortieralgorithmus unbedingt mitberücksichtigt werden
- Allgemein gilt: Umso häufiger gesucht wird, umso weniger fällt ein Sortiervorgang ins Gewicht und die binäre Suche ist vorzuziehen

Merke!



- Ungünstig wäre der Fall, dass sich vor jedem Suchvorgang die Daten ändern bzw. ändern könnten (z.B. bei einer Adressverwaltung)
- Wenn die Daten dann **nicht sortiert eingefügt** werden, wäre vor jedem Suchvorgang ein erneutes Sortieren unabdingbar ...
- ... und damit vom Aufwand her ggf. die lineare Suche vorzuziehen

Interpolationssuche:

- Voraussetzung: Sortiertes Feld
- Neben dem Elemente-Vergleich wird auch Indexrechnung zu gelassen

Beispiel:

- In einer 50 Elemente fassenden Liste werden die Zahlen zwischen 1 und 100 gespeichert
- Gefragt: Ist 30 in der Liste enthalten?
- Vorgehen: Anders als bei der binären Suche, wird man mit den Vergleichen nicht beim mittleren Element (Index 25), sondern (eine ungefähre Gleichverteilung annehmend) beim

$$(30 / 100) * 50 = 15. \text{ Element beginnen}$$

Hashsuche:

- Der Begriff des Sortierens wird erweitert
- → Der Index (und damit der Speicherort) wird aus dem Elementinhalt selbst mittels einer Hash-Funktion berechnet

Beispiel:

- Es sollen Zeichenketten gespeichert (und gesucht) werden
- Zur Verfügung steht ein Array of Strings mit 52 Elementen
- Man ordne den Buchstaben A bis Z die Werte 1 bis 26 zu
- Die Hash-Funktion $h : \text{String} \rightarrow \text{Integer}$ summiert die entsprechenden Werte der ersten beiden Buchstaben minus 1:
 - ◆ $h(\text{"AB"}) = 2$
 - ◆ $h(\text{"MONTAG"}) = 27$
 - ◆ $h(\text{"MITTWOCH"}) = 21$

Vorteil 👍

- Das "Suchen" reduziert sich auf ein Berechnen des Index

Nachteil 👎

- $h(\text{"SONNTAG"}) == h(\text{"SONNABEND"}) == 33$
- Liefert die Hash-Funktion für zwei unterschiedliche Elementwerte den gleichen Hash-Wert, dann spricht man von einer Kollision
 - ◆ Entweder die Hash-Funktion besser wählen, oder
 - ◆ Kollisionsstrategien (z.B. Index ist bereits vergeben \Rightarrow neuer Versuch mit diesem Index + 1 modulo 52)
- Diese Kollisionsstrategien müssen dann aber auch wieder beim Suchen berücksichtigt werden

??? **Fragen**



***Welche Fragen
gibt es?***

Merke!



- Der komplette Quellcode der Klasse **SuchenSortieren.java** mit den Implementierungen von
 - ◆ Linearer Suche
 - ◆ Binärer Sucheund
 - ◆ Insertion-Sort
 - ◆ Shell-Sort
 - ◆ Quick-Sort

steht auf meiner Web-Site unter INF / „Quellen“ für das **Selbststudium** und für **eigene Experimente** zum Download zur Verfügung!

JETZT



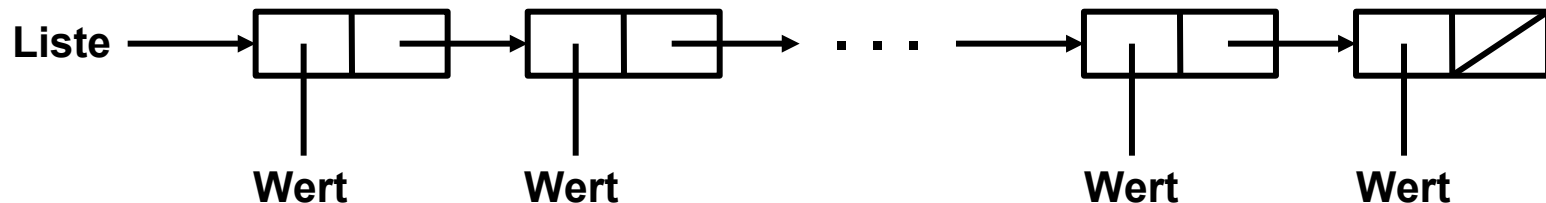
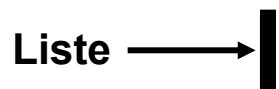
Dynamische Datenstrukturen

- Verkettete Listen sind dynamische Datenstrukturen,
- die aus einzelnen Elementen (Knoten) bestehen
- Jeder Knoten enthält Daten und einen oder mehrere Verweise auf weitere Knoten
- Verkettete Listen sind Arrays ähnlich, **ABER**
 - ◆ Vorteil von dynamischen Datenstrukturen:
Man muss sich "vorher" keine Gedanken darüber machen, wie viele Knoten gespeichert werden sollen
 - ◆ Nachteil von dynamischen Datenstrukturen:
Die Verwaltung, das Ansteuern der Elemente sind schwieriger
- Es gibt bei verketteten Listen keine vorgeschriebene Lese- und Schreibrichtung

Einfach verkettete Listen

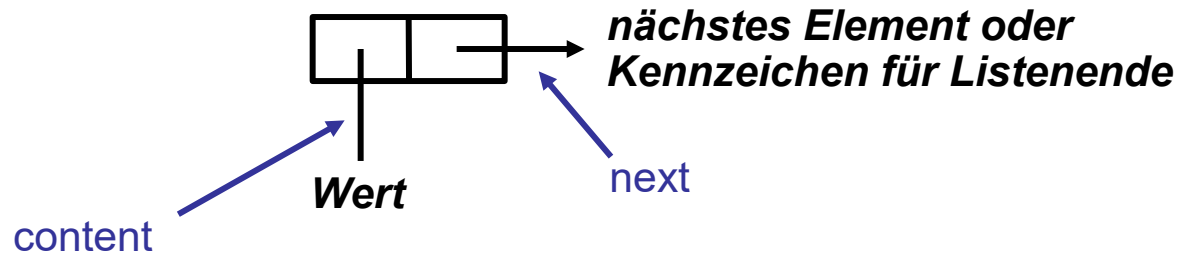
- Verkettete Listen eignen sich gut als didaktisches Beispiel... Sie sind aber auch nützlich!
- Mathematische Definition von Listen:
 - ◆ Eine Liste ist entweder die leere Liste oder
 - ◆ sie ist durch Einfügen eines neuen Elements vorne in eine existierende Liste entstanden.

Visualisierung:



Listendarstellung in Java – 1

- Ein Listenelement kann wie folgt dargestellt werden

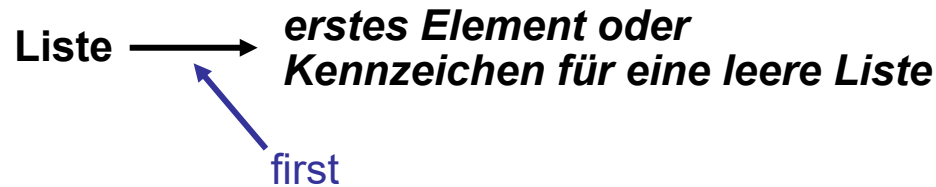


- In Java kann ein solches Element als Klasse mit zwei Attributen wie folgt dargestellt werden

```
class ListElement {  
    <Wertetyp> content;  
    ListElement next;  
}
```

**content, next dürfen gerne
aber auch deutsche
Bezeichner erhalten**

- Der Listenkopf kann wie folgt dargestellt werden



- In Java kann der Listenkopf als Klasse mit einem Attribut folgt dargestellt werden:

```
class LinearList {  
    ListElement first;  
}
```

- Kennzeichen für eine leere Liste ist der Wert null (Schlüsselwort)
first = null;

- Der Typ der Werte, der von der Liste verwaltet wird, ist offen gehalten und mit *<Wertetyp>* bezeichnet
Für die Implementierung in Java muss natürlich der Name eines bekannten Typs, also eines einfachen Type oder einer Klasse, eingesetzt werden
- Insgesamt sind drei Typen an der Konstruktion beteiligt
 - ◆ *LinkedList*
Der Listenkopf ist der ‚Aufhänger‘ für die Liste, er repräsentiert die Liste: daher der Name der Klasse
 - ◆ *ListElement*
Die Listenelemente bilden eine Art Verwaltungsstruktur für die Inhalte, die in der Liste gespeichert werden; sie sind nicht etwa diese Inhalte!
 - ◆ *<Wertetyp>*
Das sind die eigentlichen Werte, die in der Liste gespeichert werden
(zum Beispiel Adressen, Klausurergebnisse etc.)

Die folgenden Operationen sind auf linearen Listen sinnvoll:

- **addFirst**
fügt ein neues Element am Beginn einer Liste ein
- **removeFirst**
löscht das erste Element der Liste
- **getFirst**
liefert den Inhalt des ersten Elements der Liste
- **getLast**
liefert den Inhalt des letzten Elements der Liste
- **size**
liefert die Länge der Liste
- **isEmpty**
liefert true genau dann, wenn die Liste leer ist

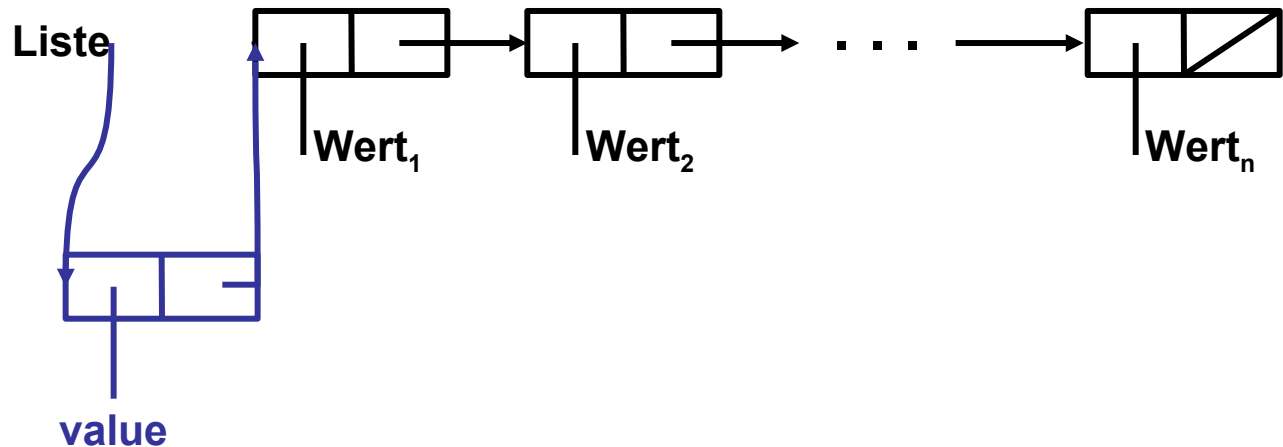
Unter Benutzung dieser Typen kann die Signatur der Operationen wie folgt angegeben werden:

- ◆ `addFirst: LinearList × <WerteTyp>`
fügt ein neues Element am Beginn einer Liste ein
- ◆ `removeFirst: LinearList`
löscht das erste Element der Liste
- ◆ `getFirst: LinearList → <WerteTyp>`
liefert den Inhalt des ersten Elements der Liste
- ◆ `getLast: LinearList → <WerteTyp>`
liefert den Inhalt des letzten Elements der Liste
- ◆ `size: LinearList → Integer`
liefert die Länge der Liste
- ◆ `isEmpty: LinearList → Boolean`
liefert `true` genau dann, wenn die Liste leer ist

- Alle aufgelisteten Operationen arbeiten auf linearen Listen und erhalten als erstes Argument einen Wert vom Typ `LinearList`
- Aus diesem Grund werden sie der Klasse `LinearList` zugeordnet und als deren so genannte (Instanz-) Methoden realisiert
- Beim Aufruf werden sie dann über ein Objekt der Klasse `LinearList` aufgerufen, auf das sie sich dann beziehen, anstatt ein solches Objekt als ersten Parameter zu akzeptieren
- Auf den nächsten Folien wird eine mögliche Definition der Operationen in Java vorgestellt

Implementierung Lineare Liste – 1

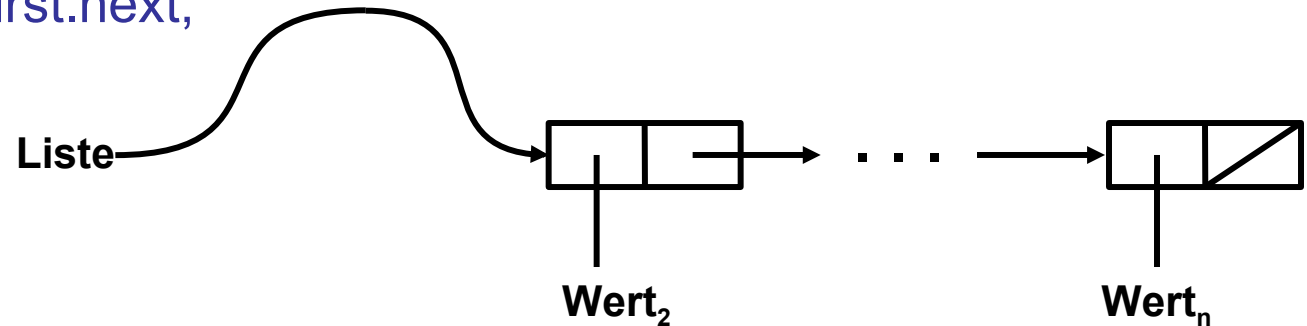
```
■ addFirst: LinearList × <WerteTyp>  
public void addFirst ( <WerteTyp> value ) {  
    ListElement n = new ListElement();  
    n.content = value;  
    n.next = first;  
    first = n;  
}
```



Implementierung Lineare Liste – 2

- removeFirst: LinearList → LinearList

```
public void removeFirst () {  
    if (first != null)  
        first = first.next;  
}
```



Dieses Objekt ist nicht mehr erreichbar, der Garbage Collector übernimmt die Speicherfreigabe

Vorsicht *FALLE!*



- Die automatische Speicherfreigabe nicht mehr referenzierter (= nicht mehr benötigter) Objekte ist spezielles Feature der Programmiersprache Java
- In nahezu **ALLEN** anderen Programmiersprachen (etwa C, C++, Pascal, Basic etc.) müssen dynamisch angelegte Speicherbereiche durch den Programmierer mittels eines Kommandos wie z.B.
 `free(first);`
EXPLIZIT freigegeben werden!
- Ansonsten entstehen **MEMORY LEAKS** (Speicherlöcher), also Speicherbereiche, die bis zum nächsten Aus-/Einschalten des Rechners dem System nicht mehr zur Verfügung stehen!

Implementierung Lineare Liste – 3

Die Methodenköpfe der weiteren Operationen sehen wie folgt aus:

- ◆ `getFirst: LinearList → <WerteTyp>`
`public <WerteTyp> getFirst () {`
 ...
`}`
- ◆ `getLast: LinearList → <WerteTyp>`
`public <WerteTyp> getLast () {`
 ...
`}`
- ◆ `length: LinearList → Integer`
`public int length () {`
 ...
`}`
- ◆ `isEmpty: LinearList → Boolean`
`public boolean isEmpty () {`
 ...
`}`

Die Implementierung ist jeweils relativ einfach

```
public class ListElement {  
    int content;  
    ListElement next;  
}
```

```
public class LinearList {  
    // Referenz auf das erste Element  
    ListElement first;  
  
    public void addFirst(int value) {  
        // fügt am Kopf der Liste ein neues Element ein  
        ListElement n = new ListElement();  
  
        n.content = value;  
        n.next = first;  
        first = n;  
    }  
}
```

LinearList.java – 1

```
public void output() {  
    // Ausgabe des Inhalts der Liste auf der Konsole  
    ListElement n = first;  
  
    System.out.print("\n( ");  
    while (n != null) {  
        System.out.print(n.content + " ");  
        n = n.next;  
    }  
    System.out.println(")");  
}
```

- Die Quellcodes stehen zum **Selbststudium** und für **eigene Experimente** auf meiner Web-Site unter GDI / Übungen zum Download zur Verfügung
- Benutzen Sie ListElement.java und LinkedList.java als **Ausgangspunkt** für die Lösung der 4. Pflichtübung!

```
public class LinListTest {  
  
    public static void main(String[] args) {  
        LinearList testList = new LinearList();  
  
        testList.addFirst(3);  
        testList.addFirst(4);  
        testList.addFirst(5);  
        testList.output();  
    }  
}
```

■ Ausgabe:

(5 4 3)

Implementierung Lineare Liste – 4

- Es ist günstig, einen neuen Konstruktor für die Klasse 'ListElement' zu definieren, der den Wert und das Folgeelement eines Knotens als Parameter akzeptiert und an die eigenen Felder zuweist

```
ListElement ( <WerteTyp> value, ListElement nextNode ) {  
    content = value;  
    next = nextNode;  
}
```

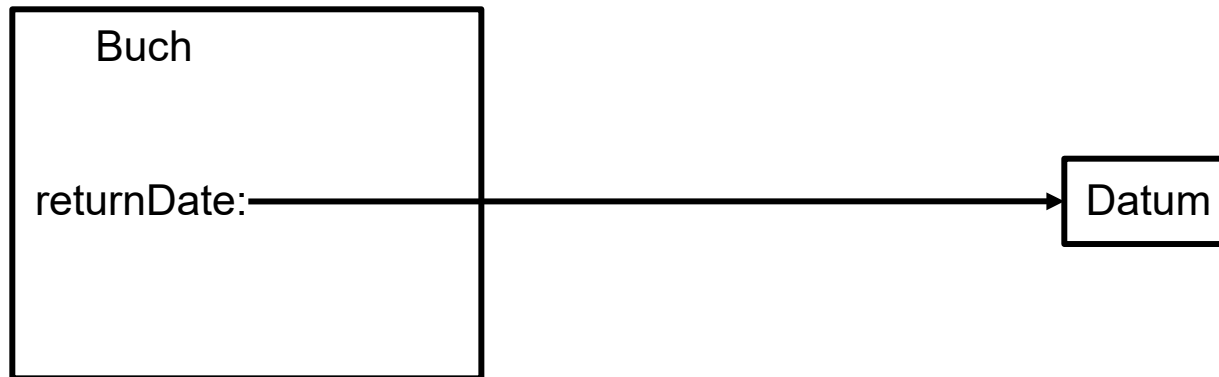
- Damit kann zum Beispiel die Methode addFirst vereinfacht werden

```
void addFirst ( <WerteTyp> value ) {  
    ListElement n = new ListElement( value, first );  
    first = n;  
}
```

Merke!

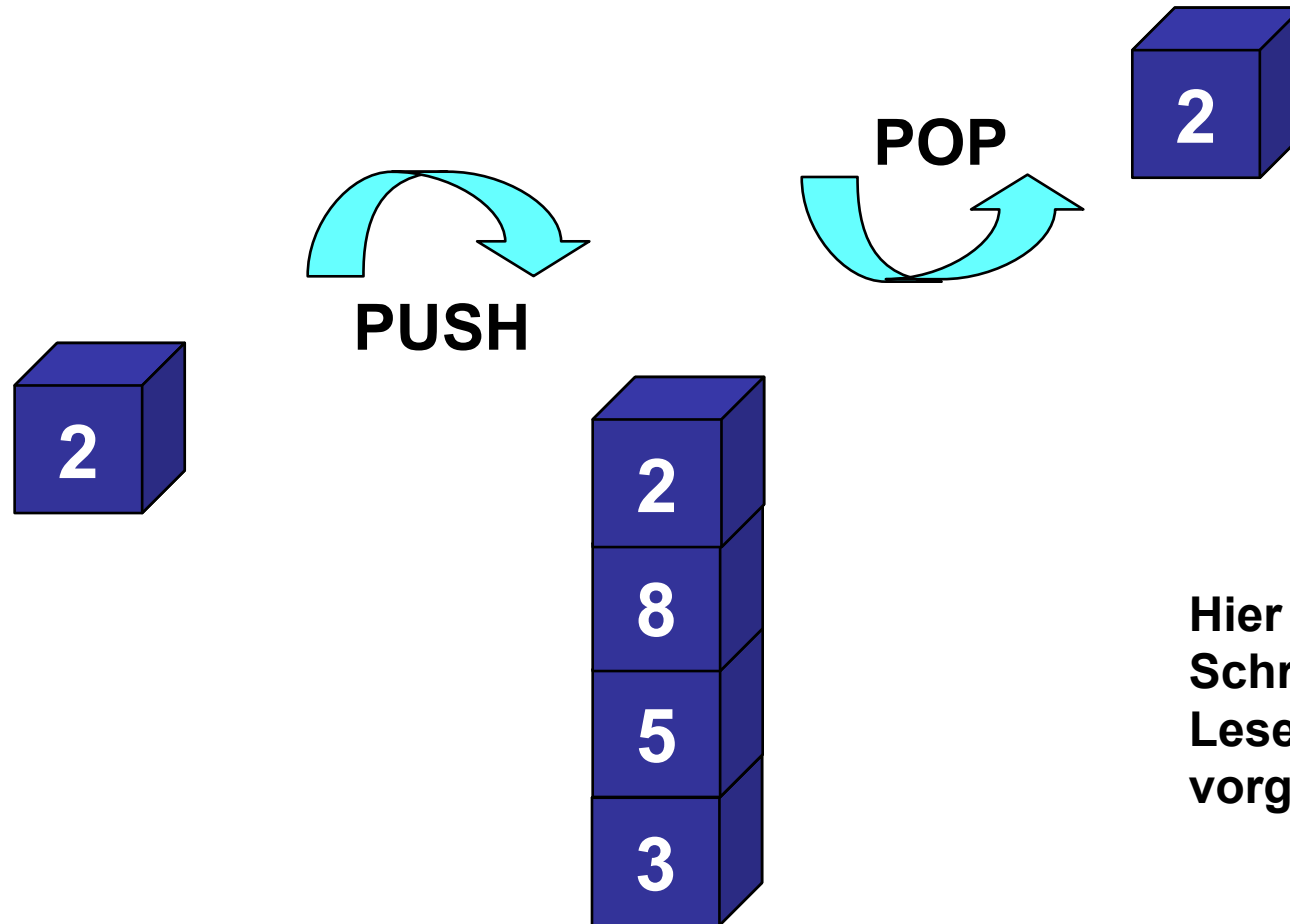


- In der Vorlesung über Klassen und Objekte haben wir eine solche Struktur als „geschachtelte Objekte“ bezeichnet
- Im Beispiel dort hatte ein Buch-Objekt ein Datums-Objekt eingeschachtelt
- Während dieses Bild für das Buch gut vorstellbar ist, passt das für die linearen Listen nicht besonders gut



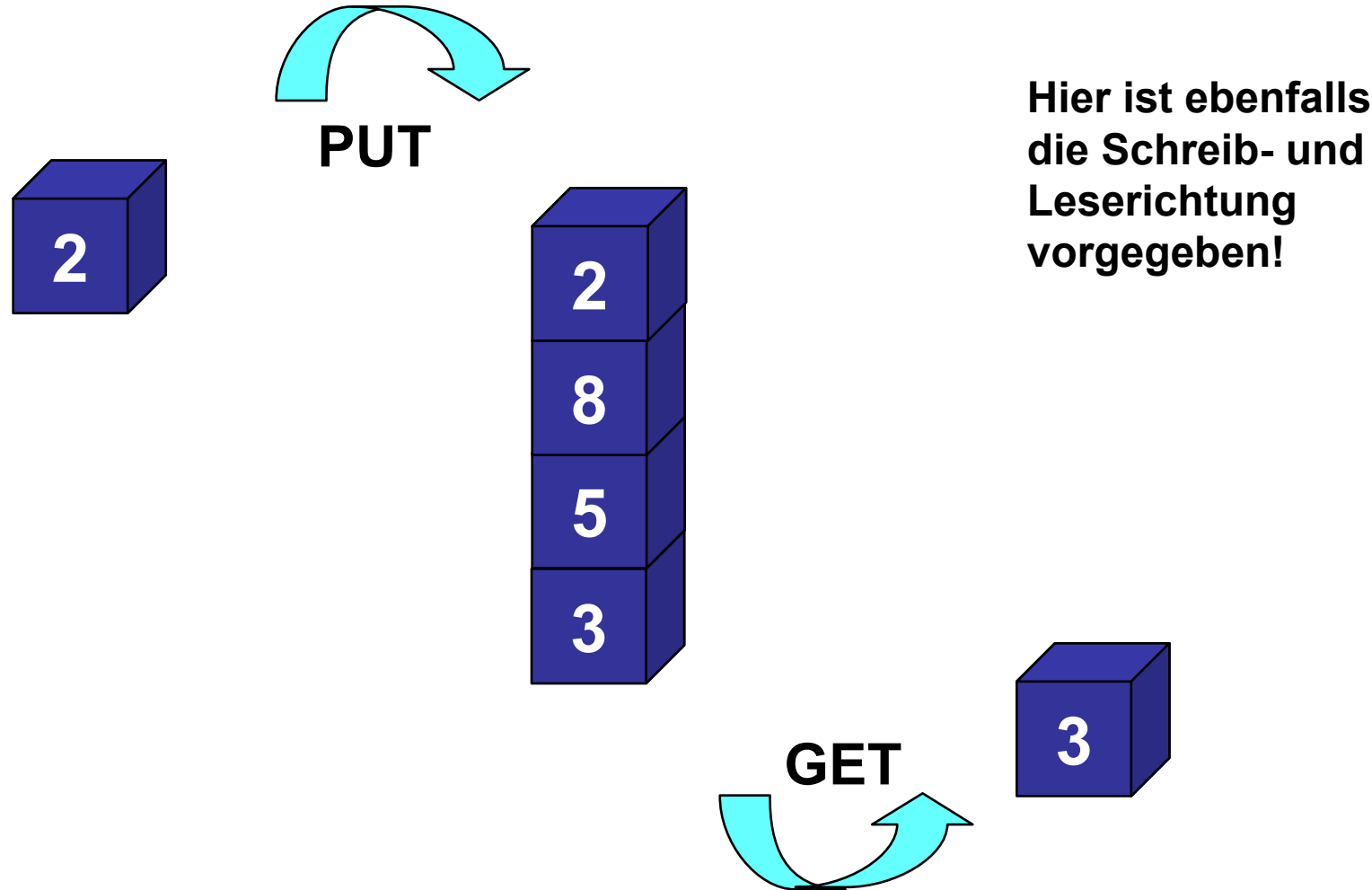
- Die eigentliche Struktur entspricht aber derjenigen der linearen Listen

Der Stapel (Keller, Stack)



Hier ist dann die
Schreib- und
Leserichtung
vorgegeben!

Die Schlange (Queue)



- Die beiden Datenstrukturen werden in der Praxis häufig eingesetzt

Stapel:

- Wir haben bereits den Stapel kennen gelernt: als Speicherorganisation für die lokalen Variablen einer Methode
- Weitere Einsatzmöglichkeiten für den Stapel:
 - ◆ UPN-Taschenrechner
 - ◆ rekursive Auswertung von Ausdrücken
- Der Stapel folgt dem sog. Last In – First Out (LIFO) Prinzip

■ Operationen:

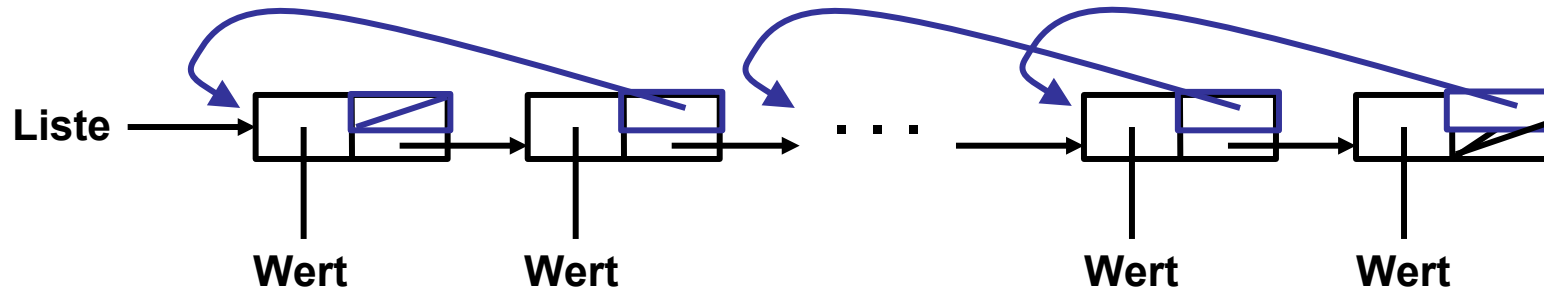
- ◆ **push**: $\text{Stack} \times \langle \text{Wertetyp} \rangle$
Ablegen eines Elementes auf der Spitze des Stapels
- ◆ **pop**: $\text{Stack} \rightarrow \langle \text{Wertetyp} \rangle$
Herunternehmen eines Elementes von der Spitze des Stapels
- ◆ **isEmpty**: $\text{Stack} \rightarrow \text{boolean}$
Test, ob Stapel leer

Schlange:

- Im Gegensatz zum Stapel folgt die Schlange dem First In – First Out (FIFO) Prinzip

- Typische Einsatzgebiete für Schlangen:
 - ◆ Warteschlangen (z.B. die Druck-Queue)
 - ◆ Ereignis-Auswertungen
- Operationen:
 - ◆ put: Schlange \times \langle Wertetyp \rangle
Anfügen eines Elementes an das Ende der Schlange
 - ◆ get: Schlange \rightarrow \langle Wertetyp \rangle
Herauslösen eines Elementes vom Kopf der Schlange
 - ◆ isEmpty: Schlange \rightarrow boolean
Test, ob die Schlange leer ist
- Sowohl die Schlange als auch der Stapel können als lineare Liste implementiert werden
- Bei der Schlange ist es empfehlenswert nicht nur einen, sondern zwei Zeiger (auf den Anfang und das Ende der Schlange) zu speichern

Doppelt verkettete Liste

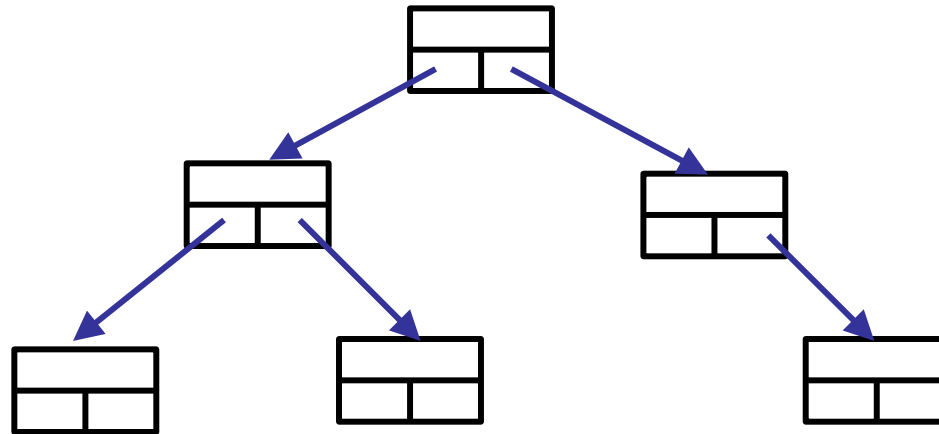


- Die Klasse ListElement erweitert sich dementsprechend um eine weitere Referenz auf das vorhergehende Element

```
class ListElement {  
    <Wertetyp>    content;  
    ListElement  next;  
    ListElement prev;  
}
```

- Dadurch ist der Verwaltungsaufwand beim Einfügen bzw. Löschen von Elementen zwar höher (da zwei Referenzen betrachtet werden müssen) aber das Durchlaufen der Liste kann in beide Richtungen erfolgen

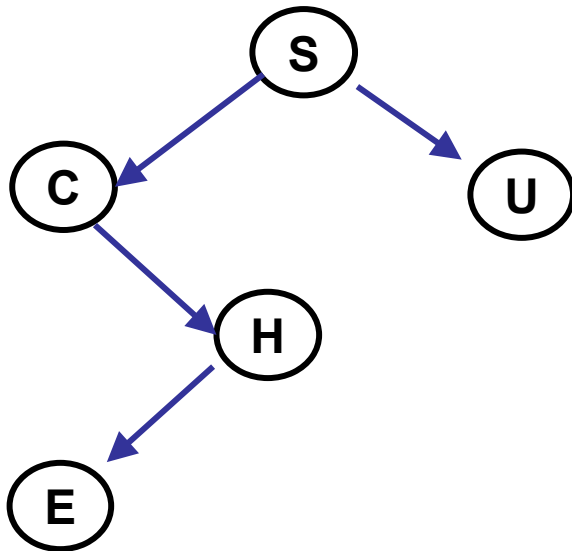
- Doppelt verkettete Strukturen können aber auch einen binären Baum modellieren:



- Alle Listenelemente werden auch Knoten genannt
- Jeder Knoten kann 0, 1 oder 2 Nachfolger (Söhne) haben
- Jeder Knoten hat genau einen Vorgänger (Vater) bis auf den obersten Knoten (die Wurzel)
- Knoten ohne Nachfolger heißen Blätter

Idee: (in Analogie zur binären Suche)

- Jeder Knoten kann bis zu zwei Verweise auf Nachfolger fassen
- Sofern in einem Knoten schon ein Wert gespeichert ist, kann man ab jetzt alle kleineren Werte als linke Nachfolger, größere Werte als rechte Nachfolger speichern
- So entsteht aus "SUCHE" die folgende Struktur:



- Die Suche startet in der Wurzel
- Ist das Vergleichselement kleiner wird links weitergesucht, bei größer rechts
- Dies wiederholt sich, bis das Element gefunden wird, oder der Vergleich bei einem Blatt fehlschlägt und damit das Gesuchte nicht enthalten ist.

??? Fragen



*Welche Fragen
gibt es?*

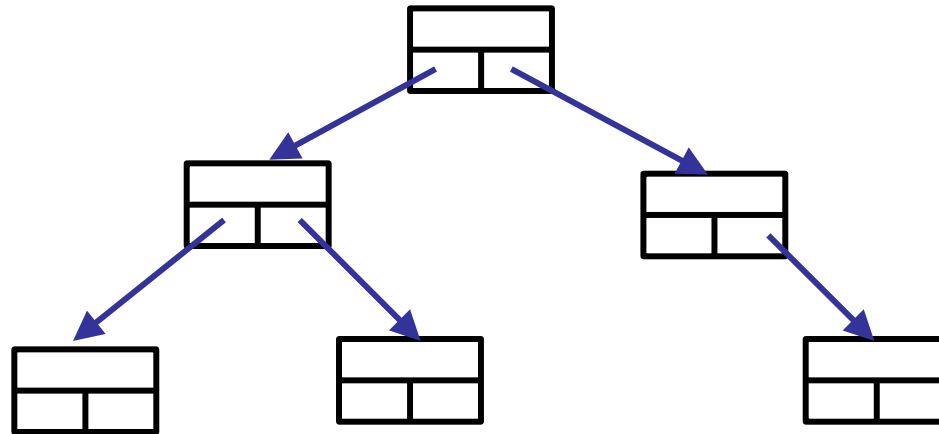
JETZT!



uninformierte Suchalgorithmen

Uninformierte Suchalgorithmen – 1

- Suche eines Knotens in einem Graphen

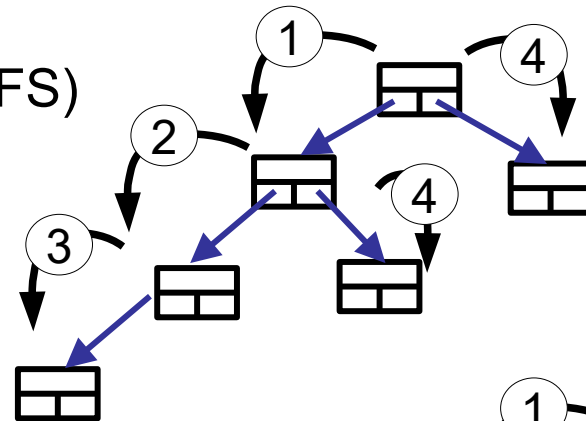


- Die Suchverfahren haben keine Auskunft darüber, wo sich das Ziel befindet.
- Nutzen eine Art Abarbeitungsplan

Arten Uninformierte Suchalgorithmen – 1

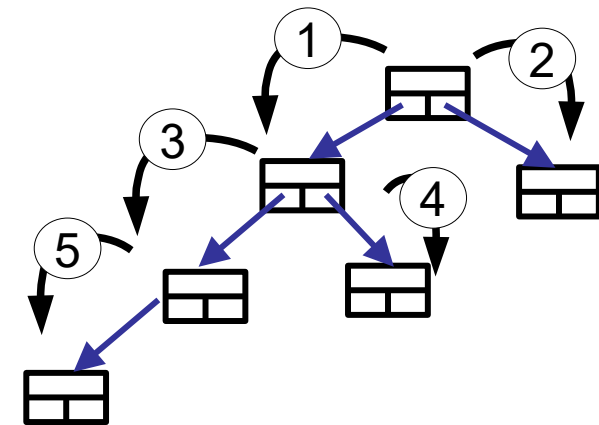
■ Tiefensuche (Depth-First-Search, DFS)

- ◆ Abarbeitungsplan: Stack



■ Breitensuche (Breadth-First-Search, BFS)

- ◆ Abarbeitungsplan: Queue



■ Dijkstra-Suche

- ◆ Erst vielversprechende Knoten.
- ◆ Kostenfunktion
- ◆ Abarbeitungsplan: priorisierte Queue

JETZT!



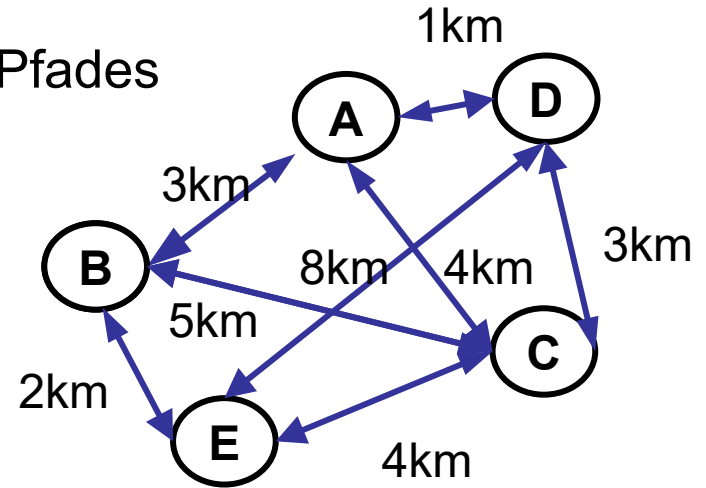
informierte Suchalgorithmen

Informierte Suchalgorithmen – 1

- Verwenden Heuristiken (deshalb auch heuristische Suche)
- Besitzen Kenntnis über den Suchraum oder das Ziel
 - ◆ Entfernungen
 - ◆ Datenverteilung
- Orientieren sich z.B. an menschlichen Problemlösungsstrategien
- Beispiele:
 - ◆ Bestensuche
 - ◆ A*-Algorithmus

A*Algorithmus – 1

- Er dient zur Berechnung des kürzesten Pfades
- Gehört zu den Informierten Suchalgorithmen
- Enthält eine Heuristik
- Arbeitet zielgerichtet und reduziert Laufzeit
- Priorisierte Queue: Kostenfunktion + Schätzfunktion
- 1968 von Peter Hart, Nils Nilsson, Bertram Raphael .



- Füge Startknoten in leere Abarbeitungsliste ein
- Solange günstigster Knoten nicht Zielknoten
 - ◆ Expandiere oberstes Element der Abarbeitungsliste
 - ◆ Entferne expandiertes Element aus Abarbeitungsliste
 - ◆ Füge verbleibende Expansionen nach Wegkosten und Zustandsbewertung **sortiert** in Abarbeitungsliste ein
- Günstigster Weg gefunden, beenden

JETZT!



Software-Entwurf

Schrittweise Verfeinerung – 1

- Die Beschreibung eines Algorithmus wird zu Anfang sehr allgemein formuliert
- Hohe Abstraktionsebene: WAS soll gemacht werden?
- Die schrittweise Verfeinerung liefert: WIE und WOMIT ist die Aufgabenstellung zu lösen?

Top-down-Methode (Abstraktion nach unten)

- Ausgangspunkt ist das Gesamtproblem
- wird in kleinere Teilprobleme zerlegt
- Bei sehr komplexen Aufgabenstellungen können Teilprobleme rekursiv wieder in Teilprobleme zerlegt werden
- Bei der Entwicklung eines einzelnen Programms ist diese Methode vorzuziehen

Bottom-up-Methode (Abstraktion nach oben)

- Diese Methode ist für Vorgehensweisen interessant, bei denen eine Aufgabenstellung nicht exakt beschrieben ist bzw. noch Änderungen erwartet werden
- Einzelne entwickelte Module werden später zu einem großen Ganzen zusammengesetzt
- Die Bottom-up-Methode ist beim Entwurf von großen Programmsystemen durchaus sinnvoll

Up-down-Methode (Middle-Out, Gegenstromverfahren)

- Gesamtaufgabe wird durch Top-down-Methoden verfeinert
- Teilaufgaben werden bottom-up abstrahiert
- Auf diese Weise werden kritische Teilaufgaben zuerst getestet

- Für Algorithmen hatten wir Allgemeinheit gefordert – Die Lösung sollte sich auf ähnliche Probleme übertragen lassen
- Bei Software-(Teil-)Lösungen werden wir analog die Wiederverwendbarkeit fordern.
- Das führt (fast) automatisch zu Software-Modulen (bzw. –Komponenten) mit nach außen bekannt gemachter Schnittstelle

Module ...

- ... können einzeln getestet werden
- ... senken Entwicklungskosten und –zeiten
- ... erlauben eine über Teams verteilte Software-Entwicklung
- ... können Daten und Funktionalitäten kapseln
- ... unterstützen die Zerlegung eines Gesamtproblems in Teilaufgaben
- ... bilden bei guter Programmierung einen wiederverwendbaren Baukasten

Typische Aufgabenbereiche

Aktivität	Inhalt	Ergebnis
Planung	Problemanalyse, Projektzielsetzung	Produktbeschreibung (Lastenheft)
Anforderungsanalyse	Beschreibung Systemanforderung	Anforderungsspezifikation (Pflichtenheft)
Entwurf	Grob- und Feinentwurf Softwarearchitektur	Entwurfsspezifikation
Implementierung	Umsetzung des Entwurfes	Software
Test, Integration	Integration der Module, Test des Systems	Dokumentation
Einsatz und Wartung	Fehlerkorrektur, Änderungsanpassung	neue Version des Systems

- Für eine effiziente Durchführung von Softwareentwicklungsprojekten ist eine zweckmäßige (Ablauf-)Organisation des Entwicklungsprozesses nötig
- So genannte **Vorgehensmodelle (Prozessmodelle)** definieren hierfür entsprechende Rahmenvorgaben
- Dies bezieht sich insbesondere auf Phasenschemata,
 - ◆ die die Reihenfolge der Durchführung von Aktivitäten
 - ◆ unter Berücksichtigung wechselseitiger Abhängigkeiten vorgeben,
 - ◆ ohne dabei unbedingt entwicklungstechnische Methoden festzulegen

- Grundidee (Royce; 1970):
Software wird nicht "in einem Schritt" entwickelt, sondern durchläuft gewisse Entwicklungsstufen (Phasen)

- Aus der Sicht der Umsetzung:
Gliederung der langen Laufzeit eines Projekts in überschaubare (steuer- und kontrollierbare) Intervalle

Beispiel für ein sechsstufiges Vorgehen:

- Planungsphase
- Definitionsphase
- Entwurfsphase
- Implementierungsphase
- Abnahme- und Einführungsphase
- Wartungsphase

- Ausgehend von der Projektidee werden die mit der Entwicklung verfolgten Ziele und der grobe Funktionsumfang beschrieben
- Voruntersuchungen auf Durchführbarkeit und Wirtschaftlichkeit (Machbarkeitsstudie)
- Untersuchung der vorhandenen Hard- und Software auf Verwendbarkeit
- Wahl der Entwicklungsumgebung und Tools

- Spezifikation der Anforderungen an das zu schaffende System
- **Ist-Analyse** des Bereichs, in dem das System eingesetzt werden soll
- Aus dem Ergebnis der Ist-Analyse und den Zielen des Unternehmens wird ein **Soll-Konzept (Pflichtenheft)** erstellt
- Das Pflichtenheft umfasst funktionale und ökonomische Aspekte sowie Qualitätsaspekte

- **Ziel:**
Beschreibung des Gesamtsystems als Hierarchie weitgehend unabhängig von einander entwickelbarer Teilsysteme
- Hieraus entsteht zunächst ein von der Technik unabhängiger **Fachentwurf**. Ergebnisse sind Daten- und Funktionsmodelle oder ein Objektmodell
- Ableitung des **IV-technischen Fachentwurfs**, der Umgebungsbedingungen der Hard- und Software berücksichtigt

■ Ziel:

Herunterbrechen des Systementwurfs bis auf die Ebene einzelner Befehle und Umsetzung in einer Programmiersprache

■ Feinkonzept:

- ◆ Datenschemata
- ◆ Programmabläufe und Funktionen
- ◆ Benutzeroberflächen

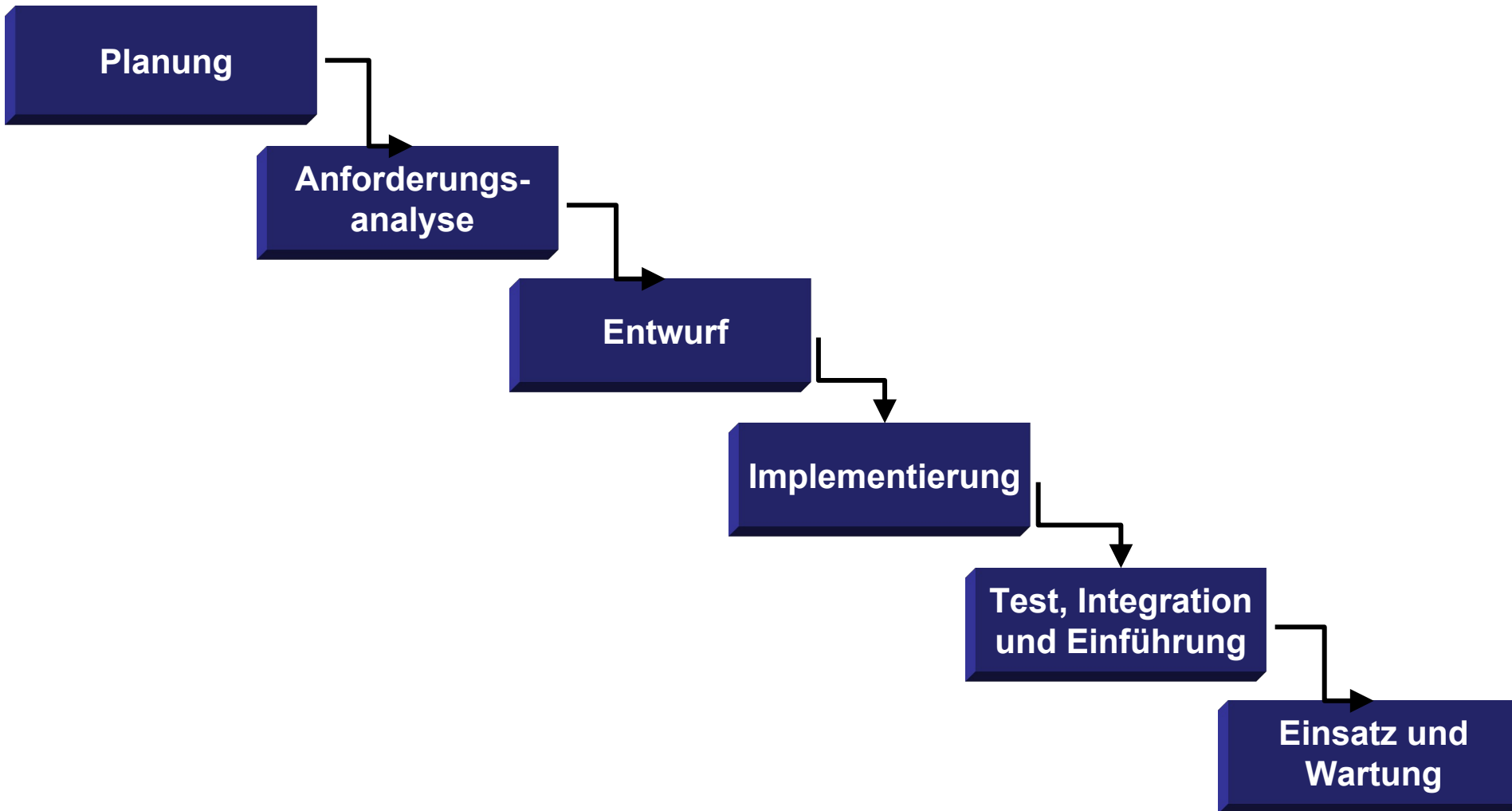
■ Systemtests

Abnahme- und Einführungsphase

- Prüfung, ob die Software die Anforderungen des Pflichtenhefts erfüllt
 - ◆ Checklisten
 - ◆ Testbeds
- Teilweise Prüfung von Entwurfsmethoden und Testprotokollen durch die Auftraggeber
- Anwenderschulungen
- **Inbetriebnahme** der Software

- Nachträgliche Programmänderungen und -anpassungen
- Mögliche Gründe:
 - ◆ Bei Systemtests nicht erkannte Fehler
 - ◆ Veränderte Benutzerwünsche
 - ◆ Änderungen der Rechtlichen Lage (z.B. Steuergesetzgebung)
 - ◆ Änderung der Systemumgebung
- **60 – 70%** aller Entwicklungsarbeiten sind Wartung und Weiterentwicklung!

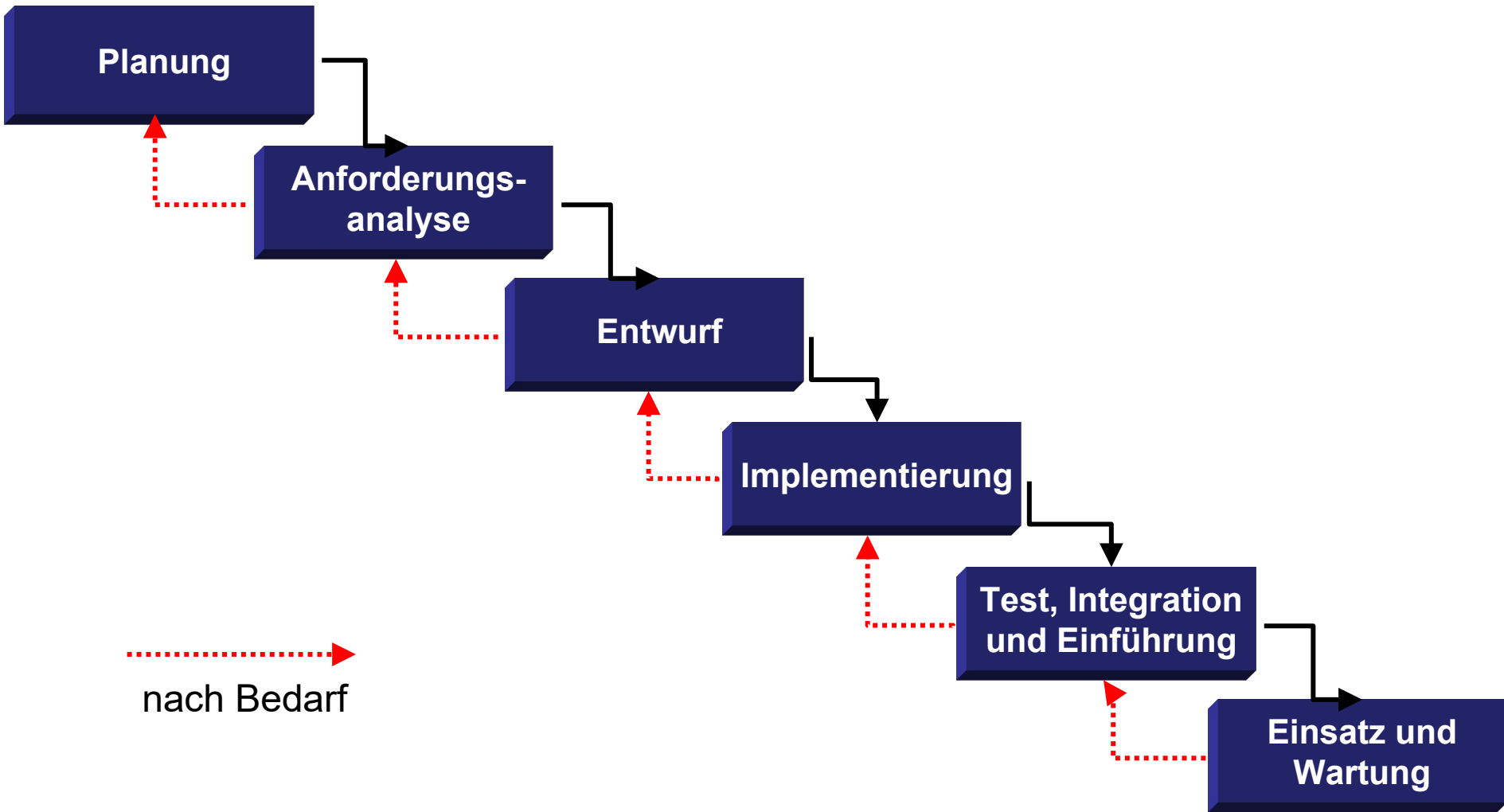
- Aktivitäten bilden abgeschlossene Phasen, die im einfachsten Fall einmalig streng sequenziell durchlaufen werden
- Rücksprünge in der Entwicklung im Sinne einer Erweiterung oder Veränderung der Ergebnisse vorangegangener Phasen sind dann nicht möglich
- Dieses führt zur Analogie des Wasserfalls, bei dem der Fluss streng in eine Richtung orientiert ist



Verbessertes Wasserfallmodell – 1

- Dies entspricht der Realität aber nur selten bzw. bei kleinen Software-Projekten
- Aufgrund von Erkenntnissen in nachgelagerten Phasen kann es notwendig sein, dass Aufgaben in einer vorherigen Phase nachbearbeitet werden müssen
 - ◆ Es kann sich z.B. in der Entwurfsphase herausstellen, dass Anforderungen unklar erfasst sind; oder bei der Implementierung wird festgestellt, dass der Entwurf unvollständig oder inkonsistent ist
- Dementsprechend beinhaltet das verbesserte Wasserfallmodell bereits Rückkopplungsschleifen um jeweils eine Phase
 - ◆ Bedeutet, z.B. ausgehend von der Implementierungsphase kann der Entwurf verändert werden, nicht aber die Anforderungsspezifikation

Verbessertes Wasserfallmodell – 2



■ 👍 Vorteile

- ◆ Einfach
- ◆ Verständlich
- ◆ Mit relativ geringem Koordinierungsaufwand durchführbar

■ 👎 Nachteile

- ◆ Späte Änderungen sind nur mit hohem Aufwand realisierbar
- ◆ Unflexibel durch strenge Sequenz
- ◆ Kein Risikomanagement: Probleme werden erst gegen Ende der Entwicklung erkannt

Vorsicht *FALLE!*



- Die Wichtigkeit des Riskomanagements wird schon allein daran deutlich, dass selbst heute etwa

50% aller kommerziellen Software-Entwicklungsprojekte scheitern oder der Zeitrahmen (und damit in aller Regel auch die Kosten-Budgets) werden gewaltig überschritten

- Neben schlechter Planung liegt die Hauptursache hierfür im schlechten oder fehlenden Risikomanagement

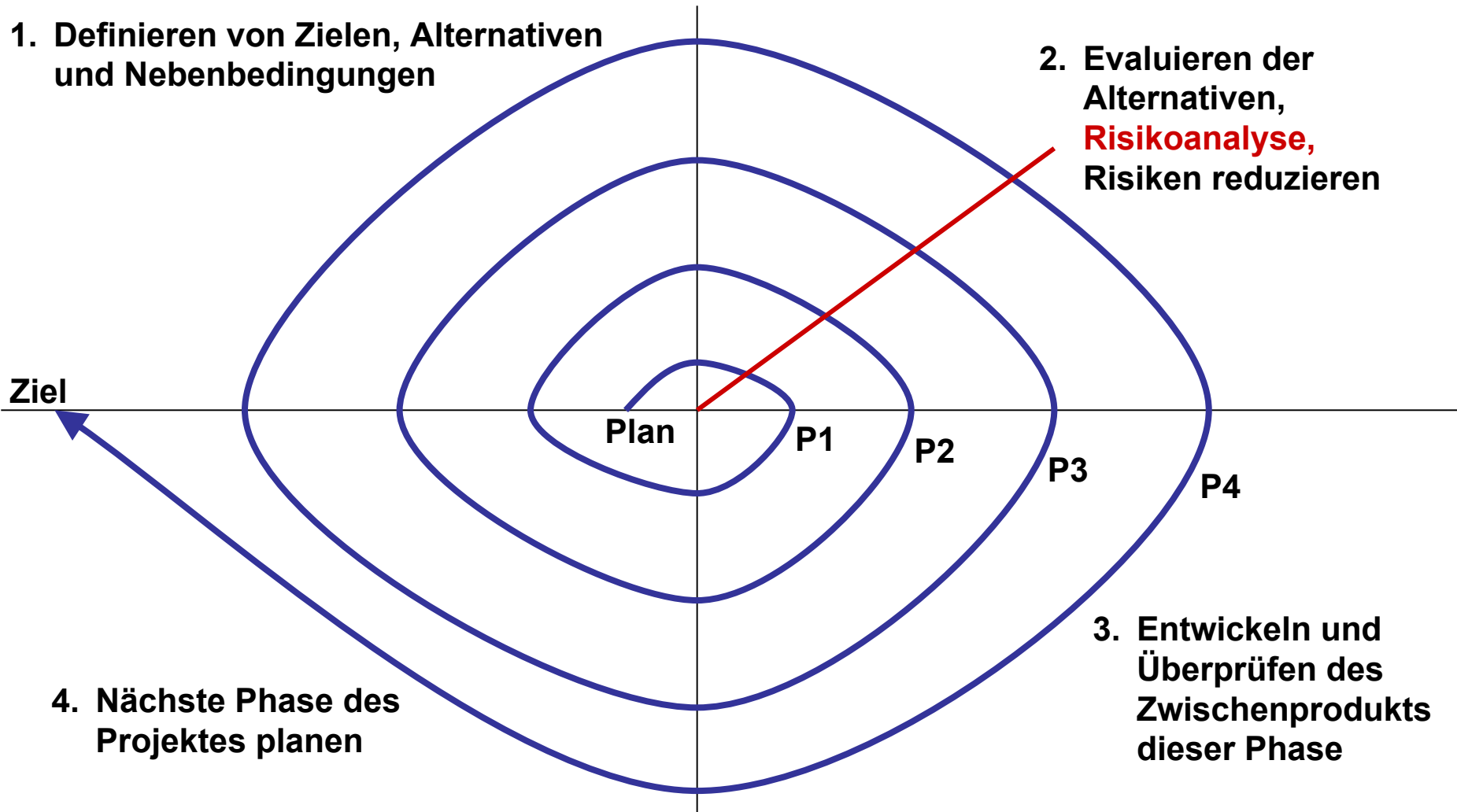
- Das Spiralmodell vereint das klassische lineare Modell mit evolutionären Konzepten (z.B. Einsatz von ablauffähigen, aber nicht voll funktionsfähigen Modell-Entwürfen für Teilaufgaben)

- Software-Entwicklung mit dem Spiralmodell gliedert sich in vier Schritte, die bis zur Fertigstellung des Produkts wiederholt werden:
 - Definition von Zielen, Alternativen und Nebenbedingungen des neuen Spiralprozesses
 - ◆ Zu Beginn jeder Phase (jeder neuen Windung der Spirale) werden inhaltliche Vorgaben für das Teilprodukt definiert

- ◆ Es werden alternative Vorgehensweisen ausgearbeitet
 - ◆ Die Randbedingungen (Nebenbedingungen) wie z.B. Personal, Finanzen und Zeit, sind festzulegen
1. Evaluieren der Ziele und Alternativen, Risiken erkennen und reduzieren
- ◆ Zu Beginn Evaluation der Alternativen unter Berücksichtigung der Ziele und Randbedingungen
 - ◆ Risikofaktoren werden benannt und so weit möglich reduziert
 - ◆ Es können hierzu unterschiedliche Techniken verwendet werden (z.B. Entwicklung von Prototypen P1, P2, P3, ...)

1. Entwickeln und Überprüfen des Zwischenprodukts dieser Phase
 - ◆ In diesem Schritt wird das Teilprodukt (z.B. einzelne Module oder in der letzten Phase die komplette Software) unter Einhaltung des Ziels und der geplanten Ressourcen realisiert und getestet
 - ◆ Die Methode der Realisierung kann flexibel unter Beachtung des Restrisikos gewählt werden
 - ◆ Die fertig getesteten Module sind beispielsweise Ergebnisse dieses Schrittes
2. Nächste Phase des Projektes planen
 - ◆ Als letzter Schritt wird die nächste Phase inhaltlich und organisatorisch geplant
 - ◆ Diese Planung kann mehrere Phasen umfassen, sodass unabhängige Teilprojekte entstehen, die später wieder zusammengeführt werden

- ◆ In einem Review (Prüfung) werden die Schritte 1-3 analysiert, Schlussfolgerungen für die weitere Entwicklung gezogen.
- ◆ Sind die technischen oder wirtschaftlichen Risiken einer Projektfortsetzung zu hoch, kann das Projekt abgebrochen werden
- ◆ Am Ende der Spirale liegt die fertige Software vor, die bezüglich der Anforderungshypothese getestet wird



■ 👍 Vorteile

- ◆ Fehler werden frühzeitig erkannt
- ◆ Regelmäßige Überprüfung der Zwischenprodukte
- ◆ Flexibel, Änderungen sind leicht möglich
- ◆ Alternativen werden betrachtet
- ◆ Risiken werden minimiert

■ 👎 Nachteile

- ◆ Für kleinere bis mittlere Projekte nicht so gut geeignet
- ◆ Hoher Managementaufwand
- ◆ Risikoaspekte werden nicht konsequent berücksichtigt

Merke!

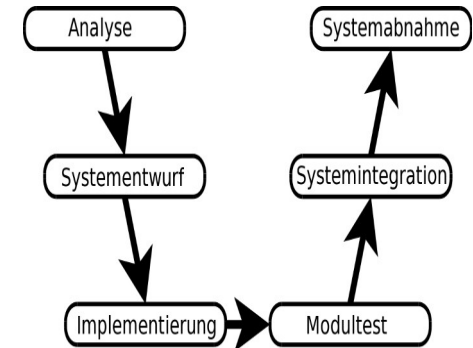


■ Es gibt viele weitere Vorgehensmodelle, wie z.B.

◆ **das V-Modell**

als Weiterentwicklung des Wasserfallmodells, dass jede Phase, die weiter ins Detail geht, treppenartig versetzt unter der vorhergehenden anordnet, und zwar:

V-förmig von Durchführbarkeitskonzept bis zur Modul-implementierung abwärts und wieder aufwärts bis zum Betrieb



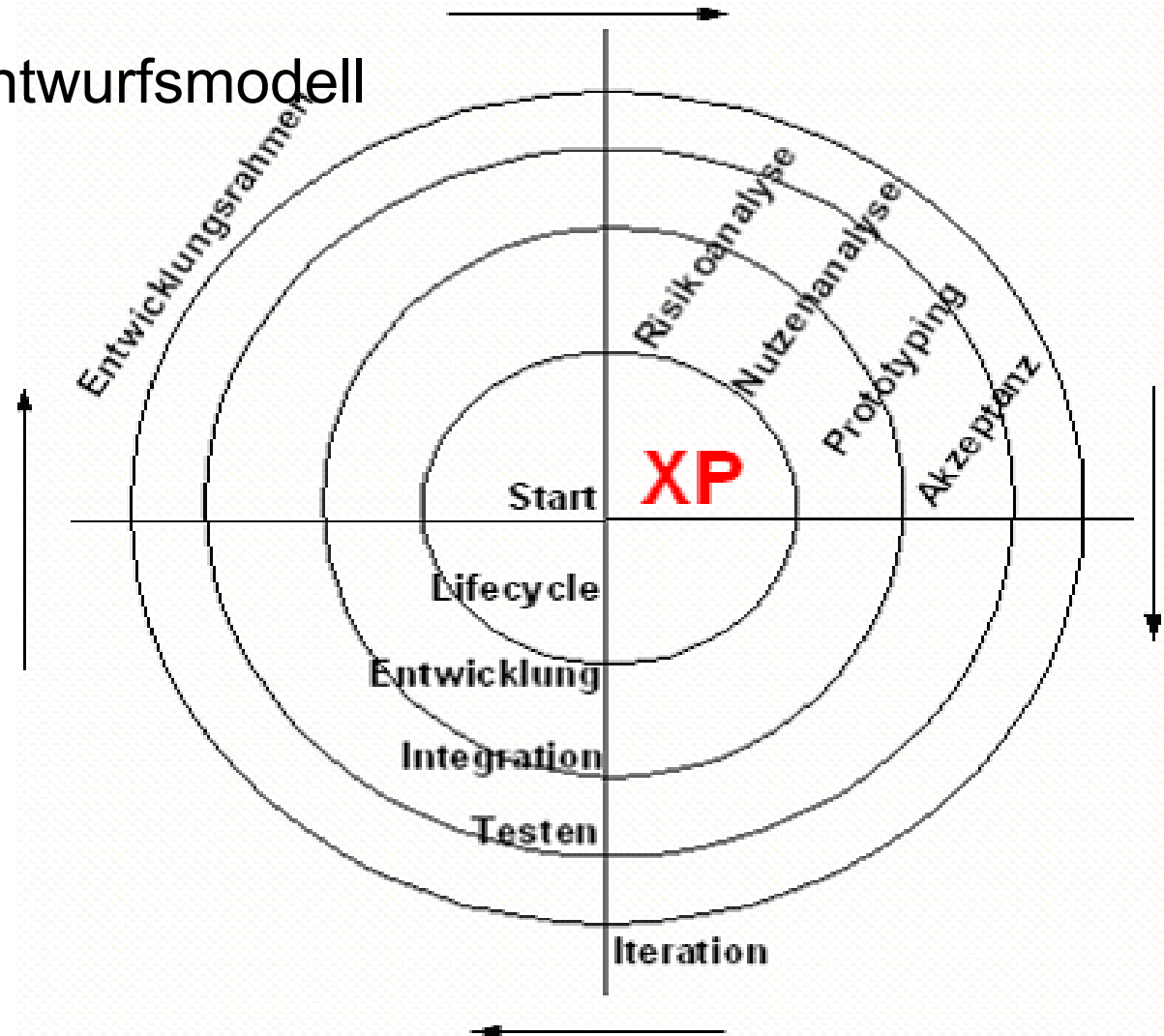
◆ **Prototyping**

ein evolutionäres Modell zur Verkürzung der Entwicklungszeiten, Überprüfung der Realisierbarkeit des Projektes und dem Aufzeigen verschiedener Lösungsansätze und –möglichkeiten
Großer Vorteil: Kann mit anderen Vorgehensmodellen verbunden werden

■ Kein „starres“ Entwurfsmodell

■ Definiert:

- Werte,
- Praktiken
- Prinzipien



Grafik: Quelle wikimedia

Praktiken/Werte XP

Praktik	Richtiges Vorgehen nach XP	Traditionelles oder falsches Vorgehen/Risiko nach XP
Pair Programming	Zu zweit am Rechner.	Jeder will und muss zunächst auf seine ihm zugewiesenen Aufgaben schauen.
Planning Poker	Team bestimmt realistische Planungszeiten	Einzelne Vorausplanung
Refactoring	Suboptimales Design und Fehler werden akzeptiert.	Fehler sind verpönt. Erstellte Artefakte laufen angeblich immer direkt perfekt.
Metapher	Ein gemeinsames Vokabular.	Kunde und Entwicklung sprechen in zwei Sprachen, häufig aneinander vorbei.

Praktik	Richtiges Vorgehen nach XP	Traditionelles oder falsches Vorgehen/Risiko nach XP
Qualität	Inhärenter Bestandteil. * Siehe QA-Tests	Der Faktor, der als erster vernachlässigt wird, wenn Zeit oder Geld knapp werden.
Testgetriebene Entwicklung	Testen hat sehr hohen Stellenwert. * Tests	Testen kostet nur Zeit. Wenige manuelle Tests.
Kommunikation	Stetiger Austausch wird gefördert und erwartet.	Jeder muss zunächst mal seine Aufgaben lösen.
Mut	Offene Atmosphäre.	Angst vor versäumten Terminen und Missverständnissen mit Kunden.
Kollektives Eigentum	Programmcode, Dokumente etc. gehören dem Team.	Jeder fühlt sich nur für seine Artefakte verantwortlich.

Praktik	Richtiges Vorgehen nach XP	Traditionelles oder falsches Vorgehen/Risiko nach XP
Integration	Stetige Integrationen erlauben Feedback und erhöhen Qualität.	Selten Integrationen, da vermeintlich unnütz und Zeitverschwendung.
Kundeneinbeziehung	Der Kunde wird zur aktiven Mitarbeit aufgerufen.	Der Kunde ist selten wirklicher Partner sondern nur die „andere Seite des Vertrages“.
Iterationen	Ein Release wird in viele handliche Iterationen unterteilt.	Iterationen sind nicht nötig, es wird an einem Release gearbeitet.
Stand-up Meeting	Täglicher, strukturierter Austausch.	Große, lange, seltenere Projektmeetings. Die Personenanzahl und der Inhalt sind häufig zu aufgebläht.

Praktik	Richtiges Vorgehen nach XP	Traditionelles oder falsches Vorgehen/Risiko nach XP
Dokumentation	Wo es sinnvoll ist.	Wichtiges Artefakt. Alles muss standardisiert dokumentiert sein. Dokumentation wird aber nicht genutzt.
Team	Das Team ist sehr wichtig. Es existieren keine Rollen. Feedback wird von jedem erwartet.	Spezialistentum. Abschottung. Wissensmonopole.
Standards	Standards, wo es sinnvoll erscheint.	Überregulierung. Starrer Prozess.

Scrum (engl. „Gedränge“)

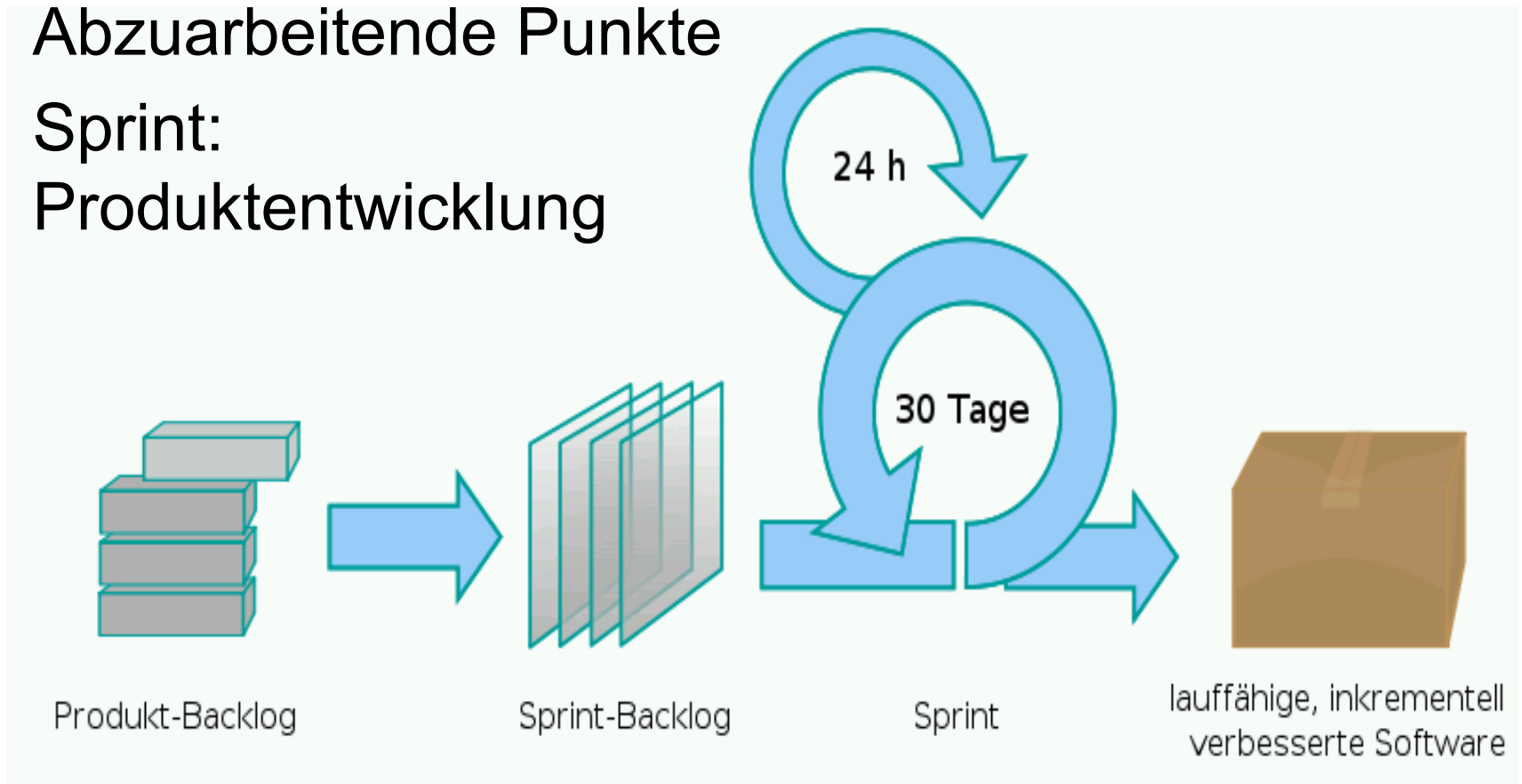
Vorgehensmodell der Softwaretechnik.

SCRUM basiert auf der Erkenntnis, dass große moderne Entwicklungsprojekte zu komplex sind, um durchgängig planbar zu sein.

Scrum versucht, die Komplexität zu reduzieren und die Flexibilität im Projekt zu erhalten (Agil)

- **Transparenz:** Der Fortschritt und Hindernisse des Projektes werden täglich und für alle Projektteilnehmer sichtbar festgehalten.
- **Überprüfung:** Produktfunktionalitäten werden in regelmäßigen Abständen geliefert und beurteilt.
- **Anpassung:** Anforderungen an das Produkt werden nach jeder Lieferung neu bewertet und bei Bedarf angepasst.

- Backlog:
Abzuarbeitende Punkte
- Sprint:
Produktentwicklung



Quelle: Wikimedia

??? **Frage**



***Welche Fragen
gibt es?***