LU01.L12 - Trace Table für die Collatz-Folge

Teil 1: Iterative Fassung, Startwert n = 6

Schritt	n (vorher)	n % 2 == 0	angewandte Regel	n (nachher)	schritte
1	6	Ja	n // 2	3	1
2	3	Nein	3 * n + 1	10	2
3	10	Ja	n // 2	5	3
4	5	Nein	3 * n + 1	16	4
5	16	Ja	n // 2	8	5
6	8	Ja	n // 2	4	6
7	4	Ja	n // 2	2	7
8	2	Ja	n // 2	1	8
–	1	–	Schleifenbedingung `n != 1` nicht mehr erfüllt	1	8

Rückgabewert: 8

Die durchlaufene Folge lautet: `6 → 3 → 10 → 5 → 16 → 8 → 4 → 2 → 1`

Typischer Fehler

Häufig wird ein neunter Schritt eingetragen, in dem `n = 1` nochmals verarbeitet wird. Die Prüfung `n != 1` erfolgt aber **vor** dem Schleifenkörper. Sobald n den Wert 1 erreicht, endet die Schleife sofort – der Zähler bleibt bei 8.

Teil 2: Rekursive Fassung, Startwert n = 6

Aufruf	n	schritte	nächster Aufruf	Rückgabewert
1	6	0	collatz_rekursiv(3, 1)	8
2	3	1	collatz_rekursiv(10, 2)	8
3	10	2	collatz_rekursiv(5, 3)	8
4	5	3	collatz_rekursiv(16, 4)	8
5	16	4	collatz_rekursiv(8, 5)	8
6	8	5	collatz_rekursiv(4, 6)	8
7	4	6	collatz_rekursiv(2, 7)	8
8	2	7	collatz_rekursiv(1, 8)	8
9	1	8	– (Basisfall)	**8**

Der Basisfall in Aufruf 9 liefert `schritte = 8`. Jeder darüberliegende Aufruf gibt diesen Wert unverändert weiter – deshalb steht in der ganzen Spalte dieselbe 8.

Antworten auf die Auswertungsfragen

1. Welche Spalten werden überflüssig?

Die beiden Spalten *n (vorher)* und *n (nachher)* fallen zusammen zu einer einzigen Spalte `n`. Grund: In der rekursiven Fassung wird `n` nie überschrieben. Jeder Aufruf bekommt seinen eigenen `n`-Wert und behält ihn bis zum Ende.

Ebenso verschwindet die Notwendigkeit, `schritte` als laufend veränderten Zähler mitzuführen – der Wert wird als Parameter weitergereicht.

Der Kern: Veränderlicher Zustand zwingt dazu, in der Tabelle einen *Zeitverlauf* zu dokumentieren: Welchen Wert hatte `n` wann? Bei Parameterübergabe genügt eine *Momentaufnahme* pro Aufruf. Genau das meint die funktionale Programmierung, wenn sie sagt, Unveränderlichkeit mache Code leichter nachvollziehbar.

2. Warum die zweite Variante leichter nachzuvollziehen ist

In der ersten Fassung muss man beim Lesen einer Zeile immer wissen, **wie viele Schleifendurchgänge schon gelaufen sind** – der Name `n` allein sagt nichts über seinen Wert aus. In der zweiten Fassung ist jedes `n` innerhalb seines Aufrufs eine Konstante. Man kann jeden Aufruf isoliert betrachten und verstehen, ohne die Vorgeschichte zu kennen.

Das ist derselbe Gedanke wie die **referenzielle Transparenz** aus LU01c: Ein Ausdruck lässt sich durch seinen Wert ersetzen, weil er nicht vom bisherigen Verlauf abhängt.

3. Was passiert bei collatz(0)?

`n != 1` ist wahr, also läuft die Schleife. `n % 2 == 0` ist wahr, also wird `n = 0 // 2 = 0`. Der Wert ändert sich nie – die Schleifenbedingung wird nie falsch.

Fassung	Verhalten
iterativ	**Endlosschleife.** Das Programm hängt ohne Fehlermeldung und ohne Ausgabe. Man merkt den Fehler nur daran, dass nichts mehr passiert.
rekursiv	Nach rund 1000 Aufrufen bricht Python mit `RecursionError: maximum recursion depth exceeded` ab.

Die rekursive Fassung macht den Fehler also **früher und deutlicher sichtbar**: Eine Fehlermeldung mit Stack Trace ist wesentlich hilfreicher als ein Programm, das schweigend hängt.

Korrekt wäre in beiden Fällen eine Eingabeprüfung:

```
if n < 1:
    raise ValueError("n muss eine positive ganze Zahl sein")
```

4. Startwert n = 7

16 Schritte. Die Folge lautet:

`7 → 22 → 11 → 34 → 17 → 52 → 26 → 13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1`

Der Grund für die vielen Schritte: Ungerade Zahlen werden auf `3n + 1` vergrössert – die Folge steigt zunächst auf 52 an, bevor sie fällt. Bei 7 folgen mehrere ungerade Zahlen kurz hintereinander (7, 11, 17, 13, 5), sodass die Folge immer wieder nach oben springt.

Die Anzahl Schritte lässt sich der Startzahl nicht ansehen: 6 braucht 8 Schritte, 7 deren 16, 8 nur 3. Genau das macht die Collatz-Folge zu einem guten Beispiel dafür, warum man Algorithmen ausführen oder tracen muss, statt sie zu erraten.
