0) Grundmuster & Tricks

Sequenz (Liste) rekursiv verarbeiten

```
def solve_list(xs):
    if xs == []:                # Basisfall
        return <neutrales_Element>  # z.B. 0, 1, [] oder ""
    head, *tail = xs            # Zerlegung

    # Fallunterscheidung auf head
    if <Bedingung für head>:
        return <Kombiniere(head, solve_list(tail))>
    else:
        return <solve_list(tail)>
```

String normalisieren (nur alnum, lower) – rekursiv

```
def normalize(s):
    if s == "":
        return ""
    c = s[0]
    rest = s[1:]
    if c.isalnum():
        return c.lower() + normalize(rest)
    else:
        return normalize(rest)
```

Baum-Traversierung (Binärbaum)

```
class Node:
    def __init__(self, val, left=None, right=None):
        self.val, self.left, self.right = val, left, right

def traverse(root):
    if root is None:
        return <neutrales_Element>
    # z.B. inorder:
    return combine(
        traverse(root.left),
        root.val,
        traverse(root.right),
    )
```

Akkumulator (ohne Mutation)

```
def helper(xs, acc):
    if xs == []:
        return acc
    head, *tail = xs
    return helper(tail, <neuer_acc_aus(acc, head)>)
```

1) Zahlen & Formeln

ggT (Euklid)

```python
def gcd(a, b):
    if b == 0:
        return abs(a)
    return gcd(b, a % b)
```

Schnelle Potenz (Exponentiation by squaring)

```python
def fast_pow(x, n):
    if n == 0:
        return 1.0
    if n < 0:
        return 1.0 / fast_pow(x, -n)
    if n % 2 == 0:
        half = fast_pow(x, n // 2)
        return half * half
    else:
        return x * fast_pow(x, n - 1)
```

Digitensumme (Digital Root, simple)

```python
def digit_sum(n):
    n = abs(n)
    if n < 10:
        return n
    return digit_sum(n // 10 + (n % 10))
```

---

2) Josephus & Varianten

Josephus (1-basiertes Ergebnis)

```python
def josephus(n, k):
    if n == 1:
        return 1
    return ((josephus(n - 1, k) + k - 1) % n) + 1
```

Hot-Potato mit zyklischen Schrittweiten

```python
def hot_potato(n, steps):
    def rec(m):  # m = aktuelle Personenanzahl
        if m == 1:
            return 1
        k = steps[(n - m) % len(steps)]
        return ((rec(m - 1) + k - 1) % m) + 1
    return rec(n)
```

---

3) Listen-Grundrezepte

Länge (ohne len)

```
def length(xs):
    if xs == []:
        return 0
    _, *tail = xs
    return 1 + length(tail)
```

Zählen eines Zeichens

```
def count_char(s, ch):
    if s == "":
        return 0
    first, rest = s[0], s[1:]
    add = 1 if first == ch else 0
    return add + count_char(rest, ch)
```

Reverse (funktional)

```
def reverse_rec(xs):
    if xs == []:
        return []
    head, *tail = xs
    return reverse_rec(tail) + [head]
```

Flatten (nur Listen öffnen)

```
def flatten(xs):
    if xs == []:
        return []
    head, *tail = xs
    if isinstance(head, list):
        return flatten(head) + flatten(tail)
    else:
        return [head] + flatten(tail)
```

Map (rekursiv)

```
def map_rec(f, xs):
    if xs == []:
        return []
    head, *tail = xs
    return [f(head)] + map_rec(f, tail)
```

Filter (rekursiv)

```
def filter_rec(p, xs):
    if xs == []:
        return []
    head, *tail = xs
    if p(head):
        return [head] + filter_rec(p, tail)
    else:
        return filter_rec(p, tail)
```

4) Suchen & Sortieren

Binäre Suche (Index oder None) – Liste sortiert

```
def binary_search(xs, target, offset=0):
    if xs == []:
        return None
    mid = len(xs) // 2
    x = xs[mid]
    if x == target:
        return offset + mid
    if target < x:
        return binary_search(xs[:mid], target, offset)
    else:
        return binary_search(xs[mid+1:], target, offset + mid + 1)
```

Mergesort (funktional)

```
def merge(a, b):
    if a == []:
        return b
    if b == []:
        return a
    ah, *at = a
    bh, *bt = b
    if ah <= bh:
        return [ah] + merge(at, b)
    else:
        return [bh] + merge(a, bt)

def mergesort(xs):
    if length(xs) <= 1:
        return xs
    mid = length(xs) // 2
    left  = mergesort(xs[:mid])
    right = mergesort(xs[mid:])
    return merge(left, right)
```

5) Strings & Muster

Palindrom (mit Normalisierung)

```
def is_palindrome(s):
    ns = normalize(s)
    if len(ns) <= 1:
        return True
    return ns[0] == ns[-1] and is_palindrome(ns[1:-1])
```

Run-Length Encoding (nur Kodierung)

```
def rle_encode(s):
    if s == "":
        return []
    first = s[0]
    # zähle gleiche vorne
    def take_same(s, ch):
        if s == "" or s[0] != ch:
            return ("", 0)
        rest, n = take_same(s[1:], ch)
        return (rest, n + 1)
    rest, n = take_same(s, first)
    # Achtung: take_same hat "alles" aufgefressen – rekursiv auf rest
    # (Alternative: schreibe take_same so, dass es rest der Kette zurückgibt)
```

Pragmatische Variante ohne Hilfsrekursion:

```
def rle_encode(s):
    if s == "":
        return []
    ch = s[0]
    # ziehe gleichen Prefix
    def consume(s, ch):
        if s == "" or s[0] != ch:
            return (0, s)
        n, tail = consume(s[1:], ch)
        return (n + 1, tail)
    n, tail = consume(s, ch)
    return [(ch, n)] + rle_encode(tail)
```

Zeichen entfernen (alle Vorkommen)

```
def remove_char(s, ch):
    if s == "":
        return ""
    first, rest = s[0], s[1:]
    if first == ch:
        return remove_char(rest, ch)
    else:
        return first + remove_char(rest, ch)
```

6) Kombinatorik

Permutationen

```
def permutations(xs):
    if xs == []:
        return [[]]
    head, *tail = xs
    perms_tail = permutations(tail)
    def insert_all_positions(x, ys):
        if ys == []:
            return [[x]]
        first, *rest = ys
        # x vorne + x in alle Positionen von rest
        return [[x] + ys] + [[first] + p for p in insert_all_positions(x, rest)]
    # kombiniere
    result = []
    for p in perms_tail:             # (falls Schleifen tabu, ersetze durch rekursive map/concat)
        result += insert_all_positions(head, p)
    return result
```

Hinweis: Wenn Schleifen strikt tabu sind, schreibe `concat_map_rec(fn, list)` rekursiv und nutze es statt der `for`-Schleife.

Potenzmenge (Subsets)

```
def subsets(xs):
    if xs == []:
        return [[]]
    head, *tail = xs
    subs = subsets(tail)
    # mit head
    with_head = [[head] + s for s in subs]
    return subs + with_head
```

K-Kombinationen

```
def combinations(xs, k):
    if k == 0:
        return [[]]
    if xs == []:
        return []
    head, *tail = xs
    with_head = [[head] + c for c in combinations(tail, k-1)]
    without_head = combinations(tail, k)
    return with_head + without_head
```

Gültige Klammerfolgen

```
def gen_parentheses(n):
    def rec(open_left, close_left, prefix):
        if open_left == 0 and close_left == 0:
            return [prefix]
        res1 = []
        if open_left > 0:
            res1 = rec(open_left - 1, close_left, prefix + "(")
        res2 = []
        if close_left > open_left:
            res2 = rec(open_left, close_left - 1, prefix + ")")
        return res1 + res2
    return rec(n, n, "")
```

Coin Change – Anzahl der Möglichkeiten

```
def count_change(amount, coins):
    if amount == 0:
        return 1
    if amount < 0 or coins == []:
        return 0
    head, *tail = coins
    # Fall 1: head verwenden
    use_it = count_change(amount - head, coins)
    # Fall 2: head nicht verwenden
    skip_it = count_change(amount, tail)
    return use_it + skip_it
```

7) Verschachtelte Strukturen

Summe in Listen/Dicts (ohne bool)

```
def sum_nested(data):
    # bool ausschließen: isinstance(True, int) == True
    if isinstance(data, bool):
        return 0.0
    if isinstance(data, (int, float)):
        return float(data)
    if isinstance(data, list):
        if data == []:
            return 0.0
        head, *tail = data
        return sum_nested(head) + sum_nested(tail)
    if isinstance(data, dict):
        # nur Values
        return sum_nested(list(data.values()))
    return 0.0
```

Ausdrucksbaum evaluieren

```
def eval_expr(expr):
    if isinstance(expr, (int, float)):
        return float(expr)
    op, a, b = expr
    x, y = eval_expr(a), eval_expr(b)
    if op == "add": return x + y
    if op == "sub": return x - y
    if op == "mul": return x * y
    if op == "div": return x / y
    # optional: Fehlerfall
```

Pascal-Zeile (0-basiert)

```
def pascal_row(n):
    if n == 0:
        return [1]
    prev = pascal_row(n - 1)
    # Mittelwerte paarweise addieren
    def mids(xs):
        if xs == [] or xs[1:] == []:
            return []
        a, b = xs[0], xs[1]
        return [a + b] + mids(xs[1:])
    return [1] + mids(prev) + [1]
```

8) Testschnipsel (schnell checken)

```
# Josephus
print(josephus(5,2))     # 3
print(josephus(7,3))     # 4
print(josephus(10,1))    # 10

# Flatten
print(flatten([1,[2,[3],4],5]))  # [1,2,3,4,5]

# Palindrome
print(is_palindrome("No lemon, no melon!"))  # True

# gcd / fast_pow
print(gcd(54,24))          # 6
print(fast_pow(2,10))      # 1024

# Mergesort
print(mergesort([3,1,4,1])) # [1,1,3,4]

# Subsets / Combos
print(subsets([1,2]))             # [[],[2],[1],[1,2]] (Reihenfolge egal)
print(combinations([1,2,3], 2))   # [[1,2],[1,3],[2,3]]

# Coin change
print(count_change(4, [1,2,3]))   # 4

# Pascal
print(pascal_row(4))              # [1,4,6,4,1]

# Sum nested
print(sum_nested({"a":1,"b":[2,{"c":3.5},True]}))  # 6.5
```

9) Häufige Fehler & schnelle Checks

- **Basisfall vergessen** → Endlosschleife/Maximum Recursion Depth.
- **Tail falsch** → `flatten(0)` o.ä. (Tail ist **immer eine Liste**).
- **Mutation** (append/extend/+=) vermeiden → stattdessen **neue** Listen bauen.
- **Bool als int** zählen → vorher `isinstance(x, bool)` ausschließen.
- **1-basierte Ergebnisse** (z. B. Josephus) beachten.