Chair of Embedded Systems
Department of Computer Science
University of Augsburg, Germany

Martin Frieb

Universität Augsburg University

Wintersemester 2020/2021

Informatik 1

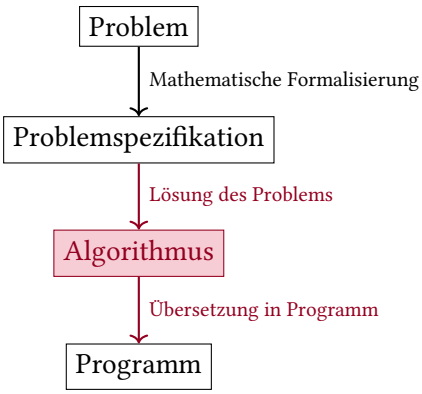
Kapitel 11 – Algorithmen

Inhaltsverzeichnis

Version 20210226-2113

11.1. Motivation

Im letzten Kapitel haben wir uns mit der Problemspezifikation befasst. Ausgehend von dieser wollen wir uns nun den Grundlagen der Algorithmen zuwenden.



Nachdem wir also gelernt haben, wie man Probleme in einer Problemspezifikation darstellt, erstellen wir nun basierend auf der Problemspezifikation einen Algorithmus, der zur Lösung eines Problems führt. Im Allgemeinen sind Algorithmen nichts anderes als ein präzise formuliertes Verfahren, das in endlicher Zeit ausgeführt werden kann, um ein Problem zu lösen. Im Großen und Ganzen können wir deswegen auch ein Kochrezept als Algorithmus verstehen, oder die Bauanleitung für ein Lego Projekt.

Möchte man nun ein Lösungsverfahren für ein Problem entwerfen und in ein Programm integrieren, beschreibt man vor dem Programmieren das Verfahren zum Lösen als ein **programmiersprachenunabhängiges Modell**. Hierfür zerlegt man die Problemlösung in mehrere Schritte und schreibt diese so auf, dass sie ohne Spezifika von Programmiersprachen auskommt. Dadurch erhalten wir ein schrittweises Lösungsverfahren, welches sich nun leichter in einer bestimmten Programmiersprache umsetzen lässt.

Was ist ein Algorithmus?
Schon lange vor der Informatik sind Algorithmen entstanden, um die Lösung von Problemen in Teilschritte zu zerlegen.

> Definition: 11.1 Algorithmus
>
> Ein Algorithmus ist
>
> - eine exakte Formulierung
> - von mechanisch/technisch/logisch/arithmetisch ausführbaren Abläufen
> - zur Lösung beliebig vieler Instanzen / Einzelfälle eines Problems.

Beispiel: Vorgreifendes Beispiel für ein Algorithmus in der Informatik

Euklidischer Algorithmus (3. Jahrhundert vor Chr.)
Der Euklidische Algorithmus berechnet den größten gemeinsamen Teiler (ggT) von zwei positiven ganzen Zahlen. Sie können ihn anhand des folgenden C-Programms nachvollziehen:

```
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;
}
```

Programme/Codestücke sind nur eine mögliche Darstellung von Algorithmen. Wir wollen den Euklidischen Algorithmus nun programmiersprachen-unabhängig darstellen. Im folgenden Beispiel sehen wir das obige C-Programm daher als *Pseudocode*:

Beispiel: Euklidischer Algorithmus als Pseudocode

```
1 Algorithmus: ggT
  Eingabe: m, n ∈ ℕ
2 a ← m;
3 b ← n;
4 solange b > 0 tue
5 |   r ← a mod b;
6 |   a ← b;
7 |   b ← r;
  Ausgabe: a
```

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

←: Wertzuweisung

a, b, r: Variablen

Versuchen Sie, den Euklidischen Algorithmus im Pseudocode nachzuvollziehen. Wir wählen nun ein konkretes m und ein konkretes n und schreiben in einer Tabelle auf, was passiert. Durch die Schleife verändern sich die Variablen jede Iteration.

Beispiel: m=12, n=8

Zeile	1. Iteration	2. Iteration	3. Iteration
2	a=12		
3	b=8		
4	8 > 0?	4 > 0?	0 > 0?
5	r = 4	r = 0	
6	a= 8	a = 4	
7	b = 4	b = 0	
8			Ausgabe: 4

Stupide gesagt findet man auch viele alltäglichen Beispiele für Algorithmen wie beispielsweise Kochrezepte, Bauanleitungen oder Bedienungsanleitungen. Jedoch interessieren wir uns in erster Linie für mathematische Konzepte, welche sich in C-Code überführen lassen.

Im Kapitel 11.2 erklären wir zunächst die Grundlagen, um **Algorithmen** schreiben und verstehen zu können. Anschließend stellen wir verschiedene Darstellungsarten vor und wie dort die Grundlagen umgesetzt werden (Kapitel 11.3). Dann folgt im Kapitel 11.4 der Begriff des **Nichtdeterminismus** und wie dieser in Algorithmen auftaucht und verwendet werden kann. Schließlich befassen wir uns im Kapitel 11.5 mit dem Konzept der Rekursion, bevor wir mit einem Fazit enden.

11.2. Bestandteile von Algorithmen

Algorithmen in der Informatik:
Ein **Algorithmus** ist eine exakte Formulierung mechanisch/technisch/logisch/arithmetisch 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

Ausgehend von folgendem Beispiel in C-Code lassen sich relativ einfach die verschiedenen Bestandteile identifizieren.

Beispiel:

```
1  int main(void)
2  {
3      /* Daten */
4      int x = 1;
```

```
5       int y = 7;
6       int a = -1;
7
8       /* Anweisung */
9       a = x + y;
10
11      /* Fallunterscheidung */
12      if (a > 0) {
13          printf("Ja");
14      } else {
15          printf("Nein");
16      }
17
18      return 0;
19  }
```

Die Folge von mehreren Befehlen, die nicht z.B. durch Fallunterscheidungen unterbrochen wird (z.B. die Initialisierung der Daten) nennen wir *Sequenz*.

Für uns lässt sich für den weiteren Verlauf der Vorlesung folgende Regeln festhalten:

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*. Wir schauen uns diese im Kapitel 11.3 genauer an.

Unterschiede der Darstellungen 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.
  In C müssen diese durch (Kombination von) C-Elementaroperationen ausgedrückt (implementiert) werden.
- Abstraktion von Datentypen und Bibliotheksfunktionen.
  Diese müssen in C ergänzt werden.
- Mathematische Operationsbezeichnungen ($\leftarrow$, $\leqslant$, ...).
  Müssen in C durch entsprechende C-eigene Schreibweisen ersetzt werden.

11.3. Darstellungsweisen von Algorithmen

Hinweis
Die Reihenfolge der folgenden Seiten unterscheidet sich stark vom Foliensatz. Es werden nacheinander Pseudocode, Programmablaufplan sowie das Struktogramm vorgestellt. Für eine Gegenüberstellung der Darstellungsarten wird hier auf den Foliensatz verwiesen.

Zum besseren Verständnis der neuen Darstellungsformen wird auf der linken Seitenhälfte das neue Konzept vorgestellt und rechts der zugehötige C-Code als Referenz aufgeführt. So können sie das neu erlangte Wissen gleich ihrem bisherigen zuordnen und hoffentlich schneller anwenden.

11.3.1. Pseudocode

Beginn, Ende, Name und Eingabe
Der Anfang eines Pseudocodes ist immer gleich. Es muss zum Anfang immer ein Name sowie die Eingabevariablen gegeben sein.

Eingabevariablen = Parameter

```
1 Algorithmus: <Name>
  Eingabe: <Eingabevariablen>
2 <Anweisung>
3 ...
```

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

Sequenz von Anweisungen X, Y, Z, ...
Unter einer Sequenz von Anweisungen versteht man das „ganz normale Abarbeiten von oben nach unten“. Als Anweisungen verstehen wir (Wert-)Zuweisungen, arithmetische Operationen, Fallunterscheidungen, Wiederholungen und weitere Algorithmen(Funktionsaufrufe). Es gilt zu beachten, dass auch im Pseudocode die Semikolons eingefügt werden müssen.

```
1 X;
2 Y;
3 Z;
4 ...
```

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

Wertzuweisungen
Um einer Variable einen Wert zuzuweisen wird im Pseudocode das Pfeilsymbol ← verwendet.

```
1 ...;
2 a ← 3;
3 ...;
```

```
...
a = 3;
...
```

Fallunterscheidungen: X,Y Anweisungen / U,V Teilmodelle

Unter einem Teilmodell versteht man einen Anweisungsblock, der z.B. bedingt oder wiederholt ausgeführt wird. In der folgenden Beschreibung also `U` und `V`. Teilmodelle können dabei aus mehreren Anweisungen bestehen oder auch leer sein.

```
1 X;
2 wenn B dann
3 |  U
4 sonst
5 |  V
6 Y;
```

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

Es gilt hierbei zu beachten, dass die Striche links vom `U` bzw. `V` Block immer angegeben werden müssen. Sie geben eine klare visuelle Trennung der Fallunterscheidung an.

Beispiel:

```
1 Algorithmus: Beispiel
  Eingabe: x ∈ ℕ
2 x ← x + 1;
3 wenn x > 10 dann
4 |  x ← x ÷ 2;
5 |  Wähle m zufällig mit m ∈ {1, 2, ..., 100};
6 |  x ← x − m;
```

Beachte hier: Der Strich geht über alle drei Anweisungen im wenn-Fall. Wenn das eingegebene `x` also $\leqslant 10$ ist, wird der Block übersprungen und `x` nicht verändert.

Wiederholungen: X,Y Anweisungen / U Teilmodell

Im Pseudocode kann man natürlich auch Schleifen darstellen. Auch hier gilt, dass die Teilmodelle leer sein können. Wie in C können innerhalb von Schleifen die Befehle `break` und `continue` verwendet werden. Es ist möglich, `while`, `for` und `do-while` Schleifen darzustellen:

```
1 X;
2 solange B tue
3 |  U
4 Y;
```

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

Beachten Sie bei der folgenden Darstellung einer `for` Schleife in Pseudocode, dass bei der Umsetzung in C die Schleife bis einschließlich `m` durchlaufen wird. Sofern Sie eine Schleife schreiben wollen, in der über die Einträge eines Feldes iteriert, beachten Sie, dass Felder im Pseudocode i.d.R. bei 1 beginnen, wohingegen sie in C-Code bei 0 beginnen. Entsprechend müsste die Schleife im Pseudocode von 1 bis m iterieren.

```
X;
für i ← n bis m tue
    U
Y;
```

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

```
X;
wiederhole
    U
bis B;
Y;
```

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

Hier muss man aufpassen, da die Logik im Pseudocode genau umgekehrt zu der in C ist: In C wird eine `do-while` Schleife durchlaufen, solange `B` **nicht** erfüllt ist. Im Pseudocode wird der Anweisungsblock `U` wiederholt, bis `B` **erstmalig** erfüllt ist.

Einwertige Ausgabe

Die Ausgabe ist im Algorithmus immer 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

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

```
...
return o;
...
```

Mehrwertige Ausgabe

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

```
...;
Ausgabe: x, y
...;
```

```
void <Name>(T *x, T *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. (Der Name vom Algorithmus `algo` sowie der Variable `out` sind natürlich frei wählbar).

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

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

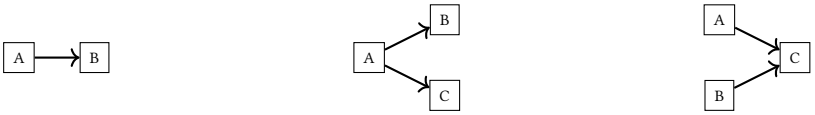
Beachte: Bei der Übersetzung zu C-Code sollte auf die korrekte Typisierung der Variable *out* geachtet werden.

11.3.2. Programmablaufplan

Ein Programmablaufplan ist eine grafische Darstellung bestehend aus Pfeilen und sogenannten *Knoten*. Ein Knoten ist „nur ein Kästchen“ und gibt bestimmte Objekte an. Jeder Knoten besitzt eine Beschriftung innerhalb des Kastens: eine Bedingung, ein Stück Programmcode, etc.

Die Pfeile führen von einem Knoten zu einen oder mehreren anderen Knoten. Ein Knoten kann dabei mehrere eingehende und ausgehende Pfeile besitzen.

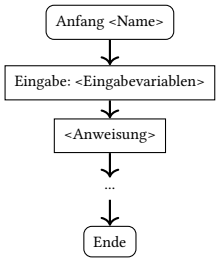
Beispiel:



Beginn, Ende, Name und Eingabe
Auch bei einem Programmablaufplan müssen Anfang und Ende einheitlich geregelt sein.

Für Start und Ende gilt, dass deren Knoten an den Ecken (sichtbar) abgerundet sein müssen. Alle anderen Knoten sind (sichtbar!, hinsichtlich der Klausur) eckig zu markieren.

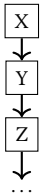
Mithilfe von Pfeilen wird die Abarbeitungsreihenfolge dargestellt. Das bedeutet, es folgt zuerst der Anfang, gefolgt von der Eingabe. Danach die `<Anweisung>` und am Schluss das Ende.



Eingabevariablen = Parameter

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

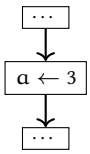
Sequenz von Anweisungen X, Y, Z, ...
Wie oben schon angesprochen, geben Pfeile die Abarbeitungsreihenfolge sowie die Sequenz an, in der ein Programm abgearbeitet wird.




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

Wertzuweisungen

Um einer Variable einen Wert zuzuweisen wird im Programmablaufplan ebenso wie beim Pseudocode das Pfeilsymbol ← verwendet.

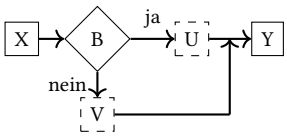



```
...
int a = 3;
...
```

Fallunterscheidungen: X,Y Anweisungen / U,V Teilmodelle

Für Fallunterscheidungen und Schleifen werden Bedingungen benötigt. Im Programmablaufplan sind diese Bedingungen in einem eigenen Knoten dargestellt, der Diamant-förmig ist. Die aus dem Bedingungs-knoten ausgehenden Pfeile besitzen eine Beschriftung, die die Fälle angibt.

Pfeile können auch zusammengefügt werden. Das dient der besseren Übersicht.

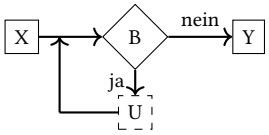



```
X;
if (B) { U }
else { V }
Y;
```

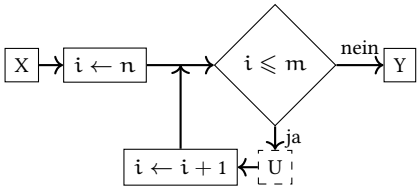
Wiederholungen: X,Y Anweisungen / U Teilmodell

Eigene Anweisungen für Schleifen existieren im Programmablaufplan nicht. Sie können aber geschickt mithilfe des Diamand-Knoten für Bedingungen und kluger Platzierung der Pfeile erstellt werden.

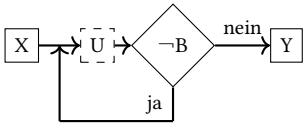
Auch hier gilt, dass die `break` und `continue` Befehle verwendet werden dürfen.




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



Beachten Sie wie beim Pseudocode auch beim Programmablaufplan, dass Felder in dieser abstrakten Darstellung i.d.R. bei 1 beginnen und Sie Ihre Schleifenbedingung und -zähler entsprechend formulieren müssen.



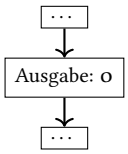

```
X;
do {
     U
} while (!B);
Y;
```

Während beim Pseudocode die Logik der Bedingung genau umgekehrt zum C-Programm war, entspricht sie hier 1:1 der des C-Programms: Solange `B` **nicht** erfüllt ist, wird der Anweisungsblock `U` wiederholt.

Einwertige Ausgabe

Die Ausgabe ist im Programmablaufplan, ähnlich wie im Pseudocode, immer eine eigene Anweisung.

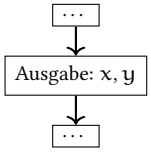
Gibt es **genau eine Ausgabevariable im Algorithmus** (**einwertige Ausgabe**), dann wird deren Wert in der C-Funktion mit einer `return`-Anweisung zurückgegeben.




```
...
return o;
...
```

Mehrwertige Ausgabe

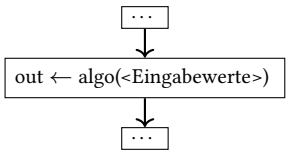
Gibt es **mehrere Ausgabevariablen im Algorithmus** (**mehrwertige Ausgabe**), dann werden die Ausgabewerte über adresswertige Eingabeparameter gespeichert.




```
void <Name>(T *x, T *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. (Der Name vom Algorithmus `algo` sowie der Variable `out` sind natürlich frei wählbar).




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

Beachte: Bei der Übersetzung zu C-Code sollte auf die korrekte Typisierung der Variable *out* geachtet werden.

11.3.3. Struktogramm

Ein Struktogramm wird als großes Rechteck dargestellt. Dieses Rechteck kann in weitere Abschnitte unterteilt werden. Diese Unterteilung stellt den Ablauf von oben nach unten sowie Programmcode etc. dar.

Beginn, Ende, Name und Eingabe

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

Algorithmus: <Name>

Eingabe: Eingabevariablen>
<Anweisung>
...

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

Wertzuweisungen

Um einer Variable einen Wert zuzuweisen wird im Struktogramm ebenso wie beim Pseudocode und Programmablaufplan das Pfeilsymbol ← verwendet.

...
a ← 3
...

```
...
int a = 3;
...
```

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

Sequenzen werden von oben nach unten in gleich breiten Blöcken angeordnet

X
Y
Z
...

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

Fallunterscheidungen: X,Y Anweisungen / U,V Teilmodelle

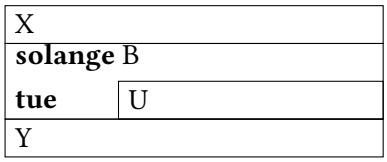
Teilmodelle können aus mehreren Anweisungen bestehen oder auch leer sein. Die Blöcke für U und V stehen unter dem Block für den jeweiligen Fall.

X	
ja B	nein
U	V
Y	

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

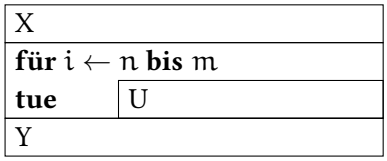
Wiederholungen: X,Y Anweisungen / U Teilmodell

In Struktogrammen können wir `while`, `for` und `do-while` Schleifen Darstellen. Dabei können die Teilmodelle aus mehreren Anweisungen bestehen oder auch leer sein. Wie in C dürfen in Wiederholungen die Anweisungen `break` und `continue` benutzt werden. Der Schleifen-Block für U wird in den Block für die Wiederholung verschachtelt:

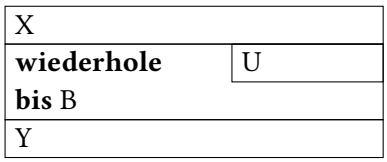



```
X;
while (B) {
    U
}
Y;
```

Wie bei den vorherigen Darstellungsformen beginnen Felder i.d.R. bei 1 und Sie müssen Ihre Schleifenbedingung und -zähler entsprechend formulieren.




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



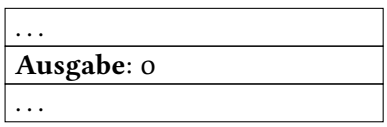

```
X;
do {
    U
} while (!B);
Y;
```

Wie beim Pseudocode ist die Logik beim Struktogramm wieder genau umgekehrt zu der in C: In C wird eine `do-while` Schleife durchlaufen, solange `B` **nicht** erfüllt ist. Im Struktogramm wird der Anweisungsblock `U` wiederholt, bis `B` **erstmalig** erfüllt ist.

Einwertige Ausgabe

Die Ausgabe ist im Struktogramm, ähnlich wie im Pseudocode oder Programmablaufplan, immer 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.




```
return o;
```

Mehrwertige Ausgabe
Die Ausgabe ist im Struktogramm, ähnlich wie im Pseudocode oder Programmablaufplan, immer **eine** eigene Anweisung.

Gibt es **mehrere Ausgabevariablen im Algorithmus** (**mehrwertige Ausgabe**), dann werden die Ausgabewerte über adresswertige Eingabeparameter gespeichert.

...
Ausgabe: x, y
...

```
void <Name>(T *x, T *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. (Der Name vom Algorithmus `algo` sowie der Variable `out` sind natürlich frei wählbar).

...
out ← algo(<Eingabewerte>)
...

```
...
out = algo(<Eingabewerte>)
...
```

Beachte: Bei der Übersetzung zu C-Code sollte auf die korrekte Typisierung der Variable *out* geachtet werden.

11.3.4. Beispiel: Binäre Suche

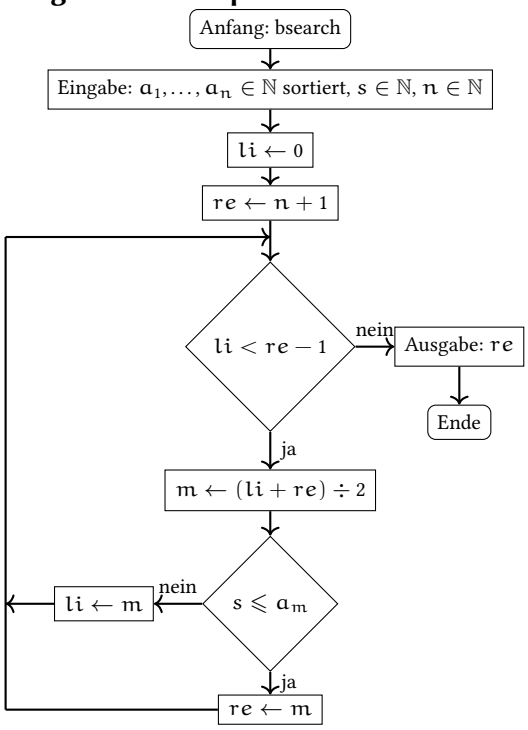
Problemstellung:
Finde für eine **aufsteigend sortierte** Zahlenfolge $a_1, \ldots, a_n$ und eine neue Zahl s die **Einfügeposition** i in $a_1, \ldots, a_n$, an der s unter Beibehaltung der Sortierung eingefügt werden kann:

Um dieses Problem zu lösen, nutzt man die sogenannte *binäre Suche*. Wie diese abläuft, entnehmen Sie den folgenden Darstellungen, die alle denselben Algorithmus widergeben.

Pseudocode

```
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
```

Programmablaufplan



Struktogramm

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	$m \leftarrow (li + re) \div 2$	
	ja — $s \leqslant a_m$ — nein	
	$re \leftarrow m$	$li \leftarrow m$
Ausgabe: re		

Die Idee hinter der binären Suche ist es, die sortierte Folge in zwei Hälften zu teilen. Anhand des Wertes in der Mitte wird geprüft, ob man in der linken oder rechten Hälfte weitersuchen muss, wo die Zahl eingefügt werden soll. Vor und nach jedem Schleifendurchlauf gilt: $a_{li} < s \leqslant a_{re}$. Die Folge wird in jedem Durchlauf halbiert und die Suche wird in der unteren oder oberen Teilfolge fortgesetzt.
Somit kommt man bei 1 000 000 Elementen auf 20 Durchläufe der Schleife.

> Beispiel: 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. $(li, re) = (0, 9)$, $m = 4$, $s \leqslant a_4$
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: $re = 4$

Aufgabe
Schreiben Sie in die freie Fläche neben den obigen Diagrammen den entsprechenden C-Code für die binäre Suche. Schauen Sie sich dazu genau an, wie der Algorithmus in den Diagrammen beschrieben ist und setzen Sie ihn 1:1 um. Wenn Sie fertig sind, können Sie Ihr Ergebnis mit der Lösung im Anhang auf Seite 25 vergleichen.

11.4. Nichtdeterministische Algorithmen

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**.

In einfachen Worten, über Anweisungen die nichtdeterministisch sind kann man nur (korrekte) Vermutungen äußern aber nicht eindeutig sagen, wie genau es weiter geht.

Beispiel:

Das obige Beispiel `bsearch` ist beispielsweise deterministisch. Jeder Schritt ist eindeutig und man weiß immer, welcher Schritt als nächstes folgt.

Ändert man es aber leicht um, wird es nichtdeterministisch. Dazu reicht es, eine Zufallsvariable mit einzubauen. Dadurch geht die Eindeutigkeit verloren.

```
1 Algorithmus: bsearchNdet
  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     Wähle m zufällig mit li < m < re;
6     wenn s ⩽ a_m dann
7         re ← m;
8     sonst
9         li ← m;
  Ausgabe: re
```

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 die wir kennengelernt hatten waren iterativ, z.B.:

- Berechnung des ggT
- Suchverfahren
- ...

Ein erstes Beispiel zur Rekursion haben wir in den Übungen kennengelernt, also die *Fibonacci-Zahlen* vorgestellt wurden. Diese sind induktiv definiert, d.h. eine Fibonacci-Zahl geht immer auf ihre beiden Vorgänger zurück.

11.5.1. Direkte Rekursion

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

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.

Beispiel:

Ein kleines Beispiel wäre:

```
int sub(int zahl)
{
    if (zahl == 0)
        return 0;
    else
        return 1 + sub(zahl - 1);
}

int main(void)
{
    sub(3);

    return 0;
}
```

Die Funktion sub ruft immer wieder sich selbst auf. Dabei wird der übergebene Parameter jedes mal kleiner. Wenn die Abbruchbedingung erfüllt ist, erfolgt kein weiterer Aufruf von sub, sondern die Lösung des Basisfalls wird zurückgegeben.

Beispiel: Binäre Suche

Betrachten wir wieder das Beispiel der binären Suche, hatten wir folgenden Pseudocode gegeben:

```
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
```

Diesen Algorithmus kann man auch mithilfe einer Rekursion beschreiben. Hierfür muss man jedoch den Aufbau des Algorithmus verändern.

```
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)
```

- Eingabe: Schranken für den Suchbereich li, re werden mit übergeben
- Zeile 2: Abbruchbedingung
- Zeile 5: Wenn die einzufügende Zahl kleiner ist als die Zahl in der Mitte, suche rekursiv in der linken Hälfte weiter.
- Zeile 6: Wenn die einzufügende Zahl größer ist als die Zahl in der Mitte, suche rekursiv in der rechten Hälfte weiter.
- **Erster Aufruf**: bsearchRek$(a_1, \ldots, a_n, s, 0, n + 1)$

Beispiel: 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.

Induktive Definitionen können direkt in rekursive Algorithmen umgesetzt werden

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

- 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**)

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

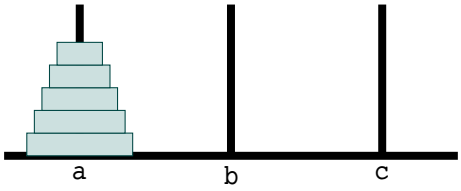
```
unsigned long int fakultaet(unsigned int n)
{
    if (n == 0)
        return 1L;
    else
        return n * fakultaet(n-1);
}
```

Beispiel: Die Türme von Hanoi

Ein beliebtes Beispiel für Rekursion sind die Türme von Hanoi. Das Spiel wurde 1883 vom französichen Mathematiker Édouard Lucas erfunden. Dazu gab es die Geschichte, dass die Mönche in einem indischen Tempel einen Turm aus 64 goldenen Scheiben versetzen müssten, wobei die größte Scheibe ganz unten liegt und die kleinste ganz oben. Eine kleinere Version des Problems sehen sie auf dem Bild unten.

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.



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);
    }
}
```

Aufruf in der `main`:
`hanoi(`Anzahl der Scheiben, Startstab, Zielstab, Hilfsstab`);`

Eine schöne Illustration der Abarbeitungsreihenfolge befindet sich im Foliensatz.

Direkte Rekursion: 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 $F(EP_1)$,...,$F(EP_n)$ werden also insgesamt n Stack Frame erzeugt und die Abarbeitung erfolgt rückwärts in der Reihenfolge $F(EP_n)$,...,$F(EP_1)$.

Im Foliensatz ab Folie 38 befindet sich ein anschauliches Beispiel für die Abarbeitung im Stack. Es wurde nicht in das Skript übernommen, da das Beispiel nur im Foliensatz funktioniert und die Veranschaulichung kaputt gehen würde.

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. Wenn der Compiler den Code optimiert und Rekursion erkennt, kann er daraus aber auch sehr effizienten Maschinencode generieren. Details dazu folgen in der Vorlesung *Systemnahe Informatik*.

Rekursive Algorithmen spielen eine sehr wichtige Rolle bei der Verwendung von Bäumen als Datenstrukturen (siehe später und Vorlesung *Informatik 3*).

11.5.2. Indirekte Rekursion

Indirekte Rekursion:
Man spricht von **indirekter Rekursion**, wenn es mehrere Algorithmen $A_1, ..., A_n$ $(n > 1)$ 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:

Ein kleines Beispiel wäre:

```
int f1(int zahl)
{
    ...
    f2(zahl);
    ...
}

int f2(int zahl)
{
    ...
    f1(zahl - 1);
    ...
}

int main(void)
{
    f1(5);

    return 0;
}
```

Beispiel: Test auf gerade / ungerade Zahl

```
1 Algorithmus: isEven
  Eingabe: n ∈ ℕ₀
2 wenn n = 0 dann
  |  Ausgabe: 1
3 sonst
  |  Ausgabe: isOdd(n − 1)
```

```
1 Algorithmus: isOdd
  Eingabe: n ∈ ℕ₀
2 wenn n = 0 dann
  |  Ausgabe: 0
3 sonst
  |  Ausgabe: isEven(n − 1)
```

Hier rufen sich die beiden Funktionen `isEven` und `isOdd` gegenseitig auf. Die Funktion, bei der die Zahl den Wert 0 erreicht, veranlasst die Ausgabe und zeigt damit an, ob die Zahl gerade oder ungerade ist. Probieren Sie es selbst mit einem Beispiel aus.

Endständig rekursive Algorithmen

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:

```
1 Algorithmus: algo
  Eingabe: x
2 wenn B(x) dann
  |  Ausgabe: algo(E(x))
3 sonst
  |  Ausgabe: A(x)
```

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

Beispiel: Berechnung des Rests bei ganzzahliger Division (modulo)

```
1 Algorithmus: mod
  Eingabe: x ∈ ℕ₀, y ∈ ℕ
2 wenn x ⩾ y dann
  |  Ausgabe: mod(x − y, y)
3 sonst
  |  Ausgabe: x
```

Ausführung endständig rekursiver Funktionen

Für endständige rekursive Algorithmen / Funktionen ist eine **direkte Umschreibung in eine iterative Form** möglich. Optimierende C-Compiler können endständig rekursive Algorithmen relativ gut erkennen und schreiben den Maschinencode dann meist in eine iterative Form um. Diese kann sogar noch optimaler sein als eine direkte iterative Implementierung, z.B. wenn der C-Compiler erkennt, dass die Iterationen voneinander unabhängig sind.

Übersetzung in iterative Form

```
1 Algorithmus: algo
  Eingabe: x
2 wenn B(x) dann
  |  Ausgabe: algo(E(x))
3 sonst
  |  Ausgabe: A(x)
```

```
1 Algorithmus: algo
  Eingabe: x
2 solange B(x) tue
3 |  x ← E(x);
  Ausgabe: A(x)
```

Unser Beispiel von oben, welches den Rest der ganzzahligen Division berechnet, lässt sich auch leicht in eine iterative Form überführen:

Beispiel:

```
1 Algorithmus: mod
  Eingabe: x ∈ ℕ0, y ∈ ℕ
2 solange x ⩾ y tue
3 |  x ← x − y;
  Ausgabe: x
```

Fazit

Nun sollten Sie in der Lage sein, die verschiedenen Darstellungsformen für Algorithmen zu notieren und zu lesen. Außerdem sollten Sie eine Darstellungsform in eine weitere übersetzen können, z.B. aus Pseudocode ein Struktogramm zu erstellen. Zusätzlich soll eine Übersetzung von und zu C-Code möglich sein, z.B. sollten Sie einen Programmablaufplan zu einem C-Programm und andersrum erstellen können.

Sie wundern sich vermutlich, warum wir Ihnen keine Anleitung zum Erstellen von Lösungswegen zu Problemem bereitstellen. Jedoch gibt es im Grunde kein Allgemeinrezept zum Finden von Algorithmen bzw. Lösungsstrategien. Oft gibt es viele verschiedene Lösungsansätze für ein und dasselbe Problem. Benötigt werden meist nur sorgfältiges analytisches und logisches Denken gepaart mit ein wenig Intuition. Auch kann die Analyse eines verwandten Problems eine gute Ausgangsposition für einen Algorithmus sein.

A. 1:1 Umsetzung der binären Suche

Die 1:1 Umsetzung der Darstellung der binären Suche von Seite 15 sieht wie folgt aus:

```
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;
}
```

Haben Sie `li` und `re` passend initialisiert? `li` sollte 1 kleiner sein als der erste Eintrag des Feldes und `re` 1 größer als der letzte Eintrag des Feldes.