Rekurencja i iteracja

Rekurencja (rekursja) – luźna definicja: definiowanie obiektu poprzez sam obiekt.

Przykład 1:

$$a_n = \begin{cases} 1, & \text{gdy } n = 0, \\ 2a_{n-1} + 1, & \text{w.p.p.} \end{cases}$$

Tę definicję ciągu łatwo przełożyć na kod w Pythonie:

```
def a(n):
    if n == 0:
        return 1
    return 2 * a(n - 1) + 1
```

Wyznaczenie a_n odpowiada rekurencyjnemu rozpisaniu wartości $(a_i)_{i \leqslant n}$, np.:

$$a_4 = 2a_3 + 1 = 2(2a_2 + 1) + 1 = 2(2(2a_1 + 1) + 1) + 1 =$$
$$= 2(2(2(2a_0 + 1) + 1) + 1) + 1 = 2(2(2(2 + 1) + 1) + 1) + 1 = 31.$$

Rekurencja i iteracja

Alternatywny sposób znalezienia a_n: wyznaczyć kolejne elementy ciągu $a_0, a_1, \ldots, a_n$ i zwrócić a_n.

```
def a_iter(n):
    a = 1
    for _ in range(n):
        a = 2 * a + 1
    return a
```

Co więcej, wartość a_n można wskazać zamkniętą formułą:
Fakt. $\forall n \in \mathbb{N}\ a_n = 2^{n+1} - 1$.
(Dowód: indukcja).

Stąd prosty kod:

```
def a_form(n):
    return 2 ** (n + 1) - 1
```

Nie każda rekurencyjna definicja jest poprawna.

Przykład 2: $a_n = 2a_{n-1} + 1$.
Brak warunku początkowego - problem nieskończonego regresu:

$$a_2 = 2a_1 + 1 = 2(2a_0 + 1) + 1 = 2(2(2a_{-1} + 1) + 1) + 1 = \dots$$

(ale: definicja rekurencja wymaga *jakiegoś* regresu)

Poprawność schematu rekurencyjnego można pokazać indukcyjnie. W Przykładzie 1:

- Dla $n = 0$, a_n jest poprawnie zdefiniowane ($a_0 = 1$).
- Jeśli a_n jest poprawnie zdefiniowane, to a_{n+1} też ($a_{n+1} = 2a_n + 1$).
- Przez indukcję: definicja jest poprawna dla wszystkich $n \in \mathbb{N}$.

Przykład 3: symbol Newtona.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \text{ dla } n, k \in \mathbb{N}, 0 \leqslant k \leqslant n.$$

Symbol Newtona spełnia (dla wszystkich $k \leqslant n$):

$$\binom{n}{k} = \begin{cases} 1, & \text{gdy } k = 0 \vee k = n, \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & \text{w.p.p.} \end{cases}$$

Powyższą własność można przyjąć za definicję symbolu! (Dowód: indukcja).

```
def binom(n, k):
    if k in (0, n):
        return 1
    return binom(n - 1, k - 1) + binom(n - 1, k)
```

A definicja iteracyjna?

Największy wspólny dzielnik

Niech $d, n \in \mathbb{Z}$. Mówimy, że d jest dzielnikiem n gdy istnieje $k \in \mathbb{Z}$ takie, że $dk = n$. Piszemy wtedy $d|n$.
Uwaga: w tej definicji każda liczba całkowita dzieli 0.

Problem: dla zadanej pary liczb $a, b \in \mathbb{Z}$, znaleźć największy wspólny dzielnik (*greatest common divisor*) a i b. Tutaj „największy" oznacza „największy według zwykłego porządku na $\mathbb{Z}$".

Dla $a = b = 0$ największy wspólny dzielnik nie istnieje i w ramach konwencji przyjmiemy $gcd(0, 0) = 0$.

Z drugiej strony (przypomnienie z WdM): 0 jest największą liczbą naturalną w relacji częściowego porządku zadanej przez podzielność!

Algorytm Euklidesa

Dla $b \neq 0$, oznaczmy resztę z dzielenia a przez b jako $a \bmod b$.
$\left[\ (a \bmod b) \in \{0, \ldots, |b| - 1\}\ \right]$

Fakt: Dla $b \neq 0$, $gcd(a, b) = gcd(b, a \bmod b)$.
Z drugiej strony: $gcd(a, 0) = |a|$. Rozważmy zatem schemat:

$$gcd(a, b) = \begin{cases} |a|, & \text{gdy } b = 0. \\ gcd(b, a \bmod b), & \text{w.p.p.} \end{cases}$$

$$\begin{aligned} gcd(372, 783) &= gcd(783, 372 \bmod 783) && = gcd(783, 372) \\ &= gcd(372, 783 \bmod 372) && = gcd(372, 39) \\ &= gcd(39, 372 \bmod 39) && = gcd(39, 21) \\ &= gcd(21, 39 \bmod 21) && = gcd(21, 18) \\ &= gcd(18, 21 \bmod 18) && = gcd(18, 3) \\ &= gcd(3, 18 \bmod 3) && = gcd(3, 0) = 3. \end{aligned}$$

Algorytm Euklidesa

Ogólnie: konstruujemy ciąg par a_i, b_i w następujący sposób:

- $a_0 = a, b_0 = b$.
- Dla kolejnych i: jeśli $b_i = 0$, przerywamy konstrukcję, przeciwnym wypadku, niech $a_{i+1} = b_i$, $b_{i+1} = a_i \bmod b_i$.

Dla $b \neq 0$, mamy: $a \bmod b < |b|$, zatem konstrukcja zawsze skończy się po skończenie wielu krokach i dla pewnego k mamy $b_k = 0$. Wtedy:

$$gcd(a, b) = gcd(a_0, b_0) = gcd(a_1, b_1) = \ldots = gcd(a_k, b_k) = |a_k|.$$

Stąd:

- Schemat z poprzedniego slajdu poprawnie zadaje funkcję gcd.
- Konstrukcja ciągu par a_i, b_i wskazuje nam pomysł na iteracyjną implementację.

Algorytm Euklidesa

Algorytm Euklidesa – rekurencyjnie:

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

Algorytm Euklidesa – iteracyjnie:

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return abs(a)
```