



SPOŁECZNA AKADEMIA NAUK

[www.san.edu.pl](http://www.san.edu.pl)

Kierunek studiów: **Informatyka**

Nazwisko i Imię: Zbigniew Miara

Numer albumu: 113962

Semestr: IV

Grupa: 2

Studia Inż.

**Wprowadzenie do Metod Numerycznych - inż. sem. 4**

**Sprawozdanie z Laboratorium 1-7**

Prowadzący zajęcia: *dr inż. Konrad Grzanek*

**Łódź 2024**

## Wstęp:

Zadanie wykonane w języku programowania F#. Zdecydowałem się na ten język ze względu na bliskość konstrukcyjną samego języka z językiem C# który jest dla mnie najlepiej na ten czas znany. W znaczącym stopniu ułatwiło mi to rozwiązywanie poniższych zadań. Kolejnym atutem platformy F# jest dobrze dostępna i opisana dokumentacja języka na stronie Microsoftu. Poniżej zamieszczam pięć rozwiązanych zadań wraz z opisem jak zostały wykonane w myśl metodyki programowania funkcyjnego.

## Zadanie 1:

The sum of the squares of the first ten natural numbers is,

$$1^2 + 2^2 + \dots + 10^2 = 385.$$

The square of the sum of the first ten natural numbers is,

$$(1 + 2 + \dots + 10)^2 = 55^2 = 3025.$$

Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is  $3025 - 385 = 2640$ .

Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum.

## Program:

```
let square x =
    printfn "Square of %d: %d" x (x * x)
    x * x

let sum_of_square max =
    let result =
        Seq.unfold (fun x -> if x > max then None else Some(x*x, x+1)) 1
        |> Seq.sum
    printfn "Sum of squares up to %d: %d" max result
    result

let square_of_sum max =
    let sum = {1 .. max} |> Seq.sum
    let result = square sum
    printfn "Square of sum up to %d: %d" max result
    result

let problem_6 max =
    let squareSum = square_of_sum(max)
    let sumSquare = sum_of_square(max)
    let result = squareSum - sumSquare
    printfn "Result of problem 6 with max %d: %d" max result
    result

problem_6 100
```

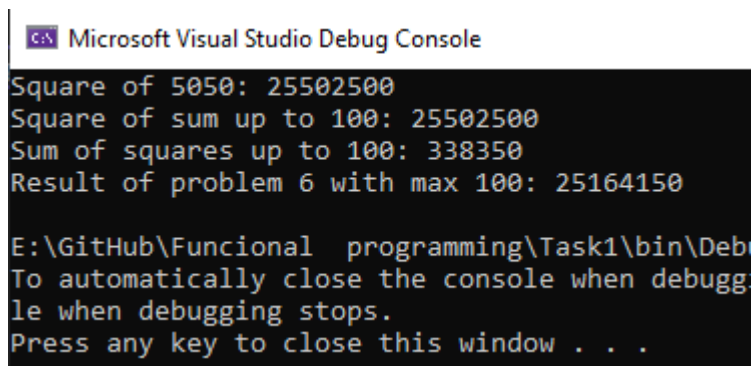
## Opis:

Program, który napisałem rozwiązuje problem nr. 6 z listy Projektu Euler. Zadanie polega na obliczeniu różnicy między sumą kwadratów pierwszych  $n$  liczb naturalnych a kwadratem tej sumy.

Program składa się z następujących funkcji:

- „square x”- Funkcja z parametrem  $x$ . Oblicza kwadrat liczby  $x$  poprzez pomnożenie przez samą siebie. Zwraca obliczoną wartość kwadratu i wyświetla informacyjnie wynik.
- „sum\_of\_square max”- Ta funkcja oblicza sumę kwadratów liczb od 1 do max. Zastosowałem tu wbudowaną funkcję `Seq.unfold` do utworzenia tablicy kwadratów liczb od 1 do max a następnie sumuje tę tablicę. **Seq.unfold** to funkcja w języku F#, która służy do generowania sekwencji wartości z rekurencyjnej strukturach danych to znaczy. „Seq.unfold (fun x -> if x > max then None else Some(x\*x, x+1)) 1”- w parametrze funkcji posiadamy funkcję anonimową która przyjmuje warunek „if” jeśli  $x$  jest większe zwraca „none” jeżeli nie to za pomocą „Seq.Unfold” zwraca krotkę mnożąc  $x$  przez siebie a następnie do zadanego  $x$  wykonuje inkrementację o 1. 1 na końcu oznacza wartość od której zaczynają się obliczenia. `|> Seq.sum` wykonuje sumowanie całej tablicy
- „square\_of\_sum max”- Ta funkcja oblicza sumę liczb od 1 do max i używa funkcji `square` zdefiniowanej wcześniej do obliczenia kwadratu tej sumy. W tej metodzie także została zastosowana sekwencja z operatorem „...” - `let sum = {1 .. max} |> Seq.sum`
- „problem\_6 max”- Ta funkcja służy do podania ostatecznego wyniku. Najpierw wywołuje funkcję `square_of_sum` potem funkcję `sum_of_square`. Po czym oblicza różnicę między tymi dwiema wartościami zwraca jako wynik końcowy.
- „problem\_6 100”-Wywołanie funkcji `problem_6` dla zadanej wartości

## Wynik:



```
Microsoft Visual Studio Debug Console
Square of 5050: 25502500
Square of sum up to 100: 25502500
Sum of squares up to 100: 338350
Result of problem 6 with max 100: 25164150

E:\GitHub\Funcional programming\Task1\bin\Debug
To automatically close the console when debugging
le when debugging stops.
Press any key to close this window . . .
```

## Zadanie 2:

### a. zaimplementować wzór na pierwiastek sześcienny :

Do obliczenia pierwiastka zastosowałem Metodę Newtona-Raphsona. Metoda ta jest iteracyjną metodą numeryczną służącą do znajdowania przybliżeń pierwiastków funkcji. W tym przypadku nasza funkcja ma wartość  $f(x)=x^3$

### Program:

```
//1
let cubeRoot x =
  let rec loop y =
    let nextY = (2.0 * y + x / (y * y)) / 3.0
    if abs (nextY - y) < 1e-10 then
      nextY
    else
      loop nextY
  loop x
printfn "Cube root of 27.0: %f" (cubeRoot 27.0)
```

- Funkcja rekurencyjna rec (Funkcja rekurencyjna to taka, która wywołuje samą siebie.) loop przyjmuje jeden argument y, który reprezentuje bieżące przybliżenie pierwiastka sześciennego.
- Wewnątrz funkcji definiowana jest zmienna nextY, która przechowuje nowe przybliżenie pierwiastka sześciennego.
- Wartość nextY jest obliczana za pomocą następującego wzoru:  $(2.0 * y + x / (y * y)) / 3.0$ . Wzór ten opiera się na metodzie Newtona-Raphsona którą poznałem na jednym z wykładów z przedmiotu „Metody Numeryczne”.
- Następnie sprawdzany jest warunek  $\text{abs}(\text{nextY} - y) < 1e-10$ . Funkcja abs (absolute value - wartość bezwzględna) jest używana w warunku zakończeniowym pętli rekurencyjnej loop. Warunek ten sprawdza, czy różnica między nowym przybliżeniem nextY a starym przybliżeniem y jest mniejsza niż pewna wartość progowa epsilon. Jeśli tak, to znaczy, że znaleźliśmy pierwiastek sześcienny z wystarczającą dokładnością i funkcja zwraca wartość nextY.

- Jeśli warunek nie jest spełniony, to znaczy, że musimy dalej iterować i obliczać kolejne przybliżenie pierwiastka sześciennego. W tym celu funkcja wywołuje się rekurencyjnie, przekazując jako argument nową wartość nextY.

b. Przebieg procedury Herona uzależnić od Epsilon :

Funkcja heronSqrt oblicza pierwiastek kwadratowy liczby x za pomocą metody Herona. Metoda ta polega na iteracyjnym przybliżaniu pierwiastka kwadratowego poprzez obliczanie średniej arytmetycznej liczby x i jej aktualnego przybliżenia.

```
//2
let heronSqrt x epsilon =
  let rec loop y =
    let nextY = (y + x / y) / 2.0
    if abs (nextY - y) < epsilon then
      nextY
    else
      loop nextY
  loop x

printfn "Square root of 25.0 with epsilon 1e-10: %f" (heronSqrt 25.0 1e-10)
```

Tak jak w poprzednim programie tworzymy funkcję heronSqrt przyjmującą dwa parametry x i epsilon. Będzie reprezentowała ona całość obliczeń. Następnie zagnieźdźdźamy w niej dwie kolejne funkcje w tym przypadku najpierw tworzymy funkcję rekurencyjną która pełni rolę „pętli” dla obliczeń ze wzoru Herona nextY gdy warunek abs wartość bezwzględna jest mniejsza od zadanego przybliżenia epsilon zwracamy nextY jako rozwiązanie jeżeli nie to zwracamy wywołanie funkcji określające nowe przybliżenie y.

c. Przebieg procedury Herona uzależnić od n (ilości kroków):

```
//3
let heronSqrtN x n =
  let rec loop y steps =
    if steps = 0 then
      y
    else
      let nextY = (y + x / y) / 2.0
      loop nextY (steps - 1)
  loop x n
```

Do poprzedniej wersji programu zamiast epsilon reprezentującego wartość przybliżenia i warunek przerywania rekurencji zamieniamy go na wartość ilości kroków n i dodajemy parametr steps służący do dekrementacji reszta programu bez zmian.

**Wynik:**

```
Cube root of 25.0: 2.924018
Square root of 25.0 with epsilon 1e-10: 5.000000
Square root of 25.0 with 10 steps: 5.000000
```

### Zadanie 3:

Dany jest ciąg Fibonacciego:

```
(defn fib [n] (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))
```

- Wyznaczyć dokładny wzór opisujący ilość kroków niezbędnych do obliczenia (fib n)
- Zaproponować procedurę rekurencyjną fib, która generuje proces iteracyjny
- Zastosować formę (recur ...) i policzyć Fib(10000)

**Program:**

```
let fibIter n =
  let rec loop a b count =
    if count = 0 then
      a
    else
      loop b (a + b) (count - 1)
  loop 0 1 n
  printfn "Fib(10) = %d" (fibIter 10)

//3
let fib10000 = fibIter 10000
printfn "Fib(10000) = %d" fib10000
```

Przyjmuje jeden argument n, który reprezentuje indeks ciągu Fibonacciego, dla którego chcemy obliczyć wartość. Następnie definiujemy rekurencyjną funkcję loop z trzema argumentami: a jako wartość poprzedniego elementu ciągu Fibonacciego, b jako wartość bieżącego elementu ciągu Fibonacciego, oraz count reprezentujący pozostała liczba iteracji. Funkcja loop sprawdza, czy count jest równe 0. Jeśli tak, to znaczy, że dotarliśmy do końca

rekurencji i zwracamy wartość a, czyli poprzedni element ciągu Fibonacciego. Jeśli count nie jest równe 0, to znaczy, że musimy dalej obliczać kolejne elementy ciągu Fibonacciego. W tym celu funkcja wywołuje się rekurencyjnie, przekazując jako argumenty nową wartość a, która jest równa wartości b. nową wartość b, która jest równa sumie a i b następnie dekrementuje count o 1. Funkcja loop będzie iterować i obliczać kolejne elementy ciągu Fibonacciego, aż do osiągnięcia indeksu n.

Z ciekawostek w programie została wykorzystana tak zwana funkcja ogonowa gdyż rekurencja jest ostatnią wykonywaną instrukcją funkcji, wszystkie argumenty są obliczane przed wywołaniem funkcji rekurencyjnej i nie zwraca żadnej wartości.

### Wynik:

```
Fib(10) = 55
Fib(10000) = 1242044891

E:\GitHub\Funcional programming\Task3\bin\Debug\net8.0\Task3.exe (process 14036)
To automatically close the console when debugging stops, enable Tools->Options->De
le when debugging stops.
Press any key to close this window . . .
```

### Zadanie 4:

Napisać procedurę, która przyjmuje kolekcję elementów (lista, wektor, zbiór) i generuje zbiór potęgowy dla tej kolekcji. Zbiór potęgowy to zbiór wszystkich podzbiorów danego zbioru łącznie ze zbiorem pustym.

$(\text{powerset } [1\ 2\ 3]) \Rightarrow ([\ ]\ [1]\ [2]\ [3]\ [1\ 2]\ [1\ 3]\ [2\ 3]\ [1\ 2\ 3])$

### Program:

```
let rec powerset (coll: 'a list) : 'a list list =
  match coll with
  | [] -> [[]]
  | x::xs ->
    let restPowerset = powerset xs
    restPowerset @ (restPowerset |> List.map (fun subset -> x :: subset))

let result = powerset [1; 2; 3]
printfn "Powerset: %A" result
```

Funkcja powerset sprawdza, czy coll jest pustą listą. Jeśli tak, to znaczy, że dotarliśmy do końca rekurencji i zwracamy listę zawierającą tylko pusty podzbiór [[]]. Jeśli coll nie jest pustą listą, to znaczy, że musimy dalej obliczać podzbiory. W tym celu funkcja dzieli listę coll na dwa elementy: pierwszy element x i pozostałą część listy xs. Następnie funkcja wywołuje się rekurencyjnie dla xs, aby obliczyć zbiór potęgowy pozostałej części listy. Wynik tej rekurencji jest przechowywany w zmiennej restPowerset. Wreszcie funkcja konstruuje dwa nowe podzbiory: pierwszy podzbiór to pusty podzbiór [], drugi podzbiór to każdy podzbiór z restPowerset, do którego dodano element x. Funkcja zwraca listę zawierającą wszystkie obliczone podzbiory: pusty podzbiór oraz podzbiory skonstruowane z restPowerset i x. W funkcji pojawiają się dwie nowe rzeczy takie jak instrukcja match która w tym kodzie służy do dopasowania wartości argumentu coll do wzorców. Coll w tym przypadku definiują dwa przypadki, które mogą wystąpić: []: Ten wzór dopasowuje pustą listę, x::xs: Ten wzór dopasowuje listę zawierającą co najmniej jeden element x oraz pozostałą część listy xs..

**Wynik:**

```
PowerSet: [[]; [3]; [2]; [2; 3]; [1]; [1; 3]; [1; 2]; [1; 2; 3]]
E:\GitHub\Funcional programming\Task4\bin\Debug\net8.0\Task4.exe (proc
To automatically close the console when debugging stops, enable Tools->
le when debugging stops.
Press any key to close this window . . .
```

### Zadanie 5:

- a. Zrealizuj pierwiastek sześcienny z wykorzystaniem average-damp oraz FIXED-POINT

```
//1
let fixedPoint f firstGuess tolerance =
  let rec loop guess =
    let next = f guess
    if abs (next - guess) < tolerance then
      next
    else
      loop next
  loop firstGuess

let averageDamp f =
  fun x -> (x + f x) / 2.0

let cubeRoot x =
  let cube y = x / (y * y)
  fixedPoint (averageDamp cube) 1.0 1e-10

printfn "Cube root of 27.0 (Average-damp): %f" (cubeRoot 27.0)
```



Program, który napisałem, oblicza pierwiastek sześcienny liczby za pomocą metody średniego tłumienia (average damping) oraz funkcji punktu stałego (fixed point).

- Funkcja fixedPoint- Zaczniemy od tego czym jest FixedPoint. Koncepcja punktu stałego polega na znalezieniu takiej wartości, dla której funkcja  $f$  zastosowana do tej wartości zwraca wartość bardzo bliską do niej samej. Nasza funkcja przyjmuje trzy argumenty: funkcję  $f$ , początkowe przybliżenie firstGuess oraz tolerancję tolerance. Rekurencyjna funkcja loop oblicza wartość next jako wynik zastosowania  $f$  do bieżącego przybliżenia guess. Jeśli różnica między next a guess jest mniejsza niż tolerance, zwraca next jako wynik. W przeciwnym razie kontynuuje rekurencję, używając next jako nowego przybliżenia. Funkcja ta iteracyjnie zbliża się do punktu stałego funkcji  $f$ , czyli takiego punktu, w którym  $f(guess)$  jest bardzo bliskie guess.
- Funkcja averageDamp: Average damping to technika, która pomaga funkcji zbiegać szybciej do punktu stałego przez uśrednianie wartości funkcji z jej argumentem. Ta funkcja przyjmuje funkcję  $f$  i zwraca nową funkcję, która dla danego argumentu  $x$  oblicza średnią wartości  $x$  i  $f(x)$ . Funkcja averageDamp działa jako technika stabilizacji, pomagająca funkcji szybciej osiągnąć punkt stały.

b. Zrealizuj pierwiastek sześcienny z wykorzystaniem Newtons-method

```
//2
let newtonCubeRoot x =
  let improve guess = (2.0 * guess + x / (guess * guess)) / 3.0
  let goodEnough guess = abs ((guess ** 3.0) - x) < 1e-10
  let rec loop guess =
    if goodEnough guess then
      guess
    else
      loop (improve guess)
  loop 1.0

printfn "Cube root of 27.0 (Newton's method): %f" (newtonCubeRoot 27.0)
```

**Metoda Newtona omówiona w zad 2.**

c. Niech  $f$  i  $g$  będą dwoma funkcjami jednoargumentowymi. Złożenie  $f$  z  $g$  jest określone jako funkcja  $x \rightarrow f(g(x))$ . Zdefiniuj procedurę `złóż` implementującą złożenie funkcji. Przykładowo:  $((\text{złóż kwadrat inc}) 6) \Rightarrow 49$

```
//3
let compose f g =
  fun x -> f (g x)

let square x = x * x
let inc x = x + 1
let composedFunction = compose square inc
printfn "Result of composed function (square and inc) for 6: %d" (composedFunction 6)
```

W tym przykładzie możemy zauważyć wykorzystanie tak zwanej kompozycji funkcji która jest potężną techniką programowania funkcyjnego. Umożliwia łączenie prostych funkcji w bardziej złożone operacje co możemy zauważyć w `compose`. `Compose` przyjmuje dwie funkcje  $f$  i  $g$  i tworzy nową funkcję, która najpierw wywołuje  $g$ , a następnie  $f$  na wyniku  $g(x)$ . Dla `composedFunction 6`, najpierw wykonuje się `inc 6`, co daje 7, a następnie `square 7`, co daje 49.

d. Jeśli  $f$  jest funkcją jednoargumentową określoną na liczbach oraz  $n$  jest dowolną liczbą naturalną, to  $n$ -krotnym złożeniem funkcji  $f$  nazywamy funkcję, której wartością jest wynik  $n$ -krotnego zastosowania funkcji  $f$ :  $x \rightarrow f(f(\dots(f(x))\dots))$  Napisz procedurę realizującą  $n$ -krotne złożenie funkcji  $f$  wykorzystując rozwiązanie z punktu c.

```
//4
let rec repeatCompose f n =
  if n = 1 then
    f
  else
    compose f (repeatCompose f (n - 1))

let repeatInc3 = repeatCompose inc 3
printfn "Result of repeatCompose (inc 3 times) for 6: %d" (repeatInc3 6)
```

Wykorzystujemy te samo założenie co z poprzedniego punktu dotyczącego tworzenia złożonych funkcji ale uzależniamy je od n krotnego złożenia tej funkcji. Repeat Compose używa rekurencji, aby aplikować funkcję  $f$  do wyniku poprzednich aplikacji, co w efekcie prowadzi do złożenia  $f$  sama z sobą  $n$  razy

**Wynik:**

```
Cube root of 27.0 (Average-damp): 3.000000
Cube root of 27.0 (Newton's method): 3.000000
Result of composed function (square and inc) for 6: 49
Result of repeatCompose (inc 3 times) for 6: 9

E:\GitHub\Funcional programming\Task5\bin\Debug\net8.0\
To automatically close the console when debugging stops,
le when debugging stops.
Press any key to close this window . . .
```