Tytuł ćwiczenia: Jednostka 7 - Paradygmat funkcyjny (cz. 2)
Autor: Joanna Dagil
Grupa: TCH-1
Data: 25.11.2025

1 Cel ćwiczenia

Celem ćwiczenia jest zapoznanie się z rekurencją i leniwą ewaluacją (przy użyciu generatorów) w paradygmacie funkcyjnym.

2 Przebieg ćwiczenia

W tym ćwiczeniu posłużę się językiem Python w edytorze Visual Studio Code. W celu wykonania zadań zapoznaję się z tutorialami, a następnie przechodzę do stworzenia pliku:

123.py

```python
import time

def suma_lista(lista, i=0):
    if i == len(lista):
        return 0
    return lista[i] + suma_lista(lista, i+1)

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

def fib_tail(n, last=1, lastlast=0):
    if n == 1: return last
    return fib_tail(n-1, last+lastlast, last)

def liczby_pierwsze():
    n = 2
    while True:
        ok = True
        for i in range(2,n):
            if n % i == 0:
                ok = False
                break
        if ok: yield n
        n += 1

if __name__ == "__main__":
    print("-------------- ZADANIE 1 --------------")
    lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    print("lista:   ", lista)
    print("suma_lista(lista):  ", suma_lista(lista))
    print("-------------- ZADANIE 2 --------------")
    time3 = time.time()
    res = fib_tail(35)
    time4 = time.time()
    print("fib_tail(35):", res, ", time: ", (time4-time3) * 1000000)
    time1 = time.time()
    res = fib(35)
    time2 = time.time()
    print("fib(35):     ", res, ", time: ", (time2-time1) * 1000000)
    print("-------------- ZADANIE 3 --------------")
    gen = liczby_pierwsze()
    for _ in range(20):
        print(next(gen), end=" ")
    print()
```

4.py

```python
def gen_fib():
    yield 0
    yield 1
    last = 1
    lastlast = 0
    while True:
        lastlast, last = last, last + lastlast
        yield last

def even(seq):
    for x in seq:
        if x % 2 == 0: yield x

def sqrt(seq):
    for x in seq: yield x*x

if __name__ == "__main__":
    print("-------------- ZADANIE 4 --------------")
    gen = gen_fib()
    res = sqrt(even(gen))
    for _ in range(10):
        print(next(res), end=" ")
    print()
```

Programy uruchamiam poleceniem

```
python {nazwa}.py
```

3 Wyniki działania

Dla przykładowych danych

```
-------------- ZADANIE 1 --------------
lista:    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
suma_lista(lista):   55
-------------- ZADANIE 2 --------------
fib_tail(35): 9227465 , time:  8.821487426757812
fib(35):      9227465 , time:  1142055.7498931885
-------------- ZADANIE 3 --------------
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
-------------- ZADANIE 4 --------------
0 4 64 1156 20736 372100 6677056 119814916 2149991424 38580030724
```

4 Wnioski

4.1 Porównanie rekurencji i iteracji na problemie sumowania listy elementów z zadania 1.

W podejściu iteracyjnym, aby obliczyć sumę elementów listy, używamy pętli (np. `for` lub `while`), która kolejno przechodzi po wszystkich elementach i dodaje je do akumulatora. Taki kod jest bezpośrednio związany z tym, jak komputer wykonuje obliczenia krok po kroku.

W podejściu rekurencyjnym, zastosowanym w funkcji:

```python
def suma_lista(lista, i=0):
    if i == len(lista):
        return 0
    return lista[i] + suma_lista(lista, i+1)
```

problem jest opisany bardziej deklaratywnie: “suma listy to pierwszy element plus suma reszty listy”. Funkcja ma wyraźnie wydzielony przypadek bazowy (`i == len(lista)`) i krok rekurencyjny (wywołanie `suma_lista` dla następnego indeksu).

Dla krótkich list oba podejścia zachowują się bardzo podobnie pod względem czasu wykonania. Różnica pojawia się przy dużych strukturach danych: pętle są zwykle bardziej wydajne i nie grozi im przepełnienie stosu, natomiast wersja rekurencyjna zużywa dodatkową pamięć na kolejne ramki stosu i przy bardzo długich listach może zakończyć się błędem.

W podejściu iteracyjnym możliwa jest również automatyczna optymalizacja pętli przez kompilator.

Natomiast z problemem przepełniania się ramek stosu w podejściu funkcyjnym można sobie poradzić stosując rekurencję ogonową.

Z punktu widzenia paradygmatu funkcyjnego rekurencja jest jednak bardziej naturalna i lepiej oddaje matematyczną definicję problemu.

4.2 Diagram drzewa wywołań dla rekurencji

Rozpatrując funkcję `fib(n)`:

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

drzewo wywołań dla przykładowego `fib(5)` ma strukturę rozgałęziającą się: w każdym kroku powstają dwa kolejne wywołania (`fib(n-1)` i `fib(n-2)`). W efekcie te same argumenty (np. `fib(2)`, `fib(3)`) są obliczane wielokrotnie, a liczba wywołań rośnie w przybliżeniu wykładniczo wraz ze wzrostem `n`.

Na diagramie drzewa wywołań widać:

- wielokrotne pojawianie się tych samych podproblemów,
- szybkie "puchnięcie" drzewa przy większych `n`,
- że przypadki bazowe (`n == 0` lub `n == 1`) stanowią liście drzewa.

W przeciwieństwie do tego, funkcja `fib_tail(n, last, lastlast)` ma tylko jedno wywołanie rekurencyjne w każdym kroku — jej drzewo wywołań jest de facto liniowym łańcuchem. Zmienia to złożoność obliczeniową z wykładniczej na liniową (liczba wywołań jest proporcjonalna do `n`), chociaż w Pythonie nadal każde wywołanie zajmuje jedno miejsce na stosie, ponieważ nie ma optymalizacji rekurencji ogonowej.