3.1 Rekursion

Versuch: Der Stapel Umschläge

Auf dem Pult liegt ein Umschlag. Darin: ein Zettel mit einer Zahl — und ein weiterer Umschlag. Darin wieder eine Zahl und ein Umschlag. Und so fort, bis irgendwann ein Umschlag kommt, der nur eine Zahl enthält und sonst nichts.

Aufgabe: Bestimmen Sie die Summe aller Zahlen. Eine Person darf immer nur **einen** Umschlag in der Hand halten.

Auswertung

Es gibt nur ein Vorgehen, das funktioniert: Sie lesen Ihre Zahl, geben den inneren Umschlag an den Nächsten weiter und warten. Irgendwann kommt von dort eine Summe zurück; Sie zählen Ihre Zahl dazu und geben das Ergebnis nach vorn.

Zwei Dinge fallen auf. Erstens hat niemand den ganzen Stapel überblickt — jeder kannte nur seinen Umschlag und die Regel. Zweitens ging es nur gut, weil es einen letzten Umschlag *ohne* Inhalt gab. Ohne ihn stünden Sie noch dort.

Definition: Rekursion

Ein Unterprogramm heißt **rekursiv**, wenn es sich selbst aufruft. Jede rekursive Lösung braucht zwei Teile:

- den **Basisfall** (Rekursionsanfang): der Fall, der ohne weiteren Aufruf beantwortet wird;
- den **Rekursionsschritt**: die Rückführung des Problems auf ein *kleineres* Problem derselben Art.

Gymnasium Ottweiler

Merke

Fehlt der Basisfall, oder wird das Problem nicht wirklich kleiner, läuft die Rekursion endlos — bis der Speicher für die offenen Aufrufe voll ist. Das Programm stürzt dann mit einer Meldung wie `RecursionError` oder `Stack overflow` ab.

Prüfen Sie bei jeder rekursiven Methode beides: **Gibt es einen Basisfall? Und führt jeder Schritt sicher dorthin?**

3.1.1 Die Fakultät

Die Fakultät kennen Sie als Produkt: $n! = 1 \cdot 2 \cdot \ldots \cdot n$. Rekursiv geschrieben sieht sie so aus:

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n-1)! & \text{falls } n > 0 \end{cases}$$

Die obere Zeile ist der Basisfall, die untere der Rekursionsschritt.

Python

```python
def fakultaet(n):
    if n == 0:
        return 1
    return n * fakultaet(n - 1)
```

Object Pascal

```
function Fakultaet(n: Integer): Int64;
begin
  if n = 0 then
    Result := 1
  else
    Result := n * Fakultaet(n - 1);
end;
```

So entfaltet sich `fakultaet(4)`:

```
fakultaet(4) = 4 * fakultaet(3)
                   fakultaet(3) = 3 * fakultaet(2)
                                      fakultaet(2) = 2 * fakultaet(1)
                                                         fakultaet(1) = 1 *
                                                         fakultaet(0)
                                                                          fakultaet
                                                                          (0) =
                                                                          1
                                                         fakultaet(1) = 1
                                      fakultaet(2) = 2
                   fakultaet(3) = 6
fakultaet(4) = 24
```

Gut zu sehen: Erst geht es nach unten bis zum Basisfall, dann rechnen sich die Werte wieder nach oben. Nichts wird berechnet, bevor der Basisfall erreicht ist.

3.1.2 Lineare und baumförmige Rekursion

Definition: Zwei Aufrufschemata

Bei der **linearen Rekursion** ruft sich die Methode pro Durchlauf **einmal** selbst auf. Die Aufrufe bilden eine Kette.

Bei der **baumförmigen Rekursion** ruft sie sich **mehrfach** selbst auf. Die Aufrufe bilden einen Baum.

Die Fakultät ist linear — eine Kette:

```
fak(4) → fak(3) → fak(2) → fak(1) → fak(0)
```

Die Fibonacci-Zahlen sind baumförmig. Sie sind definiert durch

$$\text{fib}(n) = \begin{cases} n & \text{falls } n < 2 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{falls } n \geq 2 \end{cases}$$

und ergeben die Folge 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

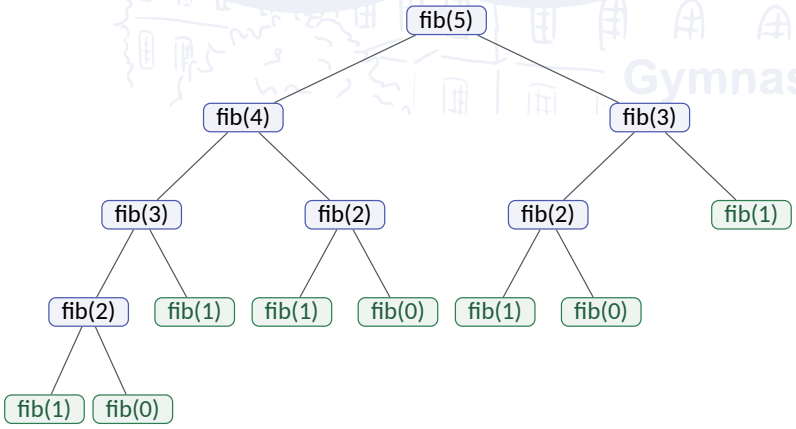
Python

```python
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

Object Pascal

```
function Fib(n: Integer): Int64;
begin
  if n < 2 then
    Result := n
  else
    Result := Fib(n - 1) + Fib(n - 2);
end;
```

Der Aufrufbaum für `fib(5)`:



15 Aufrufe für ein Ergebnis von 5. Grün = Basisfall. `fib(2)` wird dreimal gerechnet, `fib(1)` fünfmal.

Aufrufbaum von `fib(5)`.

> In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Rekursionsbaum**.

Beispiel: Was baumförmige Rekursion kostet

Zählt man die Aufrufe, wird das Problem sichtbar:

	`fib(5)`	`fib(10)`	`fib(20)`	`fib(30)`
Ergebnis	5	55	6 765	832 040
Aufrufe	15	177	21 891	2 692 537

`fib(2)` wird beim Aufruf von `fib(5)` dreimal berechnet, `fib(1)` sogar fünfmal — immer mit demselben Ergebnis. Der Baum rechnet dieselben Teilprobleme wieder und wieder.

Stolperstein

Baumförmige Rekursion ist elegant und kann sehr teuer sein. Lassen Sie `fib(50)` nicht auf einem Schulrechner laufen — die Aufrufzahl wächst exponentiell.

Arbeitsauftrag 1

1. Skizzieren Sie das Aufrufschema von `fakultaet(5)` und von `fib(4)`. Welches ist linear, welches baumförmig?
2. Bauen Sie in `fib` einen Zähler ein, der jeden Aufruf mitzählt, und prüfen Sie die Zahlen aus der Tabelle nach.
3. Die Türme von Hanoi: n Scheiben sollen von Stapel A nach C, Stapel B ist Hilfsstapel. Es darf nie eine größere auf einer kleineren liegen. Formulieren Sie Basisfall und Rekursionsschritt in Worten.

3.1.3 Rekursiv oder iterativ?

Jede rekursive Lösung lässt sich auch mit einer Schleife schreiben — und umgekehrt. Bei der Fakultät ist beides kurz:

```
def fakultaet_iterativ(n):
    ergebnis = 1
    for zahl in range(2, n + 1):
        ergebnis = ergebnis * zahl
    return ergebnis
```

Bei Fibonacci lohnt der Wechsel besonders, weil die iterative Fassung sich die beiden letzten Werte merkt, statt sie neu zu berechnen:

```
def fib_iterativ(n):
    vorletzter, letzter = 0, 1
    if n < 2:
        return n
    for _ in range(n - 1):
        vorletzter, letzter = letzter, vorletzter + letzter
    return letzter
```

```
function FibIterativ(n: Integer): Int64;
var
  vorletzter, letzter, hilf, i: Int64;
begin
  if n < 2 then
  begin
    Result := n;
    Exit;
  end;
  vorletzter := 0;
  letzter := 1;
  for i := 2 to n do
  begin
    hilf := vorletzter + letzter;
    vorletzter := letzter;
    letzter := hilf;
  end;
  Result := letzter;
end;
```

Diese Fassung braucht für `fib(30)` genau 29 Durchläufe statt 2 692 537 Aufrufe.

Merke

Vorteile der Rekursion: Sie bildet rekursiv definierte Probleme unmittelbar ab. Der Quelltext ist kurz und steht nah an der mathematischen Definition. Bei Bäumen ist sie praktisch die einzige vernünftige Wahl.

Nachteile: Jeder offene Aufruf belegt Speicher. Die Tiefe ist begrenzt. Und bei baumförmiger Rekursion werden Teilprobleme vielfach doppelt gerechnet.

Faustregel: Ist die Struktur rekursiv (Baum, Umschlag im Umschlag), nehmen Sie Rekursion. Ist sie linear (Feld, Liste, Zählschleife), nehmen Sie eine Schleife.

Arbeitsauftrag 2

4. Schreiben Sie `summe(a, b)` aus Kapitel 1 rekursiv.
5. Wandeln Sie diese iterative Methode in eine rekursive um. *(Die ganzzahlige Potenz steht nur im LK-Lehrplan, dort zusätzlich als `square-and-multiply`.)*

```
def potenz(basis, exponent):
    ergebnis = 1
    for _ in range(exponent):
        ergebnis = ergebnis * basis
    return ergebnis
```

6. Schreiben Sie eine rekursive Methode, die prüft, ob ein Wort ein Palindrom ist (`RENTNER`). Basisfall?