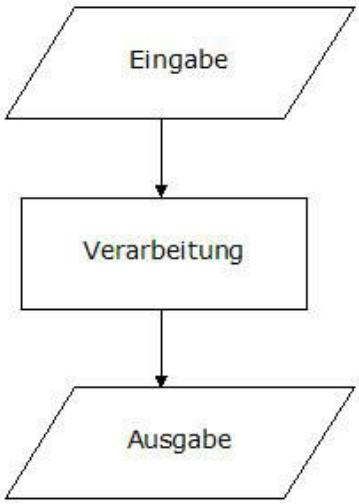
Informatik

inf11 06.02.2023

Definition:
Unter einem **Algorithmus** verstehen wir ein aus endlich vielen Anweisungen bestehendes allgemeines Verfahren, welches eine Klasse von Problemen in endlich vielen Schritten löst.

Ein Algorithmus läßt sich, unabhängig von der jeweils verwendeten Programmiersprache, als Flußdiagramm (später auch: Struktogramm) darstellen und verdeutlichen.



1. Lineare Algorithmen

Unter einem **linearen Algorithmus** verstehen wir einen Algorithmus, bei dem die nacheinander auszuführenden Anweisungen sich längs eines einzigen Pfades aneinanderreihen; insbesondere enthält ein linearer Algorithmus keine Programmverzweigungen.

Aufgabe 1 (Quaderberechnung)
Eingabedaten: Länge a, Breite b, Höhe c
Verarbeitung: Berechnung des Volumens V und der Oberfläche O
Ausgabe: Volumen V, Oberfläche O

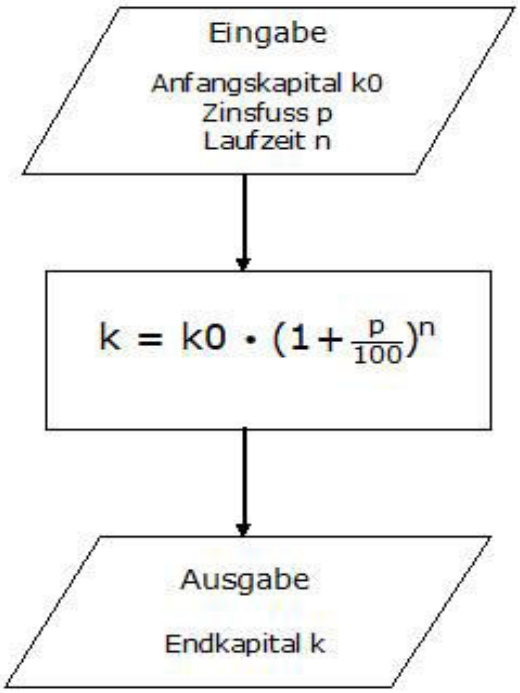
Aufgabe 2 (Zinseszins)
Wenn ein Anfangskapital **k0** zu einem jährlichen Zinssatz **p** % über einen Zeitraum von **n** Jahren mit Zinseszins angelegt wird (der Zinsbetrag wird also am Ende eines jeden Jahres dem zu verzinsenden Kapital zugeschlagen), ermittelt der Algorithmus „Zinseszins" das Endkapital **k** nach **n** Jahren.
(Bemerkung: In entsprechender Weise läßt sich die Entwicklung des Preisindex nach n Jahren bestimmen, wenn die jährliche Inflationsrate p % beträgt.)

Aufgabe 3 („Promillerechner")
Dieser Algorithmus ermittelt einen groben Schätzwert für die Blutalkoholkonzentration.
Eingabedaten:
V = Volumen des Getränks in Litern
p = Volumenanteil in % des Alkohols im Getränk
m = Gewicht (Masse) der Person in kg
Ausgabedaten:
K = Blutalkohol-Konzentration in Promille

Berechnungsvorschrift: $K = 10 \cdot V \cdot p / (m \cdot 0.7)$

Lösung zu **Aufgabe 2**

Flußdiagramm:



Programmtext in Python:

zinseszins.py - F:/Informatik_2022-23/info11/zinseszins.py (3.8.6)

File Edit Format Run Options Window Help

```
# Zinseszins

# Mit '#' eingeleitete Zeilen bilden einen Kommentar,
# der auf den Programmablauf keinen Einfluß hat.

# Eingabe

k0=float(input('Anfangskapital =  '))
p=float(input('Zinsfuß         =  '))
n=float(input('Laufzeit        =  '))

# Verarbeitung

k=k0*(1+p/100)**n

# Ausgabe

# Die folgende Anweisung erzeugt eine Leerzeile,
# ist aber auch verzichtbar.
print ()
# Ausgabe des Ergebnisses auf viele Nachkommastellen
print ('Kapital nach ',n,' Jahren = ',k)
# Runden des Ergebnisses auf 2 Nachkommastellen und Ausgabe
print ('Kapital nach ',n,' Jahren = ',round(k, 2),' Euro')
```

Wenn wir nach Eingabe des Programmtextes im „IDLE"-Editor den Button „Run" anklicken, öffnet sich ein Kontextmenue, und wir starten das Programm durch Klick auf „Run Module".
Nachdem man bestätigt hat, den eventuell geänderten Programmtext zu speichern, öffnet sich die „Python Shell", in der man die Eingaben macht und in der dann die Ausgabe des Ergebnisses oder der Ergebnisse erfolgt.

Definition: Ein **Anweisungsblock** besteht aus einer Folge zusammengehörender Anweisungen, die nacheinander ausgeführt werden.
Ein Anweisungsblock, der innerhalb einer Schleife wiederholt wird, heißt **Schleifenrumpf**.
Den zu einer Funktion gehörenden Anweisungsblock nennen wir auch **Funktionsrumpf**.

Bemerkungen:
- Anweisungsblöcke können auch ineinander verschachtelt sein.
- In Python wird ein Anweisungsblock durch Einrücken des Programmtextes gekennzeichnet (hier ist also auf die korrekte Formatierung des Programmtextes zu achten!).

2. Verzweigte Algorithmen

Ein **verzweigter Algorithmus** enthält mindestens eine Fallunterscheidung, so daß je nach Ausgang der Fallunterscheidung verschiedene Anweisungsblöcke durchlaufen werden.

Aufgabe 4

Mit dem Body-Mass-Index (**BMI**) kann man abschätzen, ob jemand Normalgewicht hat. Der BMI ist eine dimensionslose Zahl (also ohne Maßeinheit) und berechnet sich wie folgt:
BMI = gewicht / (groesse * groesse)
mit
gewicht = Maßzahl der Masse in kg
groesse = Maßzahl der Körpergröße in m

Beispiel:
Mit Masse = 70 kg und Körpergröße = 1,80 m erhält man
BMI = 70 / (1,80 * 1,80) = 70 / 3,24 ≈ 21,6.

Für BMI < 19 gilt man als untergewichtig, für BMI > 26 als übergewichtig; Normalgewicht verbindet man mit $19 \leq BMI \leq 26$.

Der Algorithmus BodyMassIndex soll folgendes leisten:
Nach Eingabe des Gewichts (in kg) und der Größe (in m) wird BMI (auf eine Dezimale gerundet) berechnet und ausgegeben, darüberhinaus erfolgt die Information, ob man als normal-, unter- oder übergewichtig gilt.

Konzipiere ein

a) Flußdiagramm, b) Struktogramm, c) Python-Programm!

Aufgabe 5 (Mobilfunkrechnung)

Der Betreiber eines Mobilfunknetzes hat folgende Tarifgestaltung:

Monatliche Grundgebühr (einschließlich 100 Gesprächsminuten): 8 €;
für die nächsten, über 100 Minuten hinausgehenden Gesprächsminuten sind 5 ct je Minute zu entrichten.

Formuliere einen Algorithmus als
Flußdiagramm, Struktogramm, Pythonprogramm,
um nach Eingabe der Anzahl **x** der monatlichen Gesprächsminuten den Rechnungsbetrag **b** zu bestimmen.

Informatik 11 27.02.2023

Numerischen Datentypen: **float** und **integer**
(float: Gleitkommazahlen oder Dezimalzahlen; integer: ganze Zahlen)

```
>>> print(11 / 6)          Quotient zweier ganzer Zahlen
1.8333333333333333

>>> print(2 ** 0.5)        Wurzel aus 2
1.4142135623730951

>>> print(27 / 4)          Quotient zweier ganzer Zahlen
6.75

>>> print(27 // 4)         ganzzahliger Quotient (27 : 4 = 6 Rest 3)
6

>>> print(27 % 4)          Rest bei ganzzahliger Division
3

>>> print(7 * 12)          Produkt ganzer Zahlen
84

>>> print(0.8 * (-7.5))    Produkt zweier Kommazahlen
-6.0
```

Datentyp **boolean**

Eine Boolesche Variable oder ein Boolescher Ausdruck (Term) nimmt nur zwei Werte an:
True oder **False**
(abkürzend: 1 oder 0, ja oder nein; in Python sind **True** oder **False** zu verwenden!)

Insbesondere sind folgende Terme Boolesche Ausdrücke, deren Wert sich auch einer Variablen zuweisen läßt:

8 > 5 hat den Wert **True**
7 == 8 hat den Wert **False**
7 != 8 hat den Wert **True**
x hat den Wert **True** nach der Wertzuweisung x = 7 < 12
x hat den Wert **False** nach der Wertzuweisung x = (0 == 6)

Wir definieren die Verknüpfungen **and** und **or** sowie die Operation **not** jeweils über eine Wahrheitstafel:

a	b	a **or** b
False	**False**	**False**
False	**True**	**True**
True	**False**	**True**
True	**True**	**True**

a	b	a **and** b
False	**False**	**False**
False	**True**	**False**
True	**False**	**False**
True	**True**	**True**

a	**not** a
False	**True**
True	**False**

Datentyp **character** (Zeichen)

```
>>> x = 'a'
>>> print(x)
a
```

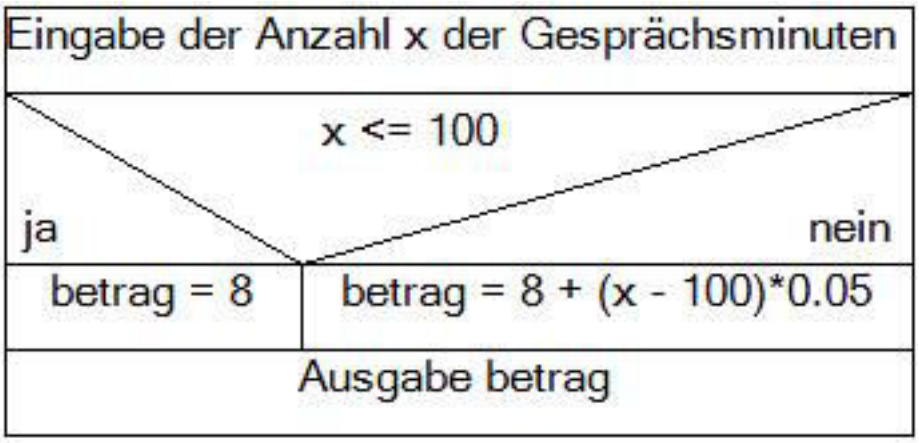
```
>>> zeichen = '&'
>>> print(zeichen)
&
```

Datentyp **string** (Zeichenkette)

```
>>> name = 'Kopernikus'
>>> print(name)
Kopernikus
```

Lösung zu **Aufgabe 5**:

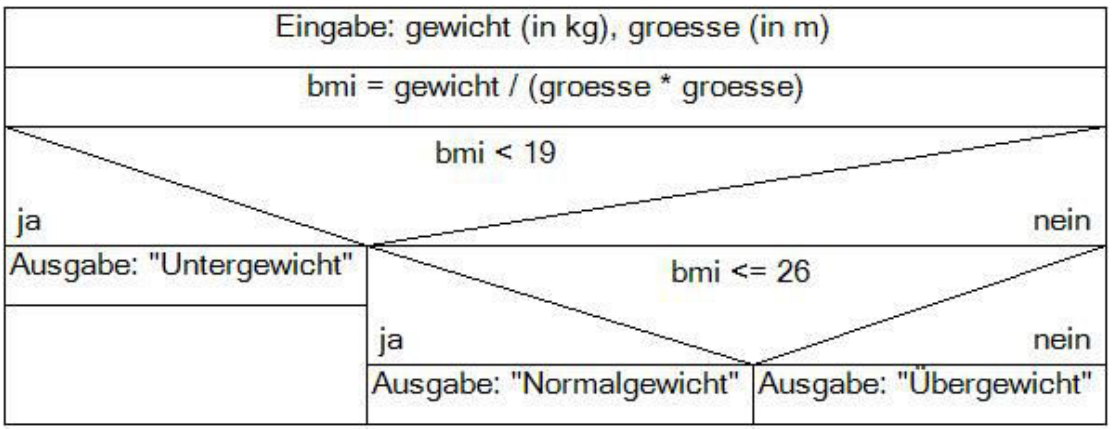
a) Struktogramm



b) Python-Quelltext:

```
# Monatsrechnung Mobilfunk
# Verzweigter Algorithmus
# Aufgabe Nr. 5

'Eingabe'
x = int(input('Anzahl der Gespraechsminuten = '))

'Verarbeitung'
if x <= 100:
    betrag = 8
else:
    betrag = 8 + (x - 100)*0.05

'Ausgabe'
print ()
print ('Rechnungsbetrag bei',x,'Minuten:',betrag,'Euro')
```

Struktogramm zu **Aufgabe 4**:



Arbeitsauftrag: Schreibe einen Python-Quelltext und teste das Programm!

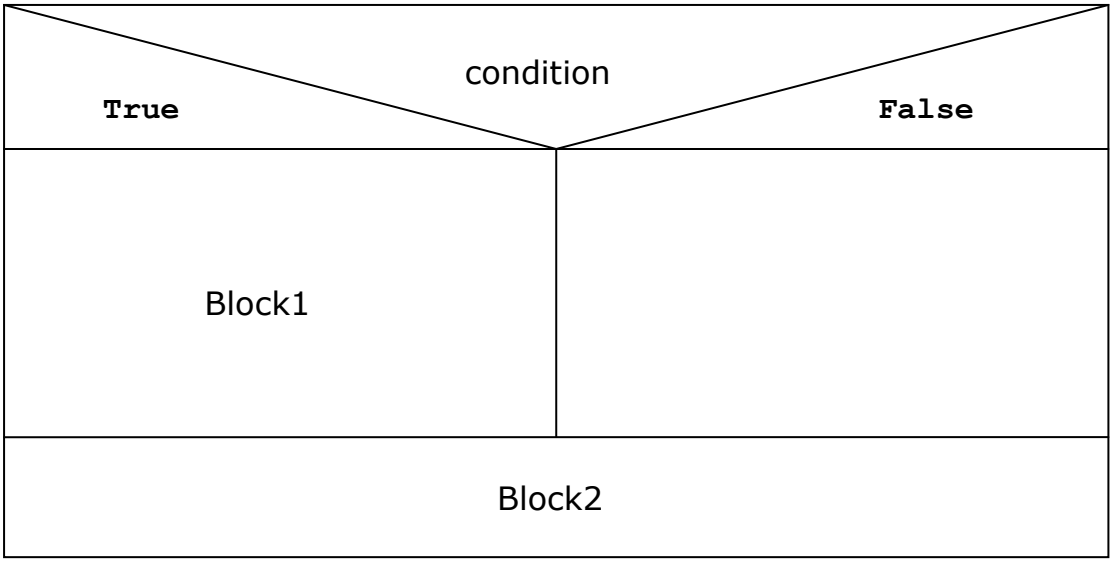
Aufgabe 6
Gegeben sei folgender Telefontarif:
Monatliche Grundgebühr: 8 € (einschließlich 100 Gesprächsminuten). Für die über das Freikontingent hinausgehenden nächsten 200 Minuten werden 3 ct/min berechnet, darüberhinausgehende Minuten kosten 5 ct/min.
Formuliere den Algorithmus als Struktogramm und Python-Programm.

Zusammenfassung: Verzweigte Algorithmen

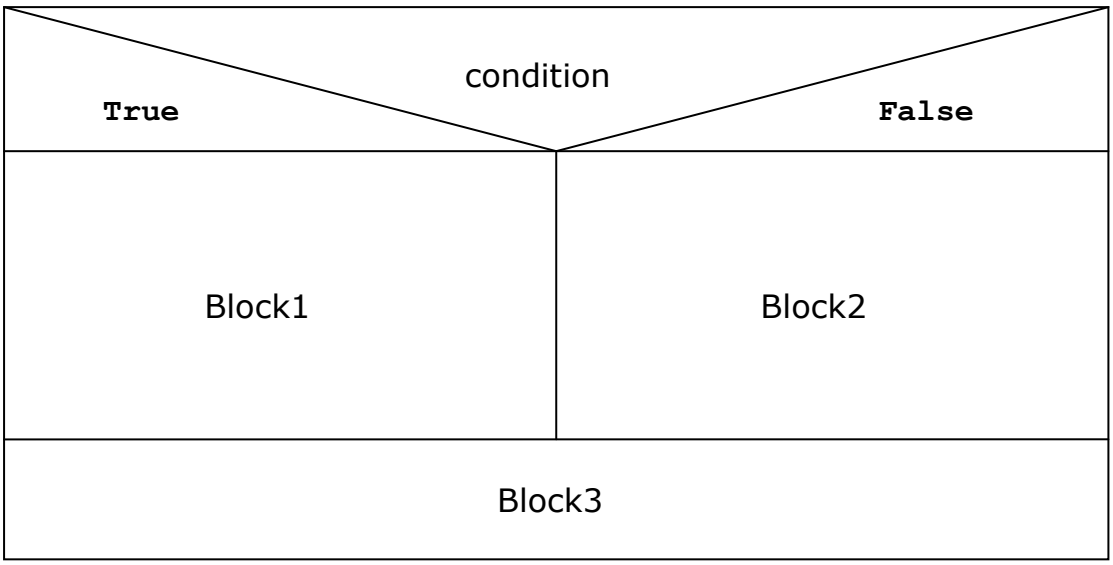
Beachte: In Python wird ein Anweisungsblock durch Einrücken des Programmtextes gekennzeichnet.

Im folgenden verstehen wir unter ***condition*** einen Booleschen Term (der auch nur aus einer Booleschen Variablen bestehen kann), der die Werte **`True`** oder **`False`** annimmt. In Struktogrammen kennzeichnen wir **`True`** auch durch , + ' oder , ja ', **`False`** durch , – ' oder , nein '.

Einseitige Auswahl



Zweiseitige Auswahl

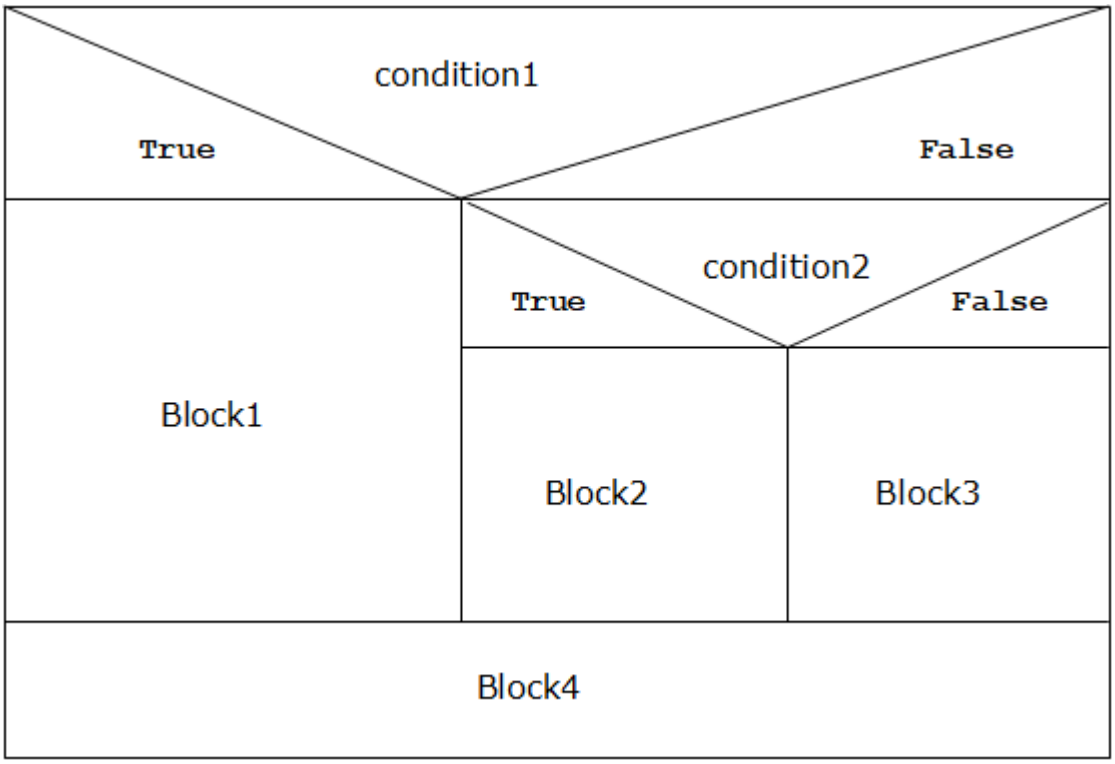


Formulierung in Python:

```
if condition:
    Block1
Block2
```

```
if condition:
    Block1
else:
    Block2
Block3
```

Mehrstufige Auswahl



Formulierung in Python:

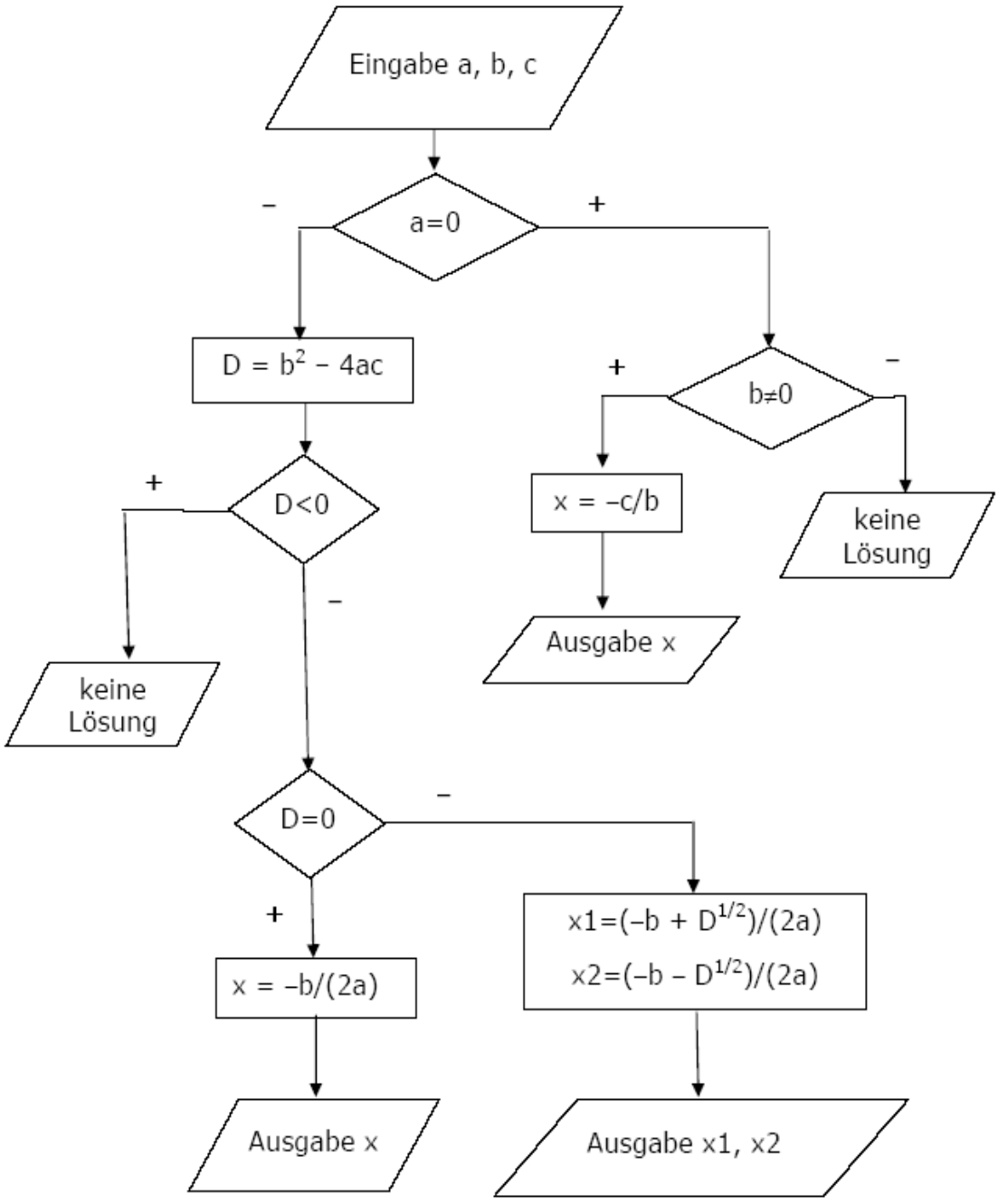
```
if condition1:
    Block1
else:
   if condition2:
       Block2
   else:
       Block3
Block4
```

```
if condition1:
    Block1
elif condition2:
    Block2
else:
    Block3
Block4
```

Aufgabe 7 (Quadratische Gleichungen)

Spezifikation des Algorithmus QuadEquation**:**
Nach Eingabe der Koeffizienten a, b, c der allgemeinen quadratischen Gleichung **$ax^2 + bx + c = 0$** ermittelt der QuadEquation die Lösungsmenge und gibt diese aus.

Flußdiagramm:



a) Erstelle ein Struktogramm.

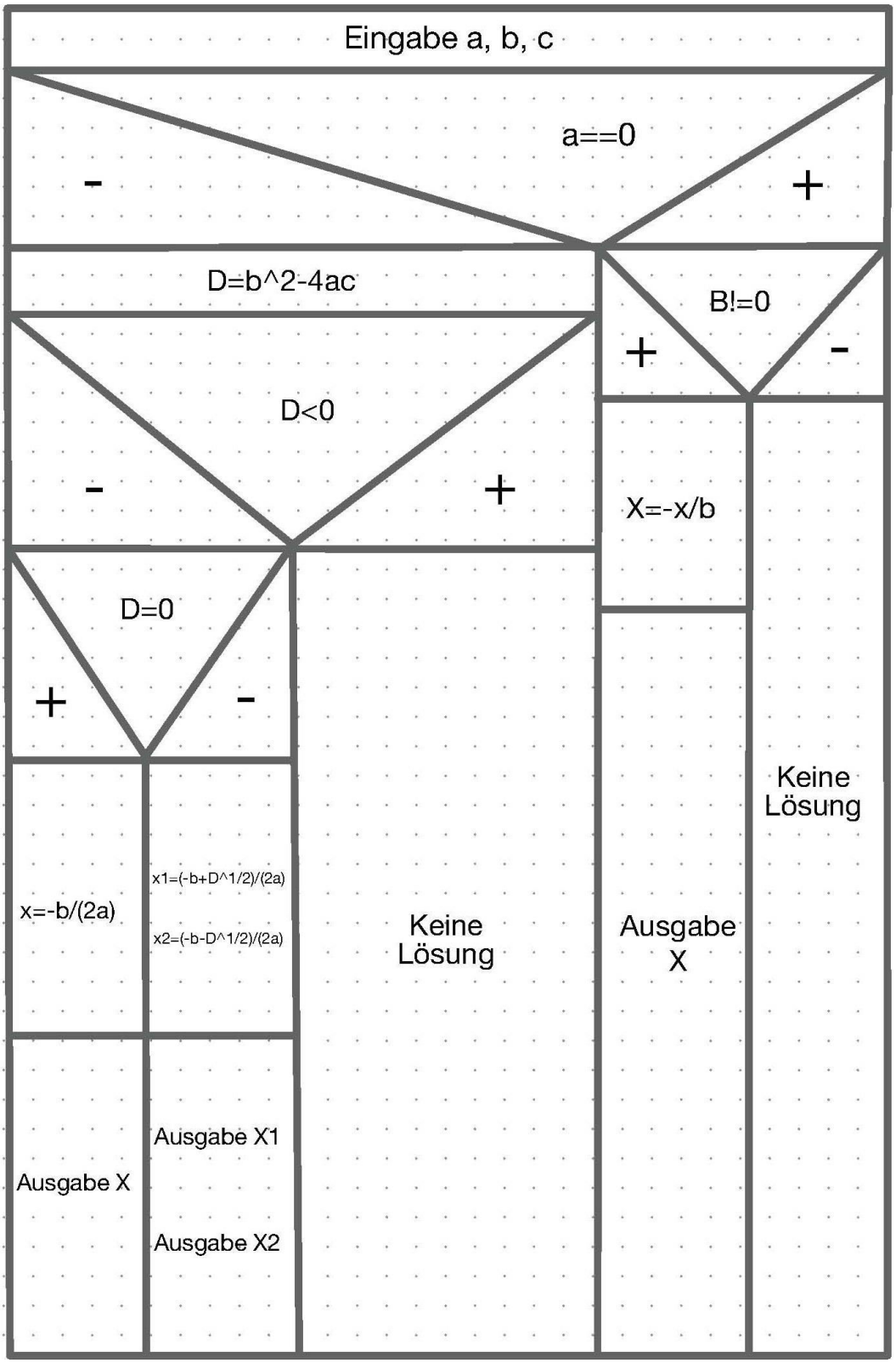
b) Schreibe und teste ein Python-Programm.

Informatik

inf11 10.03.2023

Lösung zu **Aufgabe 7**:

a) Struktogramm (Jakob)



Aufgabe:
Man überzeuge sich, daß das folgende Python-Programm gemäß obenstehendem Struktrogramm aufgebaut ist.

b) Python-Quelltext:

```
# Quadratische Gleichungen

'Eingabe der Koeffizienten'

print ("Quadratische Gleichung:  a*x^2 + b*x + c = 0")
a=float(input("a = "))
b=float(input("b = "))
c=float(input("c = "))
print()
print ("Loesungsmenge:")

'Verarbeitung der Daten und Ausgabe der Loesungen'

if a == 0:
   print ("Gleichung nicht quadratisch")
   if b!=0:
     print ("x =",-c/b)
   else:
     print ("keine Loesung!")

else:
   D = b**2 - 4*a*c

   if D < 0:
      print ("keine Loesung!")

   else:
      if D == 0:
         x = -b/(2*a)
         print ("x =",x)

      else:
         x1 = (-b + D**(1/2))/(2*a)
         x2 = (-b - D**(1/2))/(2*a)
         print ("x1 =",x1)
         print ("x2 =",x2)
```

3. Algorithmen mit Wiederholungen

Wenn ein Anweisungsblock innerhalb eines Algorithmus wiederholt auszuführen ist, verwenden wir eine Schleife (engl.: loop) als Kontrollstruktur; der zu wiederholende Anweisungsblock heißt auch Schleifenrumpf.

Die Programmiersprache Python kennt die (kopfgesteuerte) while-Schleife und die for-Schleife; in anderen Sprachen (z. B. Java, Pascal, C++) sind auch auch fußgesteuerte Schleifen (repeat-Schleife) implementiert.

while-Schleife

Syntax einer **while**-Schleife in Python:

```
while condition:                while condition:
        Anweisung1                      A
        Anweisung2
        Anweisung3
```

Dabei ist `condition` ein Boolescher Term; der aus einer Anweisung oder mehreren Anweisungen bestehende Schleifenrumpf **A** wird nur dann ausgeführt, falls `condition` den Wert `True` hat.

Beachte: Der Schleifenrumpf ist durch Einrücken des Programmtextes kenntlich zu machen!

Aufgabe 8 (Quadratzahltabelle)

Formuliere einen Algorithmus, welcher nach Eingabe einer natürlichen Zahl n die Quadrate der Zahlen 1, ... , n berechnet und ausgibt.

```
n = int(input('n = '))
```
Eingabe n

```
i = 1
```
Zuweisung eines Anfangswerts an die Zählvariable i (oder: Schleifenindex i)

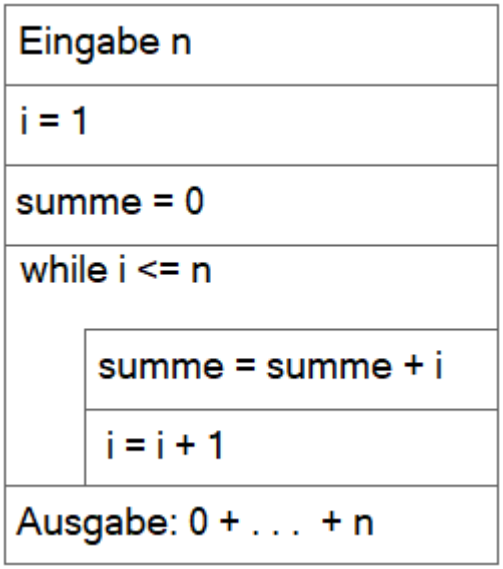
```
while i <= n:

    q = i * i
    print(i,'^2 =',q)
    i = i + 1
```
Schleifenrumpf

Aufgabe 9

Formuliere einen Algorithmus a) als Struktogramm, b) als Python-Programm, welcher nach Eingabe einer natürlichen Zahl n, $n \geq 0$, die Summe der Zahlen 0, ... , n berechnet und ausgibt.

Lösung:

a) Struktogramm:



Initialisierung des Schleifenindex i

Initialisierung der Variablen summe

Abfrage des Booleschen Terms i <= n

Schleifenrumpf

b) Quellcode in Python:

```
# Nach Eingabe einer natürlichen Zahl n wird die
# Summe der Zahlen 0, . . . , n berechnet und ausgegeben

n = int(input('n = '))

'Initialisierung des Schleifenindex i'
i = 1

'Initialisierung der Variablen summe'
summe = 0

while i <= n:
    summe = summe + i
    i = i + 1

print('Summe der Zahlen 0 , . . . ,',n,' =',summe)
```

Bemerkungen:

- Der Schleifenindex i heißt auch Zählindex oder Zähler.
- Da die als Boolescher Term formulierte Bedingung (hier: i <= n) vor Eintritt in den Schleifenrumpf abgefragt wird, handelt es sich bei der while-Schleife um eine kopfgesteuerte Schleife.

Aufgaben:

- Schreibe obenstehendes Struktogramm als Flußdiagramm.
- Verfolge gedanklich die Arbeitsschritte, die der Algorithmus nach der Eingabe von 0, 1, 2, 3 jeweils ausführt.

Trace

Anhand einer Trace-Tabelle können wir für bestimmter Eingabewerte überprüfen, ob der Algorithmus das Verlangte leistet; ein Trace liefert somit eine erste Information darüber, ob der Algorithmus korrekt ist.
Beachte: Ein Trace ersetzt nicht einen allgemeinen Korrektheitsbeweis.

In einer Tabelle notieren wir die Werte aller Variablen, Konstanten und Booleschen Terme (wir verwenden dieselben Variablennamen wie in obenstehendem Struktogramm). Solange der Term **`i <= n`** den Wert **`True`** hat, erfolgt ein weiterer Schleifendurchlauf; der Algorithmus bricht ab, sobald der Term **`i <= n`** den Wert **`False`** annimmt, denn es gibt dann keinen weiteren Schleifendurchlauf. Nach Abbruch wird der Wert der Variablen **`summe`** (hier: 15) ausgegeben.

Trace für den Eingabewert **n = 5**
Abkürzung: **SD** = Schleifendurchlauf

	n	**i**	**summe**	**i <= n**
vor dem 1. SD	5	1	0	True
vor dem 2. SD	5	2	1	True
vor dem 3. SD	5	3	3	True
vor dem 4. SD	5	4	6	True
vor dem 5. SD	5	5	10	True
nach dem 5. SD	5	6	15	False

Aufgabe 10

Formuliere den Algorithmus **FAKULTÄT** unter Verwendung einer while-Schleife
a) als Struktogramm,
b) als Python-Programm,
welcher nach Eingabe einer natürlichen Zahl n, $n \geq 1$, das Produkt der Zahlen 1, ... , n berechnet und ausgibt.

Anmerkung:
Man schreibt auch $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ (Lies: n-Fakultät).

Beispiele: $6! = 720$ $13! = 6\,227\,020\,800$

Ergänzung:
Definitionsgemäß gilt: 0! = 1. Erweitere den Algorithmus so, daß für die Eingabe n = 0 der Wert 1 ausgegeben wird.

c) Erstelle eine Trace-Tabelle für n = 5.

Informatik 11

20.03.2023

range-Anweisung

Die **range**-Anweisung definiert einen Bereich ganzer Zahlen.

range(10)	definiert den Bereich 0, 1, . . . , 9
range(4,21)	definiert den Bereich 4, 5, . . . , 20
range(4,21,3)	definiert den Bereich 4, 7, 10, . . . , 16, 19
range(-4,3)	definiert den Bereich -4, -3, -2, -1, 0, 1, 2

Allgemein gilt:

range(start, stop)
definiert den Bereich **start**, ... , **stop-1** ganzer Zahlen,

range(start, stop, step)
definiert den Bereich **start**, . . . , **stop-1** mit der Schrittweite **step**.

Erstellen einer Liste ganzer Zahlen

a = list(range(4,13)) erzeugt die Liste
[4, 5, 6, 7, 8, 9, 10, 11, 12];
die (in diesem Fall 9) Elemente dieser Liste nennen wir auch Komponenten, auf die man mit **a[0], a[1], . . . , a[8]** zugreifen kann.

*Bemerkung: Unter einem **Feld** oder einem **array** verstehen wir eine Folge von Variablen gleichen Typs; mit vorstehendem Beispiel haben wir also ein array **a** ganzer Zahlen erzeugt mit den Komponenten a[0], a[1], . . . , a[8].*

Beispiele (ausgeführt in der Python-shell):

```
>>> list(range(1,9))
[1, 2, 3, 4, 5, 6, 7, 8]
>>> a = list(range(-5,25,3))
>>> print(a)
[-5, -2, 1, 4, 7, 10, 13, 16, 19, 22]
>>> print(a[0])
-5
>>> print(a[3])
4
>>> print(a[9])
22
```

For-Schleife

Das Python-Programm

```
n = int(input('n = '))
for i in range(1,n):
    print(i)
    print(i*i)
```

liefert nach Eingabe der natürlichen Zahl n die Zahlen 1, 2, . . . , n-1 und deren Quadrate; probiert es aus!

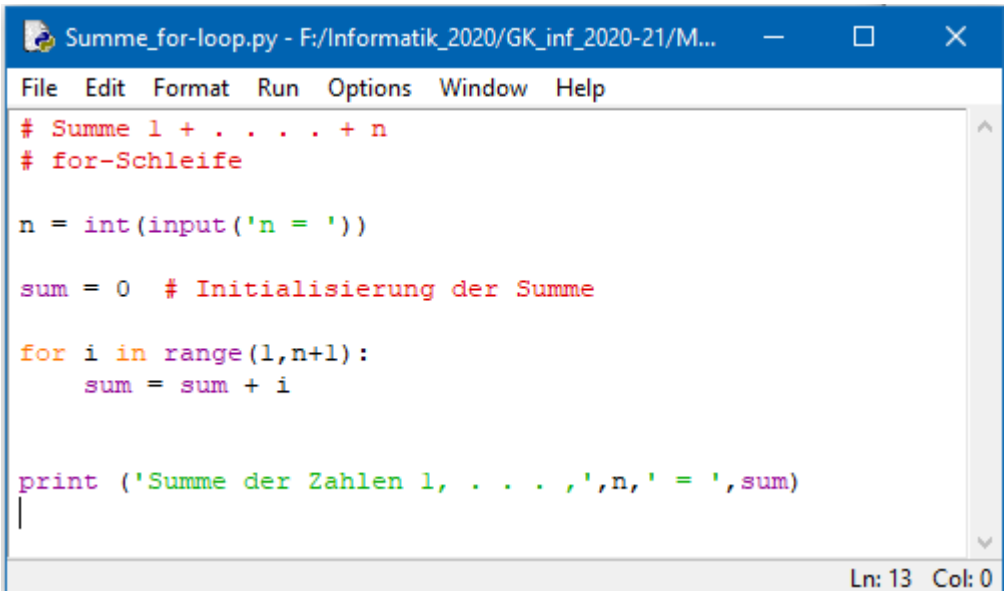
Syntax einer **for**-Schleife in Python:

```
for i in range(start, stop):
          Anweisung1
          Anweisung2
          Anweisung3
```

```
for i in range(start, stop):
                    A
```

Arbeitsaufträge:

1. Schreibe und teste ein Python-Programm, um die Siebener-Reihe auszugeben (also die Zahlen 7, 14, 21, . . .).
2. Erstelle (siehe screenshot) den Python-Programmtext zur Berechnung der Summe `1 + . . . + n` und teste das Programm mit unterschiedlichen Eingaben.



Aufgabe 11

Formuliere den Algorithmus **FAKULTÄT**, der nach Eingabe einer natürlichen Zahl **n**, $n \geq 0$, den Wert **n!** liefert, unter Verwendung einer **for**-Schleife als
a) Struktogramm,
b) Python-Programm!
c) Erstelle eine Trace-Tabelle für n = 4.

Aufgabe 12

Formuliere und teste ein Python-Programm, welches nach Eingabe der natürlichen Zahl k, $k \geq 1$, die Summe der ersten k ungeraden natürlichen Zahlen bestimmt, und zwar unter Verwendung einer for-Schleife.

Informatik

inf11 13.04.2023

Lösungen zu **Aufgabe 12**

a) **while**-Schleife

Struktogramm:

Eingabe k
summe = 1
summand = 1
i = 2
while i <= k
summand = summand + 2
summe = summe + summand
i = i + 1
Ausgabe summe

Trace für k = 6:

	k	**i**	**summand**	**summe**	**i <= k**
vor dem 1. SD	6	2	1	1	True
vor dem 2. SD	6	3	3	4	True
vor dem 3. SD	6	4	5	9	True
vor dem 4. SD	6	5	7	16	True
vor dem 5. SD	6	6	9	25	True
nach dem 5. SD	6	7	11	36	False

Quellcode Python:

```
k = int(input('k = '))

summe = 1
summand = 1
i = 2

while i <= k:
    summand = summand + 2
    summe = summe + summand
    i = i + 1

print('Summe der ersten',k,'ungeraden Zahlen =',summe)
```

b) **for**-Schleife

Struktogramm:

Eingabe k
summe = 1
summand = 1
for i = 2 to k
summand = summand + 2
summe = summe + summand
Ausgabe summe

Quellcode Python:

```
k = int(input('k = '))

summe = 1
summand = 1

for i in range(2,k+1):
    summand = summand + 2
    summe   = summe + summand

print('Summe der ersten',k,'ungeraden Zahlen =',summe)
```

Aufgabe 13

Für jede natürliche Zahl n, $n \geq 0$, und jede reelle Zahl a, $a \neq 0$, ist folgender Algorithmus als Struktogramm gegeben:

Eingabe a; n	
b:=a; u:=n; p:=1;	
while u>0	
u ungerade	
+	–
u:=u-1 p:=p*b	
u:=u div 2 b:=b*b	
Ausgabe p	

a) Schreibe und teste den Python-Quelltext zu vorstehendem Struktogramm.
b) Erstelle eine Trace-Tabelle für n = 7, a = 2.
c) Erstelle eine Trace-Tabelle für n = 18.

Prinzipien zur Formulierung eines Algorithmus

Imperativer Ansatz

Der Quellcode (formuliert in einer Programmiersprache, z. B. Pascal, Java oder Python) besteht aus einer Folge von ausführbaren Anweisungen, die in der vorgegebenen Reihenfolge nacheinander abgearbeitet werden.

Wesentliche Kontrollstruktur: **Iteration** (realisiert als for- oder while-Schleife)

Funktionaler Ansatz

Die Formulierung des Quellcodes orientiert sich der inneren, in der Regel mathematischen Struktur eines Algorithmus.

Wesentliche Kontrollstruktur: **Rekursion**

Definition:
*Eine Prozedur (Teilprogramm, Subroutine) oder eine Funktion heißt **rekursiv**, wenn deren Anweisungsblock mindestens einen Aufruf von sich selbst enthält.*

Bei beiden Ansätzen ist durch eine Abbruchbedingung sicherzustellen, daß der Algorithmus terminiert, also nach endlich vielen Schritten beendet wird und zu einem Ergebnis führt.

Beispiel 1: Die Fakultätsfunktion (engl.: factorial; Aufgabe 10)

Wir ordnen jeder natürlichen Zahl n, $n \geq 0$, die Zahl n! (lies: n-Fakultät) zu:

0! = 1
n! = 1 · 2 · . . . . . · n falls n > 0

Berechnung von n! gemäß imperativem Ansatz

```
# Fakultaet iterativ

# Eingabe
n = int(input('n = '))

# Verarbeitung

if n == 0:
   fact = 1

else:
   i = 1     # Initialisierung des Schleifenindex i
   fact = 1  # Anfangswert der Variablen fact
   while i <= n:
        fact = fact * i
        i = i + 1

# Ausgabe
print (n,'! = ',fact)
```

Berechnung von n! gemäß funktionalem Ansatz

Die Funktion $n \rightarrow fact(n)$ läßt sich rekursiv definieren:

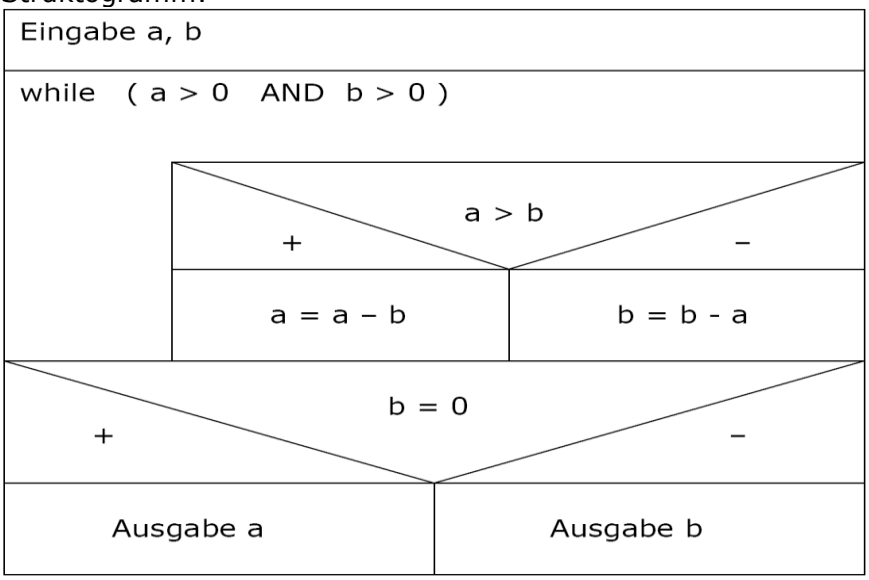
Rekursionsanfang: $fact(0) = 1$
Rekursionsvorschrift: $fact(n) = n \cdot fact(n-1)$, falls $n > 0$

```
# Fakultaet rekursiv

# Eingabe
n = int(input('n = '))

# Definition der Funktion factorial

def factorial(x):
      if x == 0:
          return 1
      else:
          return x * factorial(x - 1)


# Funktionsaufruf
fact = factorial(n)

# Ausgabe
print (n,'! = ',fact)
```

Beispiel 2: Der Algorithmus **ggT** (**g**rößter **g**emeinsamer **T**eiler)

Nach Eingabe zweier natürlicher Zahlen a und b bestimmt ggT die größte ganze Zahl, durch die sich a und b jeweils ohne Rest teilen lassen.

a) **Imperativer Ansatz**, formuliert als **iterativer Algorithmus**
(„Euklidischer Algorithmus")
Struktogramm:



b) **Funktionaler Ansatz**, formuliert als **rekursiv definierte Funktion**

Die Funktion $(a, b) \rightarrow ggT(a,b)$ läßt sich rekursiv definieren:

Rekursionsanfang: $ggT(a,a) = a$

Rekursionsvorschrift: $ggT(a,b) = ggT(a-b, b)$, falls $a > b$

$ggT(a,b) = ggT(a, b-a)$, falls $b > a$

Aufgabe:
Realisiere den Algorithmus ggT als iteratives und als rekursives Python-Programm; vergleiche die Laufzeiten.

Beispiel 3: Die **Hofstadter-Funktion**

Die Funktion **hof** ist rekursiv definiert, $n \in \{1, 2, 3, ... \}$:

Rekursionsanfang: $hof(1) = 1$
$hof(2) = 1$

Rekursionsvorschrift: $hof(n) = hof(n - hof(n - 1)) + hof(n - hof(n - 2)) , n>2$

Aufgabe:
Codiere den Algorithmus hofstadter
a) rekursiv,
b) iterativ
jeweils in Python; vergleiche insbesondere die Laufzeiten!

Hinweis zu b): Definiere in geeigneter Weise ein array (Feld; in Python: Liste), in dem bereits berechnete Funktionswerte gespeichert werden.

Aufgabe 14
Der Algorithmus **GAUSS**, der nach Eingabe einer natürlichen Zahl **n** die Summe der Zahlen **1, . . . , n** ermittelt, läßt sich sowohl imperativ als auch funktional programmieren (vgl. Aufgabe 9).
Ergreife diese beiden Möglichkeiten, indem jeweils ein Python-Quelltext erstellt wird (imperativ: Implementierung einer for- oder while-Schleife, mit Struktogramm; funktional: Implementierung einer rekursiv definierten Funktion)

Aufgabe 15
Eingabe: natürliche Zahl **k**, k > 0;
Ausgabe: Summe der ersten k ungeraden Zahlen (vgl. Aufgabe 12)

Rekursive Formulierung:

Rekursionsanfang: $summe(1) = 1$
Rekursionsvorschrift: $summe(k) = summe(k - 1) + 2 \cdot k - 1$ falls $k > 1$

Schreibe und teste ein Python-Programm mit rekursivem Funktionsaufruf!

Aufgabe 16
Nach Eingabe einer reellen Zahl **a** und einer natürlichen Zahl **n**, n ≥ 0, berechnet der Algorithmus **POTENZ** den Wert $\mathbf{a^n}$ und gibt diesen aus.

a) Formuliere den Potenzierungsalgorithmus iterativ (wahlweise while- oder for-Schleife) als Struktogramm und Python-Programm.
Implementiere eine Zählvariable z, um die Anzahl der Schleifendurchläufe zu bestimmen und auszugeben.

b) Vergleiche den Potenzierungs-Algorithmus aus Aufgabe 13 mit dem Algorithmus aus 16.a) hinsichtlich der Anzahl der benötigten Schleifendurchläufe.

c) Wegen $\mathbf{a^n = a \cdot a^{n-1}}$ und $\mathbf{a^0 = 1}$ läßt sich die Potenz als rekursive Funktion **f** definieren, die jedem **n** den Wert $\mathbf{a^n}$ zuordnet:

Rekursionsanfang: **f(0) = 1**
Rekursionsvorschrift: **f(n) = a · f(n-1)** falls n > 0

Beispiel:
a = 7, n = 4

$f(4) = 7^4 = 7 \cdot 7^3 = 7 \cdot (7 \cdot 7^2) = 7 \cdot (7 \cdot (7 \cdot 7^1)) = 7 \cdot (7 \cdot (7 \cdot (7 \cdot 7^0))) =$
$7 \cdot (7 \cdot (7 \cdot (7 \cdot 1))) = 7 \cdot (7 \cdot (7 \cdot 7)) = 7 \cdot (7 \cdot 49) = 7 \cdot 343 = 2401$

Formuliere ein Python-Programm zur Berechnung von $\mathbf{a^n}$ mit rekursivem Funktionsaufruf!

Nachtrag: Lösungen zu **Aufgabe 13**

a)

```
# Potenz a^n iterativ (Aufgabe 13)

a = float(input('a = '))
n = int(input('n = '))

b = a
u = n
p = 1

while u > 0:
    if u % 2 != 0:
        u = u - 1
        p = p * b
    u = u / 2
    b = b * b

print()
print (a,'^',n,' =',p)
```

b) Trace für a = 2, n = 7

	a	n	b	u	p	u ungerade	u > 0
vor dem 1. SD	2	7	2	7	1	+	+
vor dem 2. SD	2	7	4	3	2	+	+
vor dem 3. SD	2	7	16	1	8	+	+
nach dem 3. SD	2	7	256	0	128	–	–

c) Trace für n = 18

	a	n	b	u	p	u ungerade	u > 0
vor dem 1. SD	a	18	a	18	1	–	+
vor dem 2. SD	a	18	a^2	9	1	+	+
vor dem 3. SD	a	18	a^4	4	a^2	–	+
vor dem 4. SD	a	18	a^8	2	a^2	–	+
vor dem 5. SD	a	18	a^{16}	1	a^2	+	+
nach dem 5. SD	a	18	a^{32}	0	a^{18}	–	–

Vermutung:
Für eine reelle Zahl **a** und eine natürliche Zahl **n**, $n \geq 0$, berechnet der Algorithmus die Potenz $\mathbf{a^n}$ und gibt deren Wert aus.

Ausblick:
Die vorgenannte Vermutung läßt sich auch streng beweisen; hierzu zeigt man, daß die Beziehung

$$\mathbf{p \cdot b^u = a^n}$$

vor und nach jedem Schleifendurchlauf erfüllt, also invariant gegenüber Schleifendurchläufen ist (eine solche Beziehung heißt Schleifeninvariante).

Da während jedem Schleifendurchlauf entweder u halbiert wird, falls u gerade ist, oder u – 1 halbiert wird, falls u ungerade ist, nimmt u nach endlich vielen Schleifendurchläufen den Wert 0 an, und der Algorithmus terminiert (u>0 ist False, falls u=0 ist).

Sobald u den Wert 0 annimmt, folgt wegen $\mathbf{b^0 = 1}$ aus der Schleifeninvariante:

$$\mathbf{p = a^n}$$