Vorlesung Informatik 1 (Wintersemester 2020/2021)

Kapitel 11: Algorithmen

Martin Frieb
Johannes Metzger

Universität Augsburg
Fakultät für Angewandte Informatik

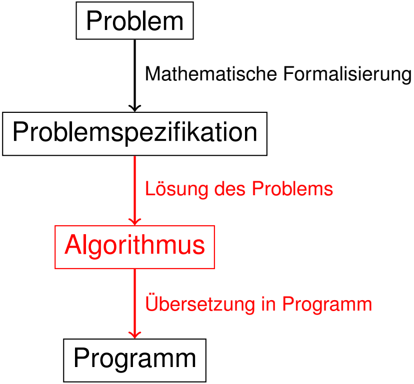
18. Januar 2021

Universität Augsburg
Institut für Informatik

11. Algorithmen
11.1 Motivation
11.2 Bestandteile von Algorithmen
11.3 Darstellungsweisen von Algorithmen
11.4 Nichtdeterministische Algorithmen
11.5 Iterative und rekursive Algorithmen

11. Algorithmen
11.1 Motivation
11.2 Bestandteile von Algorithmen
11.3 Darstellungsweisen von Algorithmen
11.4 Nichtdeterministische Algorithmen
11.5 Iterative und rekursive Algorithmen

Motivation



Zur Lösung eines Problems entwirft man vor dem Programm ein **programmiersprachen-unabhängiges Modell der Lösung**:

- Zerlegung der Problemlösung in mehrere Schritte
- Abstraktion von Spezifika von Programmiersprachen

Was ist ein Algorithmus?

Das älteste informatische Konzept zur Lösung eines Problems, erfunden lange vor der Informatik selbst, ist der **Algorithmus**

Definition 11.1 (Algorithmus)

Ein Algorithmus ist

- … eine exakte Formulierung
- … von mechanisch ausführbaren Abläufen
- … zur Lösung beliebig vieler Instanzen / Einzelfälle
- … eines Problems

Beispiel 11.2 (Beispiele aus dem Alltag)

- Kochrezepte
- Bauanleitungen
- Bedienungsanleitungen

Euklidischer Algorithmus (3. Jahrhundert vor Chr.)

Der Euklidische Algorithmus berechnet den größten gemeinsamen Teiler (ggT) von zwei positiven ganzen Zahlen

Darstellung in Pseudocode

Algorithmus : *ggT*
Eingabe : $m, n \in \mathbb{N}$
$a \leftarrow m$;
$b \leftarrow n$;
solange $b > 0$ **tue**
  $r \leftarrow a \bmod b$;
  $a \leftarrow b$;
  $b \leftarrow r$;
Ausgabe : a

m, n: Eingabe-Parameter
Jedes konkrete für m, n eingesetzte Zahlenpaar ist eine **Instanz**

$\leftarrow$: Wertzuweisung

a, b, r: Variablen

Euklidischer Algorithmus (3. Jahrhundert vor Chr.)

Der Euklidische Algorithmus berechnet den größten gemeinsamen Teiler (ggT) von zwei positiven ganzen Zahlen

Darstellung in Pseudocode

Algorithmus : *ggT*
Eingabe : $m, n \in \mathbb{N}$
$a \leftarrow m$;
$b \leftarrow n$;
solange $b > 0$ **tue**
    $r \leftarrow a \bmod b$;
    $a \leftarrow b$;
    $b \leftarrow r$;
Ausgabe : a

m, n: Eingabe-Parameter
Jedes konkrete für m, n eingesetzte Zahlenpaar ist eine **Instanz**

$\leftarrow$: Wertzuweisung

a, b, r: Variablen

Euklidischer Algorithmus (3. Jahrhundert vor Chr.)

Der Euklidische Algorithmus berechnet den größten gemeinsamen Teiler (ggT) von zwei positiven ganzen Zahlen

Beispiel-Instanz: $m = 12$, $n = 8$

Algorithmus : *ggT*
Eingabe : $m, n \in \mathbb{N}$
$a \leftarrow m$;
$b \leftarrow n$;
solange $b > 0$ **tue**
  $r \leftarrow a \mod b$;
  $a \leftarrow b$;
  $b \leftarrow r$;
Ausgabe : a

1.	2.	3.
$a = 12$		
$b = 8$		
$8 > 0?$	$4 > 0?$	$0 > 0?$
$r = 4$	$r = 0$	
$a = 8$	$a = 4$	
$b = 4$	$b = 0$	
		4

Euklidischer Algorithmus (3. Jahrhundert vor Chr.)

Der Euklidische Algorithmus berechnet den größten gemeinsamen Teiler (ggT) von zwei positiven ganzen Zahlen

Darstellung als Funktion in C

Algorithmus : *ggT*
Eingabe : $m, n \in \mathbb{N}$
$a \leftarrow m$;
$b \leftarrow n$;
solange $b > 0$ **tue**
  $r \leftarrow a \mod b$;
  $a \leftarrow b$;
  $b \leftarrow r$;
Ausgabe : a

```c
int ggT(int m, int n) {
  int a = m;
  int b = n;
  int r;
  while (b > 0) {
    r = a % b;
    a = b;
    b = r;
  }
  return a;
}
```

11. Algorithmen
11.1 Motivation
11.2 Bestandteile von Algorithmen
11.3 Darstellungsweisen von Algorithmen
11.4 Nichtdeterministische Algorithmen
11.5 Iterative und rekursive Algorithmen

Bestandteile von Algorithmen

Algorithmen in der Informatik

Ein **Algorithmus** ist eine exakte Formulierung mechanisch ausführbarer Abläufe zur Lösung beliebig vieler Instanzen eines **Problems der Informationsverarbeitung**

Bestandteile

- *Daten*: repräsentieren die zu verarbeitenden Informationen
- *Anweisungen:* verarbeiten die Daten, z.B. durch Eingabe, Wertzuweisung, Ausführung eines Algorithmus, Rechenoperation, Ausgabe
- *Ablaufstrukturen*: bringen die Anweisungen in die richtige Reihenfolge. Es gibt 3 Arten von Ablaufstrukturen: Sequenzen, Fallunterscheidungen und Wiederholungen

Bestandteile von Algorithmen

Darstellung von Daten

Daten werden mittels Variablen beschrieben.

Darstellung von Anweisungen

Anweisungen können durch Elementaroperationen auf unterschiedlichen Abstraktionsebenen ausgedrückt werden. Wir erlauben:

- natürlichsprachliche Formulierungen
- mathematische Ausdrücke

Darstellung von Ablaufstrukturen

Mögliche Darstellungen von **Ablaufstrukturen** sind Programmablaufpläne, Struktogramme und Pseudocode

Unterschiede zu C-Programmen

In der Formulierung von Algorithmen gibt es folgende Unterschiede zu einem C-Programm:

- Bedingungen, Rechenausdrücke und Anweisungen dürfen auch natürlichsprachlich formuliert werden
  Müssen in C durch (Kombination von) C-Elementaroperationen ausgedrückt (implementiert) werden
- Abstraktion von Datentypen und Bibliotheksfunktionen
  Müssen in C ergänzt werden
- Mathematische Operationsbezeichnungen ($\leftarrow$, $\leq$, ...)
  Müssen in C durch neue Schreibweisen ersetzt werden

11. Algorithmen
11.1 Motivation
11.2 Bestandteile von Algorithmen
11.3 Darstellungsweisen von Algorithmen
11.4 Nichtdeterministische Algorithmen
11.5 Iterative und rekursive Algorithmen

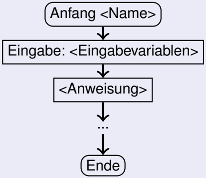
Beginn, Ende, Name und Eingabe

Pseudocode

Algorithmus : <Name>
Eingabe : <Eingabevariablen>
<Anweisung>
...

Programmablaufplan

- Es gibt Knoten für Anfang, Eingabe, Ausgabe, jede Anweisung und Ende
- Pfeile legen die Reihenfolge fest



C-Programm

Eingabevariablen = Parameter

```
<T> <Name>(<Parameter>)
{
  <Anweisung>;
  ...
}
```

Struktogramm

- Es gibt Blöcke für Eingabe, Ausgabe, jede Anweisung und jede Kontrollstruktur
- Anordnung der Blöcke legt die Reihenfolge fest

Algorithmus: <Name>

Eingabe: <Eingabevariablen>
<Anweisung>
...

Sequenz von Anweisungen X, Y, Z, ...

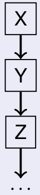
Pseudocode

```
X;
Y;
Z;
...
```

C-Programm

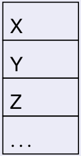
```
X;
Y;
Z;
...
```

Programmablaufplan



Struktogramm

Sequenzen werden von oben nach unten in gleich breiten Blöcken angeordnet



Fallunterscheidungen: X,Y Anweisungen / U,V Teilmodelle

Teilmodelle können aus mehreren Anweisungen bestehen oder auch leer sein

Pseudocode

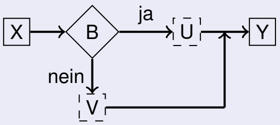
```
X;
wenn B dann
    U
sonst
    V
Y;
```

C-Programm

```c
X;
if (B) { U }
else { V }
Y;
```

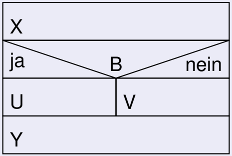
Programmablaufplan

Pfeile können zusammengeführt werden



Struktogramm

Die Blöcke für U und V werden in den Block für die Fallunterscheidung verschachtelt



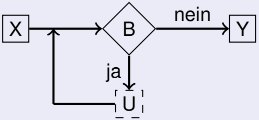
Wiederholungen: X,Y Anweisungen / U Teilmodell

- Teilmodelle können aus mehreren Anweisungen bestehen oder auch leer sein
- In allen Modellen dürfen in Wiederholungen `break` und `continue` benutzt werden

Pseudocode

```
X;
solange B tue
    U
Y;
```

Programmablaufplan

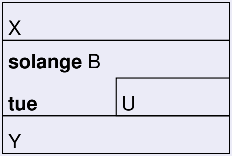


C-Programm

```c
X;
while (B) {
   U
}
Y;
```

Struktogramm

Der Block für U wird in den Block für die Wiederholung verschachtelt

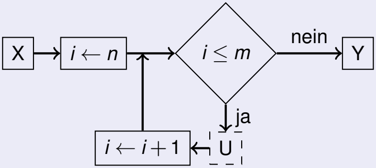


Wiederholungen: X,Y Anweisungen / U Teilmodell

Pseudocode ($n \leq m$)

X;
für $i \leftarrow n$ **bis** m **tue**
  U
Y;

Programmablaufplan ($n \leq m$)

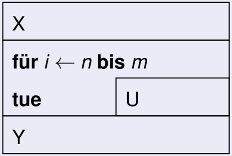


C-Programm ($n \leq m$)

(Beachte Indexverschiebung bei Feldern)

```
int i;
X;
for (i = n; i <= m; ++i) {
  U
}
Y;
```

Struktogramm ($n \leq m$)



Wiederholungen: X,Y Anweisungen / U Teilmodell

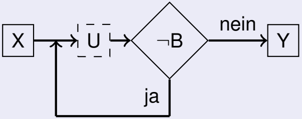
Pseudocode

```
X;
wiederhole
│   U
bis B;
Y;
```

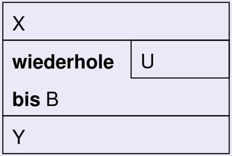
C-Programm

```c
X;
do {
  U
} while (!B);
Y;
```

Programmablaufplan



Struktogramm



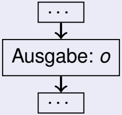
Einwertige Ausgabe

- Die Ausgabe ist im Algorithmus eine eigene Anweisung
- Gibt es **genau eine Ausgabevariable im Algorithmus** (**einwertige Ausgabe**), dann wird deren Wert in der C-Funktion mit einer `return`-Anweisung zurückgegeben

Pseudocode

```
...;
Ausgabe : o
...;
```

Programmablaufplan



C-Programm

```c
return o;
```

Struktogramm

...
Ausgabe: *o*
...

Mehrwertige Ausgabe

- Die Ausgabe ist im Algorithmus eine eigene Anweisung
- Gibt es **mehrere Ausgabevariablen im Algorithmus** (**mehrwertige Ausgabe**), dann werden die Ausgabewerte über adresswertige Eingabeparameter gespeichert

Pseudocode

$\ldots$;

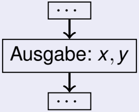
Ausgabe : x, y

$\ldots$;

C-Programm

```
void <Name>(T *x, T *y)
```

Programmablaufplan



Struktogramm

...
Ausgabe: x, y
...

Algorithmus-Aufruf

- Ein Algorithmus algo kann wie eine Funktion in der Form algo(EP) aufgerufen werden (EP = Eingabeparameter)
- Bei Aufruf eines Algorithmus **kann** seine Ausgabe (falls vorhanden) einer Variablen `out` als Wert zugewiesen werden

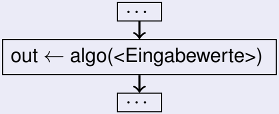
Pseudocode

```
...;
out ← algo(<Eingabewerte>);
...;
```

C-Programm

```c
out = Algo(a,...);
```

Programmablaufplan



Struktogramm

...
out ← algo(<Eingabewerte>)
...

Beispiel: Binäre Suche

Problemstellung

Finde für eine **aufsteigend sortierte** Zahlenfolge $a_1, \ldots, a_n$ und eine neue Zahl s die **Einfügeposition** i in $a_1, \ldots, a_n$, an der s unter Beibehaltung der Sortierung eingefügt werden kann:

Algorithmus : bsearch
Eingabe : $a_1, \ldots, a_n \in \mathbb{N}$ sortiert, $s \in \mathbb{N}$, $n \in \mathbb{N}$
$li \leftarrow 0$;
$re \leftarrow n+1$;
solange $li < re - 1$ **tue**
  $m \leftarrow (li + re) \div 2$;
  wenn $s \leq a_m$ **dann**
    $re \leftarrow m$;
  sonst
    $li \leftarrow m$;
Ausgabe : re

Ausführungsbeispiel

Zahlenfolge:
$a_1 = 1, a_2 = 4, a_3 = 7, a_4 = 9, a_5 = 12, a_6 = 15, a_7 = 16, a_8 = 20$
Neue Zahl: $s = 8$

1. $(li, re) = (0, 9)$, $m = 4$, $s \leq a_4$
2. $(li, re) = (0, 4)$, $m = 2$, $s > a_2$
3. $(li, re) = (2, 4)$, $m = 3$, $s > a_3$
4. $(li, re) = (3, 4)$, Ausgabe: $re = 4$

Beispiel: Binäre Suche

Algorithmus : bsearch
Eingabe : $a_1, \ldots, a_n \in \mathbb{N}$ sortiert, $s \in \mathbb{N}$, $n \in \mathbb{N}$
$li \leftarrow 0$;
$re \leftarrow n + 1$;
solange $li < re - 1$ **tue**
    $m \leftarrow (li + re) \div 2$;
    wenn $s \leq a_m$ **dann**
        $re \leftarrow m$;
    sonst
        $li \leftarrow m$;
Ausgabe : re

Ausführungsbeispiel

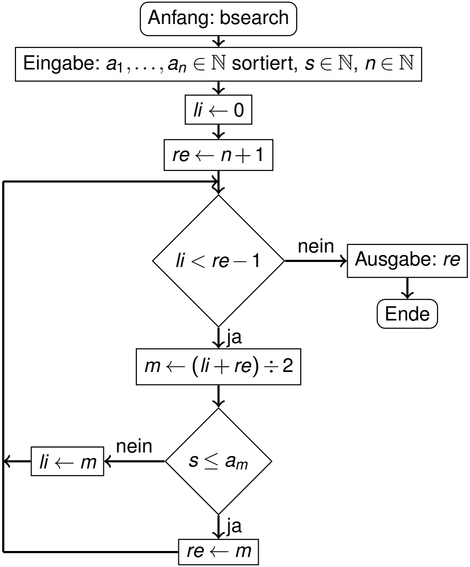
Zahlenfolge:
$a_1 = 1, a_2 = 4, a_3 = 7, a_4 = 9, a_5 = 12, a_6 = 15, a_7 = 16, a_8 = 20$
Neue Zahl: $s = 8$

1. $(li, re) = (0, 9)$, $m = 4$, $s \leq a_4$
2. $(li, re) = (0, 4)$, $m = 2$, $s > a_2$
3. $(li, re) = (2, 4)$, $m = 3$, $s > a_3$
4. $(li, re) = (3, 4)$, Ausgabe: $re = 4$

- Vor und nach jedem Schleifendurchlauf gilt: $a_{li} < s \leq a_{re}$
- Folge wird in jedem Durchlauf halbiert: Fortsetzung der Suche in unterer oder oberer Teilfolge
- Anzahl der Durchläufe bei 1000000 Elementen: 20.

Beispiel: Binäre Suche

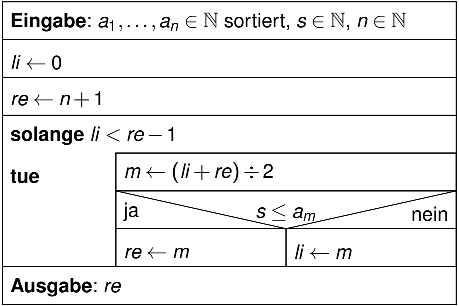
Algorithmus : bsearch
Eingabe : $a_1, \ldots, a_n \in \mathbb{N}$ sortiert, $s \in \mathbb{N}$, $n \in \mathbb{N}$
$li \leftarrow 0$;
$re \leftarrow n + 1$;
solange $li < re - 1$ **tue**
    $m \leftarrow (li + re) \div 2$;
    wenn $s \leq a_m$ **dann**
        $re \leftarrow m$;
    sonst
        $li \leftarrow m$;
Ausgabe : re



Beispiel: Binäre Suche

Algorithmus : bsearch
Eingabe : $a_1, \ldots, a_n \in \mathbb{N}$ sortiert, $s \in \mathbb{N}$, $n \in \mathbb{N}$
$li \leftarrow 0$;
$re \leftarrow n+1$;
solange $li < re - 1$ **tue**
    $m \leftarrow (li + re) \div 2$;
    wenn $s \leq a_m$ **dann**
        $re \leftarrow m$;
    sonst
        $li \leftarrow m$;
Ausgabe : re

Algorithmus: bsearch



Beispiel: Binäre Suche

Algorithmus : bsearch

Eingabe : $a_1, \ldots, a_n \in \mathbb{N}$ sortiert, $s \in \mathbb{N}$, $n \in \mathbb{N}$

$li \leftarrow 0;$

$re \leftarrow n + 1;$

solange $li < re - 1$ **tue**

  $m \leftarrow (li + re) \div 2;$

  wenn $s \leq a_m$ **dann**

    $re \leftarrow m;$

  sonst

    $li \leftarrow m;$

Ausgabe : re

```
int search_bin_sorted(int a[],
     int n, int s) {
  int li = -1;
  int re = n;
  int m;
  while (li < re - 1) {
    m = (li + re) / 2;
    if (s <= a[m])
      re = m;
    else
      li = m;
  }
  return re;
}
```

11. Algorithmen
11.1 Motivation
11.2 Bestandteile von Algorithmen
11.3 Darstellungsweisen von Algorithmen
11.4 Nichtdeterministische Algorithmen
11.5 Iterative und rekursive Algorithmen

Was ist ein nichtdeterministischer Algorithmus?

Algorithmus : bsearchNdet
Eingabe : $a_1, \ldots, a_n \in \mathbb{N}$ sortiert, $s \in \mathbb{N}$, $n \in \mathbb{N}$
$li \leftarrow 0$;
$re \leftarrow n + 1$;
solange $li < re - 1$ **tue**
  Wähle m zufällig mit $li < m < re$;
  wenn $s \leq a_m$ **dann**
    $re \leftarrow m$;
  sonst
    $li \leftarrow m$;
Ausgabe : re

Definition 11.3 (Determinismus / Lokale Eindeutigkeit)

Ein Algorithmus heißt **deterministisch**, falls die Wirkung bzw. das Ergebnis jeder einzelnen Anweisung eindeutig ist und an jeder einzelnen Stelle des Ablaufs festliegt, welcher Schritt als nächstes auszuführen ist.

Andernfalls heißt ein Algorithmus **nichtdeterministisch**.

11. Algorithmen
11.1 Motivation
11.2 Bestandteile von Algorithmen
11.3 Darstellungsweisen von Algorithmen
11.4 Nichtdeterministische Algorithmen
11.5 Iterative und rekursive Algorithmen

Was ist ein rekursiver Algorithmus?

Definition 11.4 (Rekursive / Iterative Algorithmen)

Ein Algorithmus heißt **rekursiv**, wenn er sich selbst (direkt oder indirekt) wieder aufruft.

Andernfalls heißt er **iterativ**.

Alle bisherigen Algorithmen waren iterativ, z.B.:

- Berechnung des ggT
- Suchverfahren
- ...

Direkte Rekursion

Direkte Rekursion

Man spricht von **direkter Rekursion**, wenn sich ein Algorithmus selbst wieder aufruft.

Beispiel 11.5 (Berechnung der Fakultät $n!$ einer Zahl $n \in \mathbb{N}_0$)

Wir definieren **induktiv**:

- (**Induktionsanfang**) $0! := 1$.
- (**Induktionsschritt**) $n! := n \cdot (n-1)!$ für $n > 0$.

Die Berechnung für n wird auf die Berechnung für $(n-1)$ zurückgeführt.
Berechnung von 3!:

$$\begin{aligned} 3! &= 3 \cdot (2!) = 3 \cdot (2 \cdot (1!)) = 3 \cdot (2 \cdot (1 \cdot (0!))) \\ &= 3 \cdot (2 \cdot (1 \cdot 1)) = 3 \cdot (2 \cdot 1) = 3 \cdot 2 = 6 \end{aligned}$$

Die Berechnung erfolgt nach diesem Schema also “rückwärts”: Um 3! berechnen zu können, muss zuerst 2! berechnet werden, dazu wiederum zuerst 1!, und so weiter.

Direkte Rekursion

Direkte Rekursion

Man spricht von **direkter Rekursion**, wenn sich ein Algorithmus selbst wieder aufruft.

Beispiel 11.6 (Berechnung der Fakultät $n!$ einer Zahl $n \in \mathbb{N}_0$)

Wir definieren **induktiv**:

- $0! := 1$.
- $n! := n \cdot (n-1)!$ für $n > 0$.

Induktive Definitionen können direkt in rekursive Algorithmen umgesetzt werden

```
1 Algorithmus : factorial
  Eingabe : n ∈ ℕ0
2 wenn n = 0 dann
     Ausgabe : 1
3 sonst
     Ausgabe : n·factorial(n − 1)
```

- Der Algorithmus ruft sich in Zeile 3 selbst wieder auf mit **neuen Werten für die Eingabeparameter**
- Beachte: Es gibt nur endlich viele rekursive Aufrufe factorial$(n-1)$, ..., factorial(1), factorial(0) bis zur **Abbruchbedingung** $n = 0$ in Zeile 2 (es kommt also zu keiner sog. **unendlichen Rekursion**)

Direkte Rekursion

Direkte Rekursion

Man spricht von **direkter Rekursion**, wenn sich ein Algorithmus selbst wieder aufruft.

Beispiel 11.7 (Berechnung der Fakultät $n!$ einer Zahl $n \in \mathbb{N}_0$)

Implementierung des Algorithmus als C-Funktion (ohne Test auf Bereichsüberlauf)

```c
unsigned long int fakultaet(
    unsigned int n)
{
  if (n == 0)
    return 1L;
  else
    return n * fakultaet(n-1);
}
```

Algorithmus : factorial
Eingabe : $n \in \mathbb{N}_0$
wenn $n = 0$ **dann**
  Ausgabe : 1
sonst
  Ausgabe : $n \cdot \text{factorial}(n-1)$

Direkte Rekursion: Allgemeines Schema

Allgemeines Schema für direkt rekursive Algorithmen

```
1 Algorithmus : algo
  Eingabe : EP
2 wenn Abbruchbedingung(EP) dann
3     Direkte Lösung ohne Aufruf von algo;
4 sonst
5     algo(EP');
6     Lösung des Problems für EP;
```

- Beim rekursiven Aufruf algo(EP') werden für EP neue Eingabewerte EP' eingesetzt, so dass nach endlich vielen rekursiven Aufrufen die von den Eingabewerten abhängige Abbruchbedingung erfüllt ist (ist dies nicht der Fall, so spricht man von einer **unendlichen Rekursion**)
- algo(EP) muss mit der Lösung des Problems (Zeile 6) auf die Beendigung von algo(EP') (Zeile 5) warten

Direkte Rekursion: Binäre Suche

```
1  Algorithmus : bsearch
   Eingabe : a_1, ..., a_n ∈ ℕ
             sortiert, s ∈ ℕ,
             n ∈ ℕ
2  li ← 0;
3  re ← n + 1;
4  solange li < re − 1 tue
5      m ← (li + re) ÷ 2;
6      wenn s ≤ a_m dann
7          re ← m;
8      sonst
9          li ← m;
   Ausgabe : re
```

- Eingabe: Schranken für den Suchbereich li, re werden mit übergeben
- Zeile 2: Abbruchbedingung
- Zeilen 5,6: Rekursive Aufrufe mit neuen Werten für li, re
- **Erster Aufruf**: bsearchRek$(a_1, \ldots, a_n, s, 0, n+1)$

```
1  Algorithmus : bsearchRek
   Eingabe : a_1, ..., a_n ∈ ℕ sortiert, s ∈ ℕ, n, li, re ∈ ℕ,
             0 ≤ li < re ≤ n + 1
2  wenn li ≥ re − 1 dann
       Ausgabe : re
3  sonst
4      m ← (li + re) ÷ 2;
5      wenn s ≤ a_m dann
           Ausgabe : bsearchRek(a_1, ..., a_n, s, li, m)
6      sonst
           Ausgabe : bsearchRek(a_1, ..., a_n, s, m, re)
```

Direkte Rekursion: Abarbeitung im Stack

Direkte Rekursion

Man spricht von **direkter Rekursion**, wenn sich ein Algorithmus selbst wieder aufruft.

Abarbeitung im Stack

Erfolgt bei der Abarbeitung einer C-Funktion $F(EP)$ der rekursive Aufruf $F(EP')$, dann passiert folgendes:

- Es wird für $F(EP')$ ein aktiver Stack Frame auf dem Stack erzeugt
- Da der Aufruf $F(EP)$ noch nicht abgearbeitet ist, wird der Stack Frame $F(EP)$ nicht mehr aktiv und kann erst nach $F(EP')$ beendet werden

Bei n aufeinanderfolgenden rekursiven Aufrufen $F(EP_1),\ldots,F(EP_n)$ werden also insgesamt n Stack Frame erzeugt und die Abarbeitung erfolgt rückwärts in der Reihenfolge $F(EP_n),\ldots,F(EP_1)$.

Direkte Rekursion: Abarbeitung im Stack

```
int main () {
  fakultaet(3);
}

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

- Aufruf für $n = 3$.

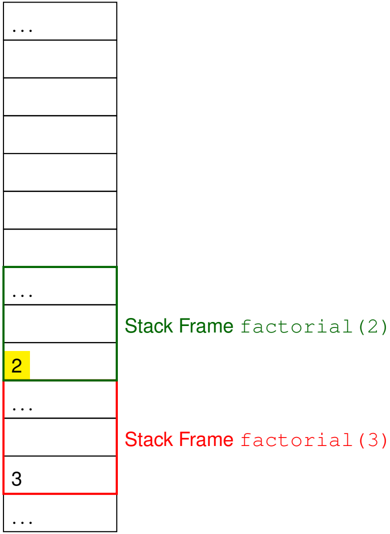
Stack Frame `factorial(3)`

Direkte Rekursion: Abarbeitung im Stack

```
int main () {
  fakultaet(3);
}

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

- Aufruf für $n = 3$.
- Zur Berechnung von `3 * factorial(2)` wird `factorial(2)` aufgerufen

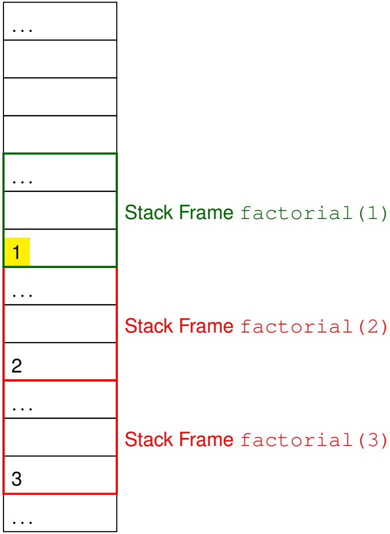


Direkte Rekursion: Abarbeitung im Stack

```
int main () {
  fakultaet(3);
}

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

- Aufruf für $n = 3$.
- Aufruf für $n = 2$
- Zur Berechnung von `2 * factorial(1)` wird `factorial(1)` aufgerufen

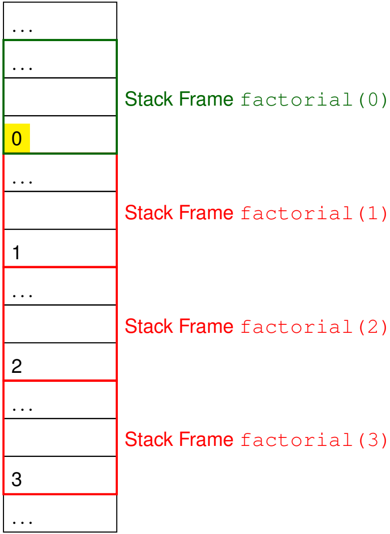


Direkte Rekursion: Abarbeitung im Stack

```
int main () {
  fakultaet(3);
}

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

- Aufruf für $n = 3$.
- Aufruf für $n = 2$
- Aufruf für $n = 1$
- Zur Berechnung von `1 * factorial(0)` wird `factorial(0)` aufgerufen

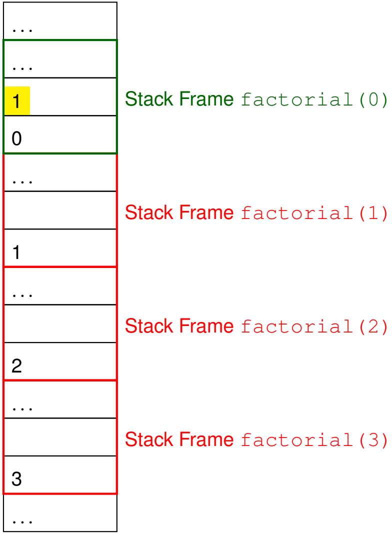


Direkte Rekursion: Abarbeitung im Stack

```
int main () {
  fakultaet(3);
}

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

- Aufruf für $n = 3$.
- Aufruf für $n = 2$
- Aufruf für $n = 1$
- Direkte Berechnung von `factorial(0)` ohne weiteren rekursiven Aufruf

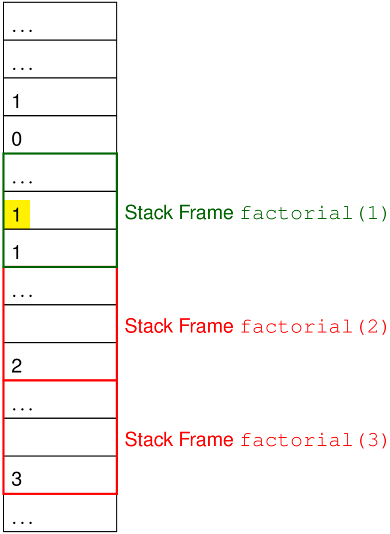


Direkte Rekursion: Abarbeitung im Stack

```
int main () {
  fakultaet(3);
}

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

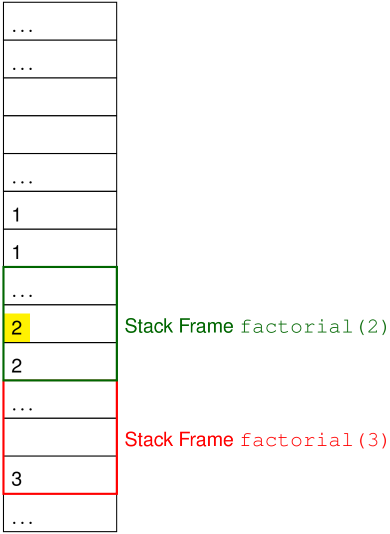
- Aufruf für $n = 3$.
- Aufruf für $n = 2$
- Berechnung von `factorial(1)` $\leftarrow$ `1 * factorial(0)`



Direkte Rekursion: Abarbeitung im Stack

```
int main () {
  fakultaet(3);
}

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



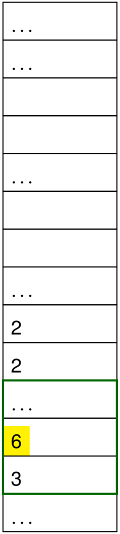
- Aufruf für $n = 3$.
- Berechnung von `factorial(2)` $\leftarrow$ `2 * factorial(1)`

Direkte Rekursion: Abarbeitung im Stack

```
int main () {
  fakultaet(3);
}

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

- Berechnung von `factorial(3)` $\leftarrow$ `3 * factorial(2)`



Stack Frame `factorial(3)`

Direkte Rekursion: Bewertung

- Rekursive Algorithmen sind oft **einfacher zu implementieren** als iterative Algorithmen
- Allerdings benötigen sie **deutlich mehr Arbeitspeicher**, und sind damit **wesentlich ineffizienter** und beinhalten außerdem die Gefahr eines Stack Overflow bei einer zu großen Rekursionstiefe
- Rekursive Algorithmen spielen eine sehr wichtige Rolle bei Verwendung von Bäumen als Datenstrukturen (siehe später und Vorlesung Informatik 3)

Indirekte Rekursion

Indirekte Rekursion

Man spricht von **indirekter Rekursion**, wenn es mehrere Algorithmen $A_1, \ldots, A_n$ $(n > 1)$ gibt, die sich im Zyklus aufrufen:
A_1 ruft A_2 auf, A_2 ruft A_3 auf, ..., A_{n-1} ruft A_n auf, A_n ruft A_1 auf.

Beispiel 11.8 (Test auf gerade / ungerade Zahl)

Algorithmus : isEven
Eingabe : $n \in \mathbb{N}_0$
wenn $n = 0$ **dann**
  Ausgabe : 1
sonst
  Ausgabe : isOdd$(n-1)$

Algorithmus : isOdd
Eingabe : $n \in \mathbb{N}_0$
wenn $n = 0$ **dann**
  Ausgabe : 0
sonst
  Ausgabe : isEven$(n-1)$

Zusammenhang zwischen Rekursion und Iteration

Endständig rekursiver Algorithmus

Bei einem **endständig rekursiven Algorithmus** (**tail recursion**) treten nur rekursive Aufrufe auf, bei denen das Ergebnis des Aufrufs nicht "nachbearbeitet" werden muss, sondern direkt zurückgegeben wird.

Allgemeines Schema für endständig rekursive Algorithmen

Algorithmus : algo
Eingabe : x
wenn $B(x)$ **dann**
  Ausgabe : algo$(E(x))$
sonst
  Ausgabe : $A(x)$

- x: Eingabeparameter
- $B(x)$: von x abhängige Bedingung
- $E(x)$, $A(x)$: Von x abhängige Ausdrücke

Gegenbeispiel: Rekursive Berechnung der Fakultät

Zusammenhang zwischen Rekursion und Iteration

Ausführung endständig rekursiver Funktinonen

- Für endständige rekursive Algorithmen / Funktionen ist eine **direkte Umschreibung in eine iterative Form** möglich.
- Dies wird bei C-Funktionen von optimierenden Compilern häufig aus Effizienzgründen automatisch gemacht

Übersetzung in iterative Form

Algorithmus : algo
Eingabe : x
wenn $B(x)$ **dann**
    Ausgabe : algo($E(x)$)
sonst
    Ausgabe : $A(x)$

Algorithmus : algo
Eingabe : x
solange $B(x)$ **tue**
    $x \leftarrow E(x)$;
Ausgabe : $A(x)$

Zusammenhang zwischen Rekursion und Iteration

Ausführung endständig rekursiver Funktinonen

- Für endständige rekursive Algorithmen / Funktionen ist eine **direkte Umschreibung in eine iterative Form** möglich.
- Dies wird bei C-Funktionen von optimierenden Compilern häufig aus Effizienzgründen automatisch gemacht

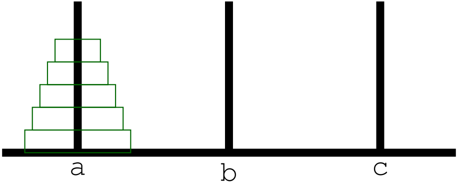
Beispiel 11.9 (Berechnung des Rests bei ganzzahliger Division (modulo))

Algorithmus : mod
Eingabe : $x \in \mathbb{N}_0$, $y \in \mathbb{N}$
wenn $x \geq y$ **dann**
  Ausgabe : $\text{mod}(x - y, y)$
sonst
  Ausgabe : x

Algorithmus : mod
Eingabe : $x \in \mathbb{N}_0$, $y \in \mathbb{N}$
solange $x \geq y$ **tue**
  $x \leftarrow x - y$;
Ausgabe : x

Die Türme von Hanoi

- Ziel des Spiels ist es, den kompletten Stapel mit *n* Scheiben von Stab `a` auf Stab `c` zu versetzen
- Bei jedem Zug darf die oberste Scheibe eines beliebigen Stapels auf einen der beiden anderen Stäbe gelegt werden, vorausgesetzt, dort liegt nicht schon eine kleinere Scheibe
- Folglich sind zu jedem Zeitpunkt des Spieles die Scheiben auf jedem Stab der Größe nach geordnet

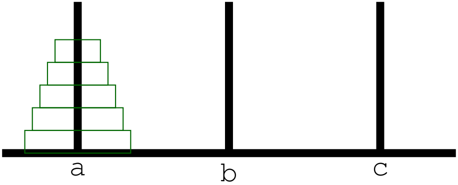


Die Türme von Hanoi

Rekursive Lösungsidee:

1. Falls $n > 1$: Versetze die obersten $(n-1)$ Scheiben von `a` nach `b` (unter Benutzung von `c`)
2. Lege die unterste Scheibe von `a` nach `c`
3. Falls $n > 1$: Versetze die $(n-1)$ Scheiben von `b` nach `c` (unter Benutzung von `a`)

$n = 5$:

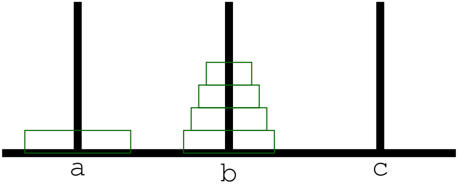


Die Türme von Hanoi

Rekursive Lösungsidee:

1. Falls $n > 1$: Versetze die obersten $(n-1)$ Scheiben von `a` nach `b` (unter Benutzung von `c`)
2. Lege die unterste Scheibe von `a` nach `c`
3. Falls $n > 1$: Versetze die $(n-1)$ Scheiben von `b` nach `c` (unter Benutzung von `a`)

$n = 5$: **Erfordert weitere rekursive Aufrufe**

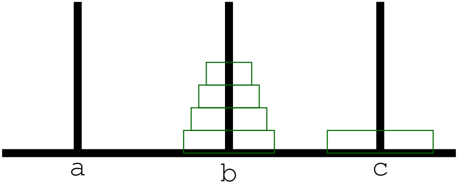


Die Türme von Hanoi

Rekursive Lösungsidee:

1. Falls $n > 1$: Versetze die obersten $(n-1)$ Scheiben von `a` nach `b` (unter Benutzung von `c`)
2. Lege die unterste Scheibe von `a` nach `c`
3. Falls $n > 1$: Versetze die $(n-1)$ Scheiben von `b` nach `c` (unter Benutzung von `a`)

$n = 5$**: Direkt ausführbar**

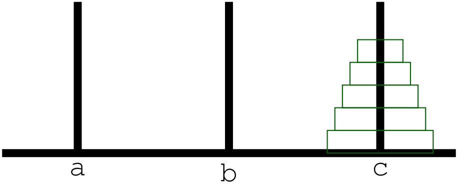


Die Türme von Hanoi

Rekursive Lösungsidee:

1. Falls $n > 1$: Versetze die obersten $(n-1)$ Scheiben von `a` nach `b` (unter Benutzung von `c`)
2. Lege die unterste Scheibe von `a` nach `c`
3. Falls $n > 1$: Versetze die $(n-1)$ Scheiben von `b` nach `c` (unter Benutzung von `a`)

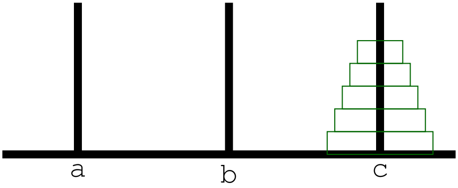
$n = 5$: **Erfordert weitere rekursive Aufrufe**



Die Türme von Hanoi

Rekursive Lösungsidee:

1. Falls $n > 1$: Versetze die obersten $(n-1)$ Scheiben von `a` nach `b` (unter Benutzung von `c`)
2. Lege die unterste Scheibe von `a` nach `c`
3. Falls $n > 1$: Versetze die $(n-1)$ Scheiben von `b` nach `c` (unter Benutzung von `a`)



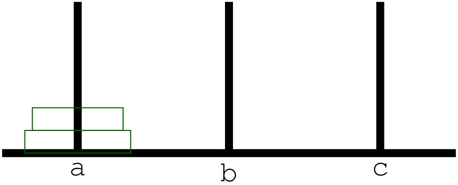
- Das Versetzen von 5 Scheiben wurde so auf das Versetzen von 4 Scheiben zurückgeführt
- Nach demselben Prinzip führt man das Versetzen des Stapels Schritt für Schritt zurück auf das Versetzen einzelner Scheiben

Die Türme von Hanoi

Rekursive Lösungsidee:

1. Falls $n > 1$: Versetze die obersten $(n-1)$ Scheiben von `a` nach `b` (unter Benutzung von `c`)
2. Lege die unterste Scheibe von `a` nach `c`
3. Falls $n > 1$: Versetze die $(n-1)$ Scheiben von `b` nach `c` (unter Benutzung von `a`)

Beispiel: 2 Scheiben

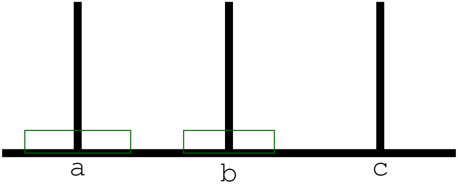


Die Türme von Hanoi

Rekursive Lösungsidee:

1. Falls $n > 1$: Versetze die obersten $(n-1)$ Scheiben von `a` nach `b` (unter Benutzung von `c`)
2. Lege die unterste Scheibe von `a` nach `c`
3. Falls $n > 1$: Versetze die $(n-1)$ Scheiben von `b` nach `c` (unter Benutzung von `a`)

Beispiel: 2 Scheiben

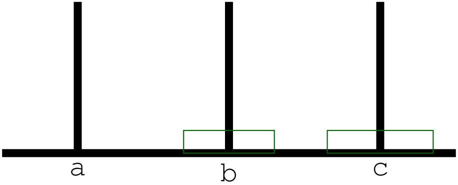


Die Türme von Hanoi

Rekursive Lösungsidee:

1. Falls $n > 1$: Versetze die obersten $(n-1)$ Scheiben von `a` nach `b` (unter Benutzung von `c`)
2. Lege die unterste Scheibe von `a` nach `c`
3. Falls $n > 1$: Versetze die $(n-1)$ Scheiben von `b` nach `c` (unter Benutzung von `a`)

Beispiel: 2 Scheiben

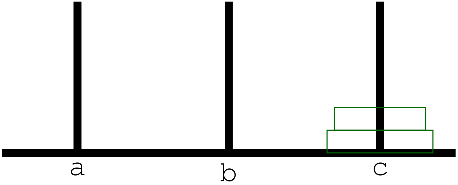


Die Türme von Hanoi

Rekursive Lösungsidee:

1. Falls $n > 1$: Versetze die obersten $(n-1)$ Scheiben von `a` nach `b` (unter Benutzung von `c`)
2. Lege die unterste Scheibe von `a` nach `c`
3. Falls $n > 1$: Versetze die $(n-1)$ Scheiben von `b` nach `c` (unter Benutzung von `a`)

Beispiel: 2 Scheiben



Die Türme von Hanoi

Spielzüge ausgeben mit einer C-Funktion:

```
void schritt(char a, char b)
{
    printf("Lege␣Scheibe␣von␣%c␣nach␣%c\n", a, b);
}

void hanoi(int n, char a, char b, char c)
{
    if (n > 0) {
        hanoi(n - 1, a, c, b);
        schritt(a, b);
        hanoi(n - 1, c, b, a);
    }
}
```