Informatik I: Einführung in die Programmierung

7. Entwurf von Schleifen, While-Schleifen, Hilfsfunktionen und Akkumulatoren

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Peter Thiemann

25. November 2020, 1. Dezember 2020

1 Entwurf von Schleifen

- Fallstudie: Rechnen mit Polynomen
- Skalarmultiplikation
- Auswertung
- Ableitung
- Integration
- Binäre Operationen
- Addition
- Multiplikation
- Extra: Lexikographische Ordnung

1 Entwurf von Schleifen

- Fallstudie: Rechnen mit Polynomen
- Skalarmultiplikation
- Auswertung
- Ableitung
- Integration
- Binäre Operationen
- Addition
- Multiplikation
- Extra: Lexikographische Ordnung

Polynome

Definition

Ein *Polynom vom Grad n* ist eine Folge von Zahlen $(a_0, a_1, \ldots, a_n)$, den *Koeffizienten*. Dabei ist $n \geq 0$ und $a_n \neq 0$.

Beispiele

- $()$
- (1)
- $(3,2,1)$

Anwendungen

Kryptographie, fehlerkorrigierende Codes.

Rechenoperationen auf Polynomen

- (Skalar) Multiplikation mit einer Zahl c

$$c \cdot (a_0, a_1, \dots, a_n) = (c \cdot a_0, c \cdot a_1, \dots, c \cdot a_n)$$

- Auswertung an der Stelle x_0

$$(a_0, a_1, \dots, a_n)[x_0] = \sum_{i=0}^{n} a_i \cdot x_0^i$$

- Ableitung

$$(a_0, a_1, \dots, a_n)' = (1 \cdot a_1, 2 \cdot a_2, \dots, n \cdot a_n)$$

- Integration

$$\int (a_0, a_1, \dots, a_n) = (0, a_0, a_1/2, a_2/3, \dots, a_n/(n+1))$$

1 Entwurf von Schleifen

- Fallstudie: Rechnen mit Polynomen
- Skalarmultiplikation
- Auswertung
- Ableitung
- Integration
- Binäre Operationen
- Addition
- Multiplikation
- Extra: Lexikographische Ordnung

Skalarmultiplikation

$$c \cdot (a_0, a_1, \ldots, a_n) = (c \cdot a_0, c \cdot a_1, \ldots, c \cdot a_n)$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `skalar_mult` nimmt als Eingabe

- `c : Complex`, den Faktor,
- `p : list[Complex]`, ein Polynom.

Der Grad des Polynoms ergibt sich aus der Länge der Sequenz.

Skalarmultiplikation

Schritt 2: Funktionsgerüst

```
def skalar_mult(
        c : Complex,
        p : list[Complex]
        ) -> list[Complex]:
    # fill in, initialization
    for a in p:
        # fill in action for each element
    return
```

Skalarmultiplikation

Schritt 3: Beispiele

```
assert(skalar_mult(42, []) == [])
assert(skalar_mult(42, [1,2,3]) == [42,84,126])
assert(skalar_mult(-0.1, [1,2,3]) == [-0.1,-0.2,-0.3])
```

Skalarmultiplikation

Schritt 4: Funktionsdefinition

```
def skalar_mult(
        c : Complex,
        p : list[Complex]
        ) -> list[Complex]:
    result = []
    for a in p:
        result = result + [c * a]
    return result
```

Muster: Akkumulator

Rumpf der Skalarmultiplikation

```
    result = []
    for a in p:
        result = result + [c * a]
    return result
```

Variable `result` ist Akkumulator

- In `result` wird das Ergebnis aufgesammelt (akkumuliert)
- `result` wird vor der Schleife initialisiert
  auf das Ergebnis für die leere Liste
- Jeder Schleifendurchlauf erweitert das Ergebnis in `result`
  indem das Ergebnis mit dem aktuellen Element a erweitert wird

1 Entwurf von Schleifen

- Fallstudie: Rechnen mit Polynomen
- Skalarmultiplikation
- Auswertung
- Ableitung
- Integration
- Binäre Operationen
- Addition
- Multiplikation
- Extra: Lexikographische Ordnung

Auswertung

$$(a_0, a_1, \ldots, a_n)[x_0] = \sum_{i=0}^{n} a_i \cdot x_0^i$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `poly_eval` nimmt als Eingabe

- `p : list[Complex]`, ein Polynom,
- `x : Complex`, das Argument.

Der Grad des Polynoms ergibt sich aus der Länge der Sequenz.

Schritt 2: Funktionsgerüst

```
def poly_eval(
        p : list[Complex],
        x : Complex
        ) -> Complex:
    # fill in
    for a in p:
        # fill in action for each element
    return
```

Schritt 3: Beispiele

```
assert(poly_eval([], 2) == 0)
assert(poly_eval([1,2,3], 2) == 17)
assert(poly_eval([1,2,3], -0.1) == 0.83)
```

Schritt 4: Funktionsdefinition

```
def poly_eval(
        p : list[Complex],
        x : Complex
        ) -> Complex:
    result = 0
    i = 0
    for a in p:
        result = result + a * x ** i
        i = i + 1
    return result
```

Schritt 4: Alternative Funktionsdefinition

```
def poly_eval(
        p : list[Complex],
        x : Complex
        ) -> Complex:
    result = 0
    for i, a in enumerate(p):   # «---
        result = result + a * x ** i
    return result
```

- `enumerate(seq)` liefert Paare aus (Laufindex, Element)
- Beispiel `list (enumerate([8, 8, 8])) == [(0, 8), (1, 8), (2, 8)]`

1 Entwurf von Schleifen

- Fallstudie: Rechnen mit Polynomen
- Skalarmultiplikation
- Auswertung
- Ableitung
- Integration
- Binäre Operationen
- Addition
- Multiplikation
- Extra: Lexikographische Ordnung

Ableitung

$$(a_0, a_1, \ldots, a_n)' = (1 \cdot a_1, 2 \cdot a_2, \ldots, n \cdot a_n)$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `derivative` nimmt als Eingabe

- `p : list[Complex]`, ein Polynom.

Der Grad des Polynoms ergibt sich aus der Länge der Sequenz.

Schritt 2: Funktionsgerüst

```
def derivative(
        p : list[complex]
        ) -> list[complex]:
    # initialization
    for a in p:
        pass # fill in action for each element
    return
```

Schritt 3: Beispiele

```
assert(derivative([]) == [])
assert(derivative([42]) == [])
assert(derivative([1,2,3]) == [2,6])
```

Schritt 4: Funktionsdefinition

```
def derivative(
        p : list[Complex]
        ) -> list[Complex]:
    result = []
    for i, a in enumerate(p):
        if i>0:
            result = result + [i * a]
    return result
```

1 Entwurf von Schleifen

- Fallstudie: Rechnen mit Polynomen
- Skalarmultiplikation
- Auswertung
- Ableitung
- Integration
- Binäre Operationen
- Addition
- Multiplikation
- Extra: Lexikographische Ordnung

Integration

$$\int(a_0, a_1, \dots, a_n) = (0, a_0, a_1/2, a_2/3, \dots, a_n/(n+1))$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `integral` nimmt als Eingabe

- `p : list[Complex]`, ein Polynom.

Der Grad des Polynoms ergibt sich aus der Länge der Sequenz.

Weitere Schritte

selbst

1 Entwurf von Schleifen

- Fallstudie: Rechnen mit Polynomen
- Skalarmultiplikation
- Auswertung
- Ableitung
- Integration
- Binäre Operationen
- Addition
- Multiplikation
- Extra: Lexikographische Ordnung

Operationen mit zwei Polynomen

- Addition (falls $n \leq m$)

$$(a_0, a_1, \ldots, a_n) + (b_0, b_1, \ldots, b_m)$$
$$= (a_0 + b_0, a_1 + b_1, \ldots, a_n + b_n, b_{n+1}, \ldots, b_m)$$

- Multiplikation von Polynomen

$$(a_0, a_1, \ldots, a_n) \cdot (b_0, b_1, \ldots, b_m)$$
$$= (a_0 \cdot b_0, a_0 \cdot b_1 + a_1 \cdot b_0, \ldots, \sum_{i=0}^{k} a_i \cdot b_{k-i}, \ldots, a_n \cdot b_m)$$

1 Entwurf von Schleifen

- Fallstudie: Rechnen mit Polynomen
- Skalarmultiplikation
- Auswertung
- Ableitung
- Integration
- Binäre Operationen
- Addition
- Multiplikation
- Extra: Lexikographische Ordnung

Addition

$$(a_0, a_1, \ldots, a_n) + (b_0, b_1, \ldots, b_m)$$
$$= (a_0 + b_0, a_1 + b_1, \ldots, a_n + b_n, b_{n+1}, \ldots, b_m)$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `poly_add` nimmt als Eingabe

- `p : list[Complex]`, ein Polynom.
- `q : list[Complex]`, ein Polynom.

Die Grade der Polynome ergeben sich aus der Länge der Sequenzen.

Achtung

Addition

Schritt 2: Funktionsgerüst

```
def poly_add(
        p : list[Complex],
        q : list[Complex]
        ) -> list[Complex]:
    # fill in
    for i in range(...):   # «---
        # fill in action for each element
    return
```

Frage

Was ist das Argument von `range`?

Addition

Schritt 3: Beispiele

```
assert(poly_add([], []) == [])
assert(poly_add([42], []) == [42])
assert(poly_add([], [11]) == [11])
assert(poly_add([1,2,3], [4,3,2,5]) == [5,5,5,5])
```

Antwort: Argument von `range`

```
maxlen = max (len (p), len (q))
```

Schritt 4: Funktionsdefinition, erster Versuch

```
def poly_add(
        p : list[Complex],
        q : list[Complex]
        ) -> list[Complex]:
    maxlen = max (len (p), len (q))
    result = []
    for i in range(maxlen):
        result = result + [p[i] + q[i]]
    return result
```

Addition

Problem

Eine Assertion schlägt fehl!

```
Traceback (most recent call last):
File ".../polynom.py", line 14, in <module>
  assert(poly_add([42], []) == [42])
File ".../polynom.py", line 10, in poly_add
  result = result + [p[i] + q[i]]
IndexError: list index out of range
```

Analyse

Zweite Assertion schlägt fehl für `i=0`!

Neuer Entwurfsschritt: Wunschdenken

Abstrahiere gewünschte Funktionalität in einer Hilfsfunktion.

Schritt 1: Bezeichner und Datentypen

Die Funktion `safe_index` nimmt als Eingabe

- `p : list[Complex]` eine Sequenz
- `i : int` einen Index (positiv)
- `d : Complex` einen Ersatzwert für ein Element von `p`

und liefert das Element `p[i]` (falls definiert) oder den Ersatzwert.

Schritt 2: Funktionsgerüst

```
def safe_index(
        p : list[Complex],
        i : int,   # assume >= 0
        d : Complex
        ) -> Complex:
    # fill in
    return 0
```

Schritt 3: Beispiele

```
assert safe_index([1,2,3], 0, 0) == 1
assert safe_index([1,2,3], 2, 0) == 3
assert safe_index([1,2,3], 4, 0) == 0
assert safe_index([1,2,3], 4, 42) == 42
assert safe_index([], 0, 42) == 42
```

Schritt 4: Funktionsdefinition

```
def safe_index(
        p : list[Complex],
        i : int,   # assume >= 0
        d : Complex
        ) -> Complex:
    return p[i] if i < len(p) else d
```

oder (alternative Implementierung des Funktionsrumpfes)

```
    if i < len(p):
        return p[i]
    else:
```

Neuer Ausdruck

Bedingter Ausdruck (Conditional Expression)

```
exp_true if cond else exp_false
```

- Werte zuerst `cond` aus
- Falls Ergebnis kein Nullwert, dann werte `exp_true` als Ergebnis aus
- Sonst werte `exp_false` als Ergebnis aus

Beispiele

- `17 if True else 4 == 17`
- `"abc"[i] if i<3 else "␣"`

Addition

Schritt 4: Funktionsdefinition mit Hilfsfunktion

```
def poly_add(
        p : list[Complex],
        q : list[Complex]
        ) -> list[Complex]:
    maxlen = max (len (p), len (q))
    result = []
    for i in range(maxlen):
        result = result + [
            safe_index(p,i,0) + safe_index (q,i,0)]
    return result
```

1 Entwurf von Schleifen

- Fallstudie: Rechnen mit Polynomen
- Skalarmultiplikation
- Auswertung
- Ableitung
- Integration
- Binäre Operationen
- Addition
- Multiplikation
- Extra: Lexikographische Ordnung

Multiplikation

$$(p_0, p_1, \ldots, p_n) \cdot (q_0, q_1, \ldots, q_m)$$

$$= (p_0 \cdot q_0, p_0 \cdot q_1 + p_1 \cdot q_0, \ldots, \sum_{i=0}^{k} p_i \cdot q_{k-i}, \ldots, p_n \cdot q_m)$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `poly_mult` nimmt als Eingabe

- `p : list[Complex]` ein Polynom
- `q : list[Complex]` ein Polynom

und liefert als Ergebnis das Produkt der Eingaben.

Multiplikation

Schritt 2: Funktionsgerüst

```
def poly_mult(
        p : list[Complex],
        q : list[Complex]
        ) -> list[Complex]:
    # fill in
    for k in range(...):
        # fill in
        # compute k-th output element
    return
```

Multiplikation

Schritt 3: Beispiele

```
assert poly_mult([], []) == []
assert poly_mult([42], []) == []
assert poly_mult([], [11]) == []
assert poly_mult([1,2,3], [1]) == [1,2,3]
assert poly_mult([1,2,3], [0,1]) == [0,1,2,3]
assert poly_mult([1,2,3], [1,1]) == [1,3,5,3]
```

Beobachtungen

- Range `maxlen = len (p) + len (q) - 1`

Multiplikation

Schritt 4: Funktionsdefinition

```
def poly_mult(
        p : list[Complex],
        q : list[Complex]
        ) -> list[Complex]:
    result = []
    for k in range(len(p) + len(q) - 1):
        rk = ... # k-th output element
        result = result + [rk]
    return result
```

Multiplikation

Das k-te Element

$$r_k = \sum_{i=0}^{k} p_i \cdot q_{k-i}$$

noch eine Schleife!

Berechnung

```
rk = 0
for i in range(k+1):
    rk = rk + (safe_index(p,i,0)
               * safe_index(q,k-i,0))
```

Schritt 4: Funktionsdefinition, final

```
def poly_mult(
        p : list[Complex],
        q : list[Complex]
        ) -> list[Complex]:
    result = []
    for k in range(len(p) + len(q) - 1):
        rk = 0
        for i in range(k+1):
            rk = rk + (safe_index(p,i,0)
                       * safe_index(q,k-i,0))
        result = result + [rk]
    return result
```

1 Entwurf von Schleifen

- Fallstudie: Rechnen mit Polynomen
- Skalarmultiplikation
- Auswertung
- Ableitung
- Integration
- Binäre Operationen
- Addition
- Multiplikation
- Extra: Lexikographische Ordnung

Erinnerung: Lexikographische Ordnung

Gegeben

Zwei Sequenzen der Längen $m, n \geq 0$:

$$\vec{a} = \text{"}a_1 a_2 \ldots a_m\text{"} \qquad\qquad \vec{b} = \text{"}b_1 b_2 \ldots b_n\text{"}$$

$\vec{a} \leq \vec{b}$ in der lexikographischen Ordnung, falls

Es gibt $0 \leq k \leq \min(m, n)$, so dass

- $a_1 = b_1, \ldots, a_k = b_k$ und

$$\vec{a} = \text{"}a_1 a_2 \ldots a_k a_{k+1} \ldots a_m\text{"} \qquad\qquad \vec{b} = \text{"}a_1 a_2 \ldots a_k b_{k+1} \ldots b_n\text{"}$$

  - $k = m$

$$\vec{a} = \text{"}a_1 a_2 \ldots a_m\text{"} \qquad\qquad \vec{b} = \text{"}a_1 a_2 \ldots a_m b_{m+1} \ldots b_n\text{"}$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `lex_ord` nimmt als Eingabe

- `a : list` eine Sequenz
- `b : list` eine Sequenz

und liefert als Ergebnis `True`, falls `a` $\leq$ `b`, sonst `False`.

Schritt 2: Funktionsgerüst

```
def lex_ord(
        a : list,
        b : list
        ) -> bool:
    # fill in
    for k in range(...):
```

Lexikographische Ordnung

Schritt 3: Beispiele

```
assert lex_ord([], []) == True
assert lex_ord([42], []) == False
assert lex_ord([], [11]) == True
assert lex_ord([1,2,3], [1]) == False
assert lex_ord([1], [1,2,3]) == True
assert lex_ord([1,2,3], [0,1]) == False
assert lex_ord([1,2,3], [1,3]) == True
assert lex_ord([1,2,3], [1,2,3]) == True
```

Beobachtungen

- Range `minlen = min (len (a), len (b))`

Lexikographische Ordnung

Schritt 4: Funktionsdefinition

```
def lex_ord(
        a : list,
        b : list
        ) -> bool:
    minlen = min (len (a), len (b))
    for k in range(minlen):
        if a[k] < b[k]:
            return True
        if a[k] > b[k]:
            return False
    # a is prefix of b or vice versa
    return len(a) <= len(b)
```

Exkursion: Typannotation für lexord (1)

Problem

- Vergleich von beliebigen Listen ist nicht möglich!
  Beispiel: `lex_ord ("abc", [1,2,3])` liefert Fehler!
- Wir müssen sicherstellen, dass
  1. die Elemente haben den gleichen Typ
  2. dieser Typ unterstützt Ordnungen

Eine Lösung

```
A = TypeVar ("A")   # from typing import TypeVar
```

definiert eine Typvariable. Damit kennzeichnet der Typ `list[A]` eine Liste, in der alle Elemente den gleichen Typ `A` haben, aber …

- wir wissen nicht, was A ist und

Exkursion: Typannotation für lexord (2)

Erweiterte Lösung

```
B = TypeVar ("B", int, float, str)
```

... wieder eine Typvariable, aber jetzt ist bekannt, dass sie für einen der aufgelisteten Typen `int`, `float` oder `str` steht.

```
def lex_ord(a : list[B], b : list[B]) -> bool:
```

bedeutet: `a` und `b` sind beides Listen, deren Elemente entweder `int` oder `float` oder `str` sind und daher vergleichbar!

Bewertung

ok, aber was ist mit `list[int]`, `list[list[int]]` usw? Alle diese Typen sind auch vergleichbar.

2 `while`-Schleifen

- Einlesen einer Liste
- Das Newton-Verfahren
- Das Collatz-Problem
- Abschließende Bemerkungen

Manchmal muss ein Schleifenrumpf wiederholt werden, ohne dass vorher klar ist, wie oft.

Beispiele

- Einlesen von mehreren Eingaben
- Newton-Verfahren zum Auffinden von Nullstellen
- Das Collatz-Problem

Die `while`-Schleife

- Syntax der `while`-Anweisung:

  `while` *Bedingung*:

  Anweisungen
- Die *Anweisungen* werden wiederholt, solange die *Bedingung* keinen Nullwert

2 `while`-Schleifen

- Einlesen einer Liste
- Das Newton-Verfahren
- Das Collatz-Problem
- Abschließende Bemerkungen

Beispiel: Einlesen einer Liste

Schritt 1: Bezeichner und Datentypen

Die Funktion `input_list` nimmt keine Parameter,
erwartet eine beliebig lange Folge von Eingaben, die mit einer leeren Zeile abgeschlossen ist, und liefert als Ergebnis die Liste dieser Eingaben als Strings.

Schritt 2: Funktionsgerüst

```
def input_list() -> list[str]:
    # fill in, initialization
    while CONDITION:
        # fill in
    return
```

Warum while?

- Die Anzahl der Eingaben ist nicht von vorne herein klar.
- Dafür ist eine while-Schleife erforderlich.
- Die while-Schleife führt ihren Rumpf wiederholt aus, solange nicht-leere Eingaben erfolgen.

Einlesen einer Liste

Beispiele

Eingabe:

```
>>> input_list()

[]
>>> input_list()
Bring
mal
das
WLAN-Kabel!

['Bring', 'mal', 'das', 'WLAN-Kabel!']
```

Schritt 4: Funktionsdefinition

```
def input_list() -> list[str]:
    result = []
    line = input()
    while line:
        result = result + [line]
        line = input()
    return result
```

2 `while`-Schleifen

- Einlesen einer Liste
- Das Newton-Verfahren
- Das Collatz-Problem
- Abschließende Bemerkungen

Das Newton-Verfahren

Suche Nullstellen von stetig differenzierbaren Funktionen

Verfahren

$f : \mathbb{R} \to \mathbb{R}$ sei stetig differenzierbar

1. Wähle $x_0 \in \mathbb{R}$, $n = 0$
2. Setze $$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$
3. Berechne nacheinander $x_1, x_2, \dots x_k$ bis $f(x_k)$ nah genug an 0.
4. Ergebnis ist x_k

Das Newton-Verfahren

Präzisierung

… für Polynomfunktionen

- Erfüllen die Voraussetzung
- Ableitung mit `derivative`

Was heißt hier “nah genug”?

- Eine überraschend schwierige Frage …
- Wir sagen: x ist nah genug an x', falls $\frac{|x-x'|}{|x|+|x'|} < \varepsilon$
- $\varepsilon > 0$ ist eine Konstante, die von der Repräsentation von `float`, dem Verfahren und der gewünschten Genauigkeit abhängt. Dazu kommen noch Sonderfälle.
- Wir wählen: $\varepsilon = 2^{-20} \approx 10^{-6}$
- Genug für eine Hilfsfunktion!

Hilfsfunktion

Die freundlichen Pythonistas waren schon aktiv. `pytest` ist ein Modul zur Erstellung von Tests.[1] Darin ist eine passende Hilfsfunktion definiert:

```
from pytest import approx
```

Diese Funktion erzeugt eine approximative Zahl, bei der Operator == ähnlich wie "nah genug" implementiert ist.
Es reicht, wenn ein Argument approximativ ist.

[1] Falls nicht vorhanden: pip3 install pytest

Schritt 1: Bezeichner und Datentypen

Die Funktion `newton` nimmt als Eingabe

- `f : list[float]` ein Polynom
- `x0 : float` einen Startwert

und verwendet das Newton-Verfahren zur Berechnung einer Zahl x, sodass $f(x)$ "nah genug" an 0 ist.

Schritt 2: Funktionsgerüst

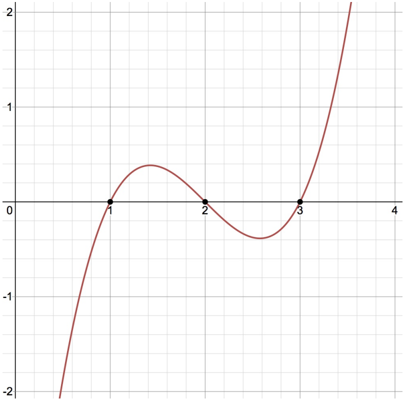
```
def newton(
        f  : list[float],
        x0 : float
        ) -> float:
    # fill in
    while CONDITION:
        pass # fill in
    return
```

Warum while?

- Das Newton-Verfahren verwendet eine Folge x_n, von der nicht von vorne herein klar ist, wieviele Elemente benötigt werden.
- Dafür ist eine while-Schleife erforderlich.
- Diese while-Schleife terminiert aufgrund der mathematischen / numerischen Eigenschaften des Newton-Verfahrens. Siehe Vorlesung Mathe I.

Newton-Verfahren

Beispielfunktion: $f(x) = x^3 - 6x^2 + 11x - 6$



Newton-Verfahren

Schritt 3: Beispiele

```
p = [-6, 11, -6, 1]
assert newton (p, 0)   == approx(1)
assert newton (p, 1.1) == approx(1)
assert newton (p, 1.7) == approx(2)
assert newton (p, 2.5) == approx(1)
assert newton (p, 2.7) == approx(3)
assert newton (p, 10)  == approx(3)
```

Schritt 4: Funktionsdefinition

```
def newton(
        f   : list[float],
        x0  : float
        ) -> float:
    deriv_f = derivative(f)
    xn = x0
    while poly_eval (f, xn) != approx(0):
        xn = xn - ( poly_eval (f, xn)
                  / poly_eval (deriv_f, xn))
    return xn
```

2 `while`-Schleifen

- Einlesen einer Liste
- Das Newton-Verfahren
- Das Collatz-Problem
- Abschließende Bemerkungen

Das Collatz-Problem

Verfahren (Collatz 1937)

Starte mit einer positiven ganzen Zahl n.

- Falls n gerade, fahre fort mit $\frac{n}{2}$.
- Sonst fahre fort mit $3n+1$.
- Wiederhole bis $n = 1$.

Offene Frage

Für welche Startwerte n wird nach endlich vielen Schritten $n = 1$ erreicht?

Beispiele (Folge der durchlaufenen Zahlen)

- `[3, 10, 5, 16, 8, 4, 2, 1]`
- `[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]`

Collatz-Problem

```
def collatz (n : int) -> list[int]:
    result = [n]
    while n > 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        result = result + [n]
    return result
```

Collatz-Problem

Warum while?

- Es ist nicht bekannt ob `collatz(n)` für jede Eingabe terminiert.
- Aber validiert für alle $n < 20 \cdot 2^{58} \approx 5.7646 \cdot 10^{18}$ (Oliveira e Silva).

2 `while`-Schleifen

- Einlesen einer Liste
- Das Newton-Verfahren
- Das Collatz-Problem
- Abschließende Bemerkungen

Termination einer Schleife

- Die Anzahl der Durchläufe einer `for`-Schleife ist stets durch den Schleifenkopf vorgegeben:
  - `for element in seq:`
    Anzahl der Elemente in der Sequenz `seq`
  - `for i in range(...):`
    Größe des Range
- Daher bricht die Ausführung einer `for`-Schleife stets ab (die Schleife terminiert).
- Bei einer `while`-Schleife ist die Anzahl der Durchläufe nicht vorgegeben.
- Daher ist stets eine Überlegung erforderlich, ob eine `while`-Schleife terminiert (Terminationsbedingung).
- Die Terminationsbedingung muss im Programm z.B. als Kommentar dokumentiert werden.

Beispiel Zweierlogarithmus (Terminationsbedingung)

Zweierlogarithmus

$$\log_2 a = b$$
$$2^b = a$$

- für $a > 0$

für ganze Zahlen

$$\texttt{l2 (n)} = m$$
$$m = \lfloor \log_2 n \rfloor$$

- für $n > 0$

Implementierung Zweierlogarithmus

```
def l2 (n : int) -> int:
    m = -1
    while n>0:
        m = m + 1
        n = n // 2
    return m
```

Terminationsbedingung

- Die `while`-Schleife terminiert, weil für alle `n>0` gilt, dass `n > n//2` und jede Folge `n1 > n2 > ...` abbricht.
- Anzahl der Schleifendurchläufe ist durch $\log_2 n$ beschränkt.

3 Zusammenfassung

Zusammenfassung

- Funktionen über Sequenzen verwenden for-in-Schleifen.
- Ergebnisse werden meist in einer Akkumulator Variable berechnet.
- Funktionen über mehreren Sequenzen verwenden for-range-Schleifen.
- Der verwendete Range hängt von der Problemstellung ab.
- Nicht-triviale Teilprobleme werden in Hilfsfunktionen ausgelagert.
- while-Schleifen werden verwendet, wenn die Anzahl der Schleifendurchläufe nicht von vorne herein bestimmt werden kann oder soll, typischerweise
  - zur Verarbeitung von Eingaben
  - zur Berechnung von Approximationen
- Jede while-Schleife muss eine dokumentierte Terminationsbedingung haben.