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:

$0! = 1$
$n! = 1 \cdot 2 \cdot \ldots \cdot n$ 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:

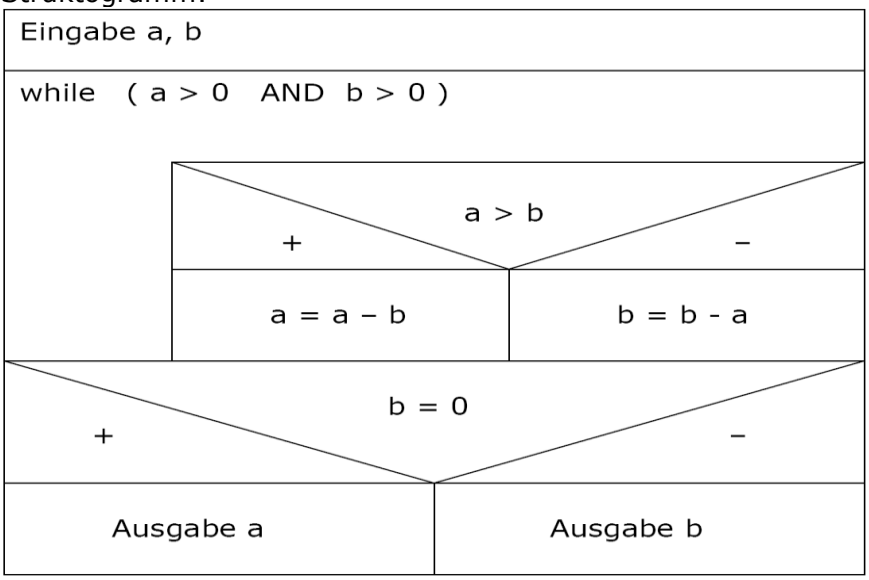
Rekursionsanfang: $fact(0) = 1$
Rekursionsvorschrift: $fact(n) = n \cdot fact(n-1)$, falls $n > 0$

```
# 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) → 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 ∈ {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·k – 1 falls k > 1

Schreibe und teste ein Python-Programm mit rekursivem Funktionsaufruf!