Rekurencja

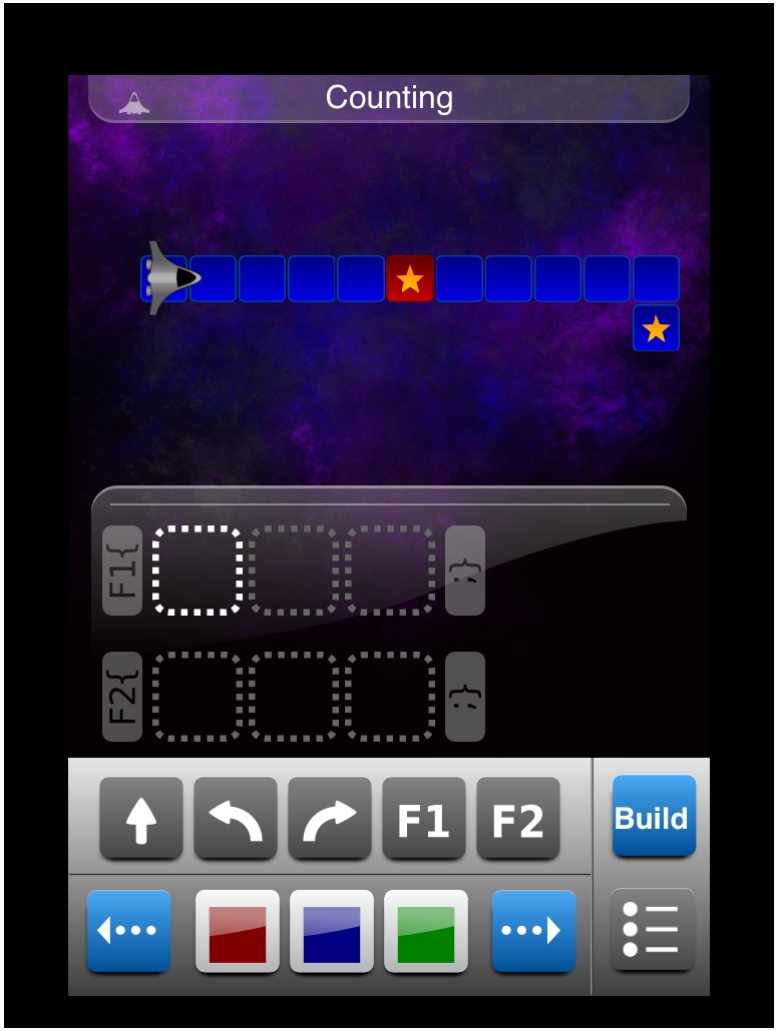
- funkcje rekurencyjne
- rozwiązywanie problemów
- niewłaściwe użycie rekurencji
- postać ogonowa rekurencji
- przekształcenie do postaci ogonowej
- zamiana postaci ogonowej na iterację

Rekurencja

- Rekurencją nazywamy metodę specyfikacji procesu w terminach definiowanego procesu.
- Dokładniej mówiąc, „skomplikowane” przypadki procesu redukowane są do „prostszych” przypadków.
- Stałymi składnikami metod rekurencyjnych są określenie warunku zatrzymania, oraz określenie metody redukcji problemu do prostszych przypadków.

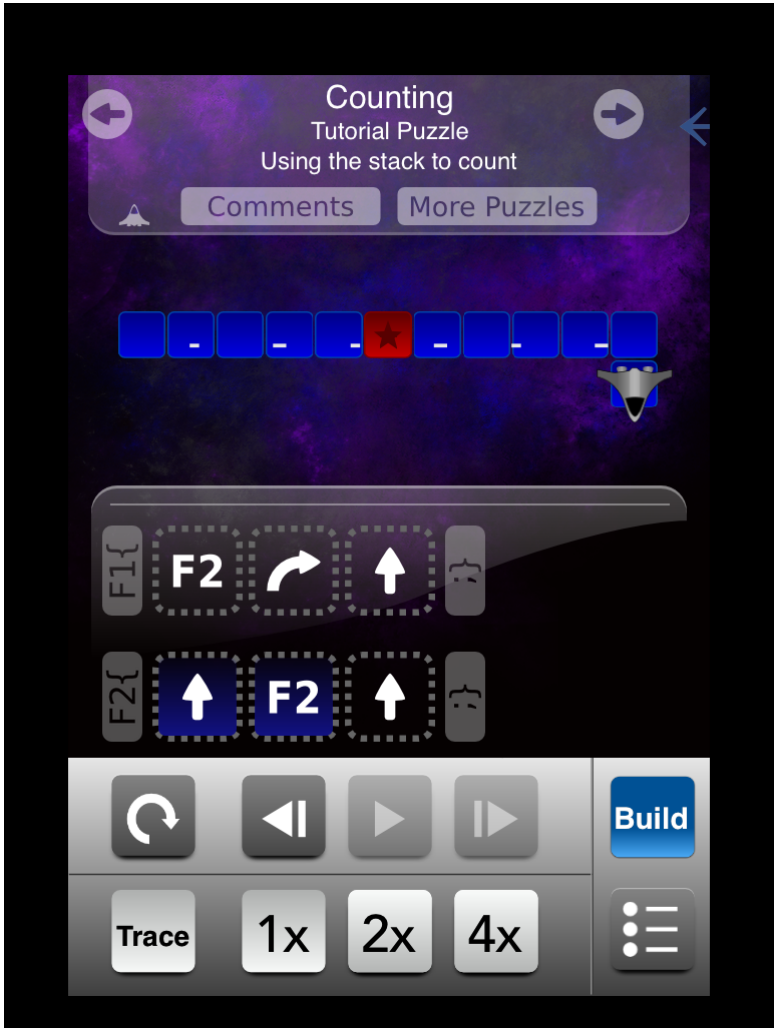
Rekurencja

Przykład z gry Robozzle: jak zebrać obie gwiazdki?



Rekurencja

Przykład z gry Robozzle: rozwiązanie



Rekurencja

Funkcje rekurencyjne

Example (Funkcja silnia)

Klasycznym przykładem funkcji zdefiniowanej rekurencyjnie jest silnia:

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

Jeśli jednak zauważymy, że $0! = 1$ oraz, że $n! = n \cdot (n-1)!$, to możemy napisać następującą funkcję:

```
1: function Silnia(n)
2:     if n = 0 then
3:         Silnia ← 1
4:     else
5:         Silnia ← n * Silnia(n − 1)
6:     end if
7: end function
```

Rekurencja

Funkcje rekurencyjne

Example (Funkcja Ackermana)

Funkcją Ackermana nazywamy funkcję dwóch zmiennych naturalnych zdefiniowaną wzorem

$$A(n,m) = \begin{cases} m+1 & \text{gdy } n = 0, \\ A(n-1,1) & \text{gdy } n > 0 \wedge m = 0, \\ A(n-1,A(n,m-1)) & \text{gdy } n > 0 \wedge m > 0. \end{cases}$$

Uwaga: powyższa funkcja bardzo szybko rośnie. Dla przykładu

$$A(4,2) = 2^{65536} - 3$$

ma 19729 cyfr.

Ćwiczenie: udowodniej, że dla każdych nieujemnych całkowitych wartości n i m obliczenia funkcji Ackermana $A(n,m)$ kończą się.

Rekurencja

Funkcje rekurencyjne

Example (Funkcja Ackermana cd.)

W pseudokodzie możemy zapisać ją następująco:

```
1:  function A(n, m)
2:      if n = 0 then
3:          A ← m + 1
4:      else
5:          if m = 0 then
6:              A ← A(n − 1, 1)
7:          else
8:              A ← A(n − 1, A(n, m − 1))
9:          end if
10:     end if
11: end function
```

Rekurencja

Rozwiązywanie problemów

- Rekursję można stosować nie tylko do definiowania funkcji.
- Można ją stosować do rozwiązywania problemów.
- W tym przypadku metoda polega na redukcji złożonego problemu na problemy prostsze.

Rekurencja

Rozwiązywanie problemów: wieże z Hanoi

- Mamy trzy patyki ponumerowane liczbami 1, 2 i 3 oraz n krążków.
- Krążki są różnych rozmiarów.
- Początkowo wszystkie krążki nawleczone są na pierwszy patyk w malejącej kolejności: największy leży na dole a najmniejszy na samej górze.
- Należy przenieść krążki na trzeci patyk przestrzegając następujących dwóch reguł:
  1. wolno przekładać tylko jeden krążek w jednym ruchu,
  2. nigdy większy krążek nie może zostać położony na mniejszym krążku.

Rekurencja

Rozwiązywanie problemów: wieże z Hanoi

- Zadanie jest banalne gdy $n = 0$, bo nic nie trzeba wtedy przenosić.
- Rozwiązanie całego zadania stanie się jasne, gdy zauważmy, że jeśli potrafimy przenieść $n - 1$ krążków z patyka i-tego na patyk j-ty, to potrafimy zadanie to rozwiązać dla n krążków.
- Oto strategia:
  1. wyznacz pomocniczy patyk $k = 6 - (i + j)$;
  2. przenieś górne $n - 1$ krążków z patyka i-tego na patyk k-ty;
  3. przenieś krążek z patyka i-tego na patyk j-ty;
  4. przenieś górne $n - 1$ krążków z patyka k-tego na patyk j-ty

Rekurencja

Rozwiązywanie problemów: wieże z Hanoi

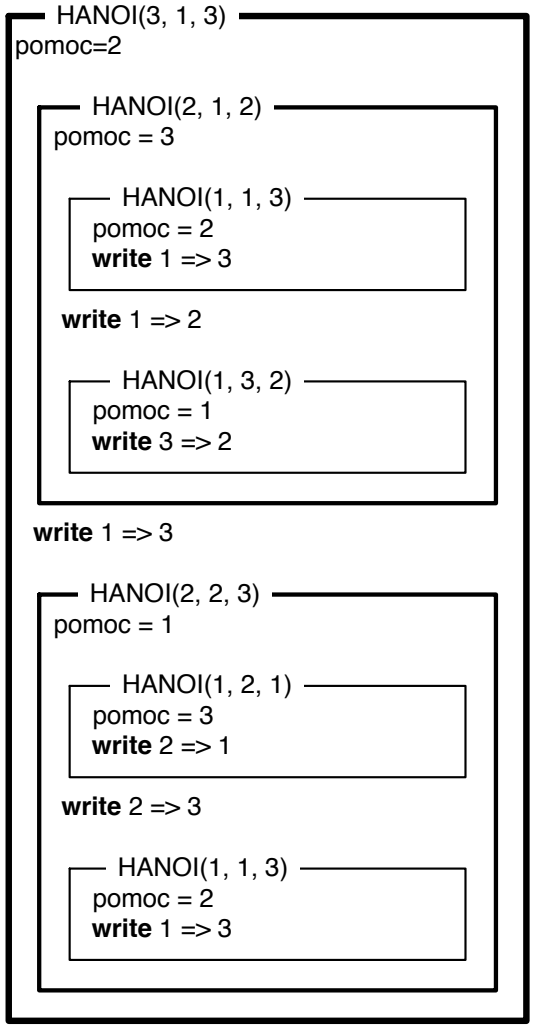
Oto jak tę strategię możemy zamienić na program w pseudokodzie:

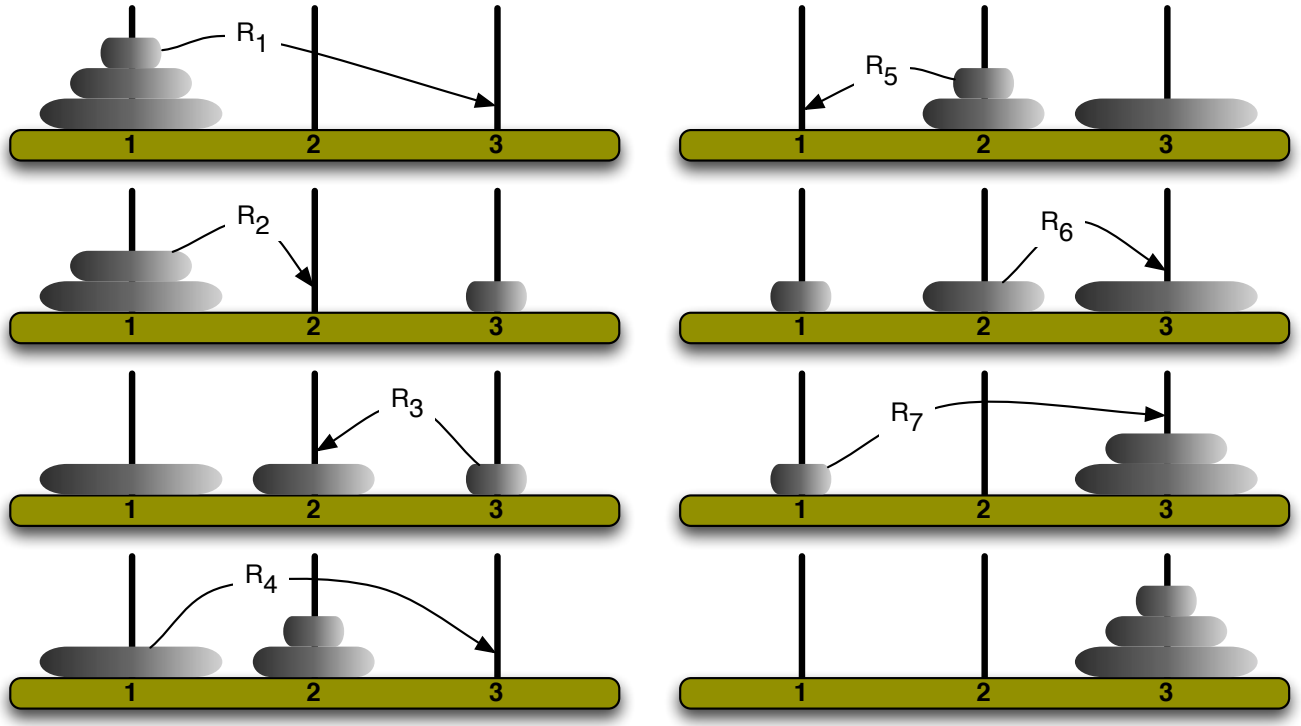
```
1: procedure Hanoi(n, skąd, dokąd)
2:     if n > 0 then
3:         pomoc ← 6 − (skąd + dokąd);
4:         Hanoi(n − 1, skąd, pomoc);
5:         write skąd ⇒ dokąd;
6:         Hanoi(n − 1, pomoc, dokąd)
7:     end if
8: end procedure
```

Rekurencja

Rozwiązywanie problemów: wieże z Hanoi

Przykład obliczeń `Hanoi(3, 1, 3)`:





Rekurencja

Rozwiązywanie problemów: wieże z Hanoi

- Problem Wież z Hanoi wymyślił Edouard Lucas w roku 1883.
- Według legendy którą wtedy opowiedział, w pewnej hinduskiej świątyni mnisi przekładają bez przerwy układ 64 krążków zgodnie z regułami wież Hanoi.
- Po zrealizowaniu zadania świat ma się zakończyć.

Rekurencja

Rozwiązywanie problemów: wieże z Hanoi

- Niech $H(n)$ oznacza ilość operacji przeniesienia krążka, którą wykonuje nasza procedura dla konfiguracji n krążków.
- Oczywiście $H(1) = 1$ oraz $H(n+1) = H(n) + 1 + H(n) = 2H(n) + 1$.
- Zajmiemy się teraz wyznaczeniem wzoru na $H(n)$.
- Wypiszmy kilka pierwszych wartości:

$$\begin{cases} H(1) &= 1 \\ H(2) &= 2H(1) + 1 \\ H(3) &= 2H(2) + 1 \\ H(4) &= 2H(3) + 1 \end{cases}$$

Rekurencja

Rozwiązywanie problemów: wieże z Hanoi

- Pomnóżmy pierwszą równość przez 8, drugą przez 4 a trzecią przez 2. Otrzymamy

$$\begin{cases} 8H(1) &= & 8 \\ 4H(2) &= & 8H(1)+4 \\ 2H(3) &= & 4H(2)+2 \\ H(4) &= & 2H(3)+1 \end{cases}$$

- Po zsumowaniu wszystkich równości stronami otrzymamy

$$H(4) = 1+2+4+8 = 1+2+2^2+2^3 = 2^4-1,$$

co powinno nam nasunąć następującą hipotezę:

$$(\forall n > 0)(H(n) = 2^n - 1).$$

- Jest ona prawdziwa, co łatwo możemy sprawdzić indukcją matematyczną.

Rekurencja

Niewłaściwe użycie rekurencji

- Pisząc procedurę rekurencyjną musimy zadbać o to aby przy każdym dopuszczalnym zestawie parametrów wejściowych zatrzymywała się ona po skończonej liczbie kroków.

```
1: function R(n)
2:     R ← R(n − 1)
3: end function
```

- W powyższej funkcji brak przypadku kończącego wywołania rekurencyjne (najprostszego przypadku, który nie wymaga wywołania rekurencyjnego).

Rekurencja

Niewłaściwe użycie rekurencji: współczynnik Newtona

Załóżmy, że chcemy wyznaczyć wartość *współczynnika Newtona*

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

korzystając z dobrze znanej równości Pascala

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k},$$

oraz z dwóch dodatkowych wzorów $\binom{n}{0} = \binom{n}{n} = 1$.

Rekurencja

Niewłaściwe użycie rekurencji: współczynnik Newtona

Zapiszemy to w pseudokodzie:

```
1: function Newton0(n, k)
2:     if k = 0 ∨ k = n then
3:         Newton0 ← 1
4:     else
5:         Newton0 ← Newton0(n − 1, k − 1) + Newton0(n − 1, k)
6:     end if
7: end function
```

Rekurencja

Niewłaściwe użycie rekurencji: współczynnik Newtona

Przyjrzyjmy się jednak, jakie obliczenia będą wykonywane przez tak zaprogramowaną funkcję dla parametrów (7,4):

$$\binom{7}{4} \to \left\{\binom{6}{3}, \binom{6}{4}\right\} \to \left\{\left\{\binom{5}{2}, \binom{5}{3}\right\}, \left\{\binom{5}{3}, \binom{5}{4}\right\}\right\} \to$$

$$\left\{\left\{\left\{\binom{4}{1}, \binom{4}{2}\right\}, \left\{\binom{4}{2}, \binom{4}{3}\right\}\right\}, \left\{\left\{\binom{4}{2}, \binom{4}{3}\right\}, \left\{\binom{4}{3}, \binom{4}{4}\right\}\right\}\right\} \to \ldots$$

Prowadzi to do wykładniczej liczby wywołań rekurencyjnych, które wielokrotnie liczą tę samą wartość.

Rekurencja

Niewłaściwe użycie rekurencji: współczynnik Newtona

Można skorzystać z innej zależności:

$$\binom{n}{k} = \binom{n-1}{k-1} \cdot \frac{n}{k},$$

który prowadzi do następującego kodu:

```
1: function Newton(n, k)
2:     if k = 0 ∨ k = n then
3:         Newton ← 1
4:     else
5:         Newton ← (Newton(n − 1, k − 1) ∗ n) div k
6:     end if
7: end function
```

o czasowej złożoności obliczeniowej $O(n)$.

Rekurencja

Niewłaściwe użycie rekurencji: ciąg Fibonacciego

Rozpatrzmy przykład ciągu Fibonacciego, w którym kolejny wyraz jest sumą dwóch poprzednich:

```
1:  function Fib0(n)
2:      if n = 0 then
3:          Fib0 ← 0
4:      else
5:          if n = 1 then
6:              Fib0 ← 1
7:          else
8:              Fib0 ← Fib0(n − 1) + Fib0(n − 2)
9:          end if
10:     end if
11: end function
```

Rekurencja

Niewłaściwe użycie rekurencji: ciąg Fibonacciego

$$
\begin{aligned}
\text{Fib0}(5) &= \text{Fib0}(4) + \text{Fib0}(3) \\
&= \text{Fib0}(3) + \text{Fib0}(2) + \text{Fib0}(3) \\
&= \text{Fib0}(2) + \text{Fib0}(1) + \text{Fib0}(2) + \text{Fib0}(3) \\
&= \text{Fib0}(1) + \text{Fib0}(0) + \text{Fib0}(1) + \text{Fib0}(2) + \text{Fib0}(3) \\
&= 1 + \text{Fib0}(0) + \text{Fib0}(1) + \text{Fib0}(2) + \text{Fib0}(3) \\
&= 1 + 0 + \text{Fib0}(1) + \text{Fib0}(2) + \text{Fib0}(3) \\
&= 1 + 0 + 1 + \text{Fib0}(2) + \text{Fib0}(3) \\
&= 1 + 0 + 1 + \text{Fib0}(1) + \text{Fib0}(0) + \text{Fib0}(3) \\
&= 1 + 0 + 1 + 1 + \text{Fib0}(0) + \text{Fib0}(3) \\
&= 1 + 0 + 1 + 1 + 0 + \text{Fib0}(3) \\
&= 1 + 0 + 1 + 1 + 0 + \text{Fib0}(2) + \text{Fib0}(1) \\
&= 1 + 0 + 1 + 1 + 0 + \text{Fib0}(1) + \text{Fib0}(0) + \text{Fib0}(1) \\
&= 1 + 0 + 1 + 1 + 0 + 1 + \text{Fib0}(0) + \text{Fib0}(1) \\
&= 1 + 0 + 1 + 1 + 0 + 1 + 0 + \text{Fib0}(1) \\
&= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 = 5
\end{aligned}
$$

Rekurencja

Niewłaściwe użycie rekurencji: ciąg Fibonacciego

```
 1: function Fib(n)
 2:     if n = 0 then
 3:         Fib ← 0
 4:     else
 5:         if n = 1 then
 6:             Fib ← 1
 7:         else
 8:             starsza ← 0; stara ← 1;
 9:             for i ← 2, n do
10:                 nowa ← stara + starsza;
11:                 starsza ← stara; stara ← nowa
12:             end for
13:             Fib ← nowa
14:         end if
15:     end if
16: end function
```

Rekurencja

Niewłaściwe użycie rekurencji: ciąg Fibonacciego

Uwaga

Stosując operacje na macierzach, jakie poznałeś na kursie **Algebra z geometrią analityczną**, można policzyć n-ty wyraz ciągu Fibonacciego za pomocą $O(\log n)$ iteracji.

Ćwiczenie: Niech macierz A ma następujące wartości elementów:

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Jakie są wartości elementów macierzy A^n, tj. w n-tej potędze macierzy A?

Rekurencja

Postać ogonowa

Powróćmy do rekurencyjnej funkcji `Silnia(n)` obliczającej $n!$.
Poniżej przedstawiono przebieg obliczeń Silnia(5):

$$\begin{aligned}
\text{Silnia}(5) &= 5 * \text{Silnia}(4) \\
&= 5 * 4 * \text{Silnia}(3) \\
&= 5 * 4 * 3 * \text{Silnia}(2) \\
&= 5 * 4 * 3 * 2 * \text{Silnia}(1) \\
&= 5 * 4 * 3 * 2 * 1 * \text{Silnia}(0) \\
&= 5 * 4 * 3 * 2 * 1 * 1 \\
&= 5 * 4 * 3 * 2 * 1 \\
&= 5 * 4 * 3 * 2 \\
&= 5 * 4 * 6 \\
&= 5 * 24 = 120
\end{aligned}$$

Rekurencja

Postać ogonowa

- Potrzeba przemnażania wartości Silnia$(n-1)$ przez n wymaga przechowywania zmiennej n na stosie (po powrocie z rekurencyjnego wywołania zmienna n musi mieć taką wartość jak przed rekurencyjnym wywołaniem).
- Gdyby po powrocie z rekurencyjnego wywołania nie wykonywała się już żadna operacja na zmiennych lokalnych, to nie zachodziłaby potrzeba przechowywania ich na stosie.
- Aby było to możliwe funkcja rekurencyjna powinna spełniać następujące warunki:
  1. Rekurencyjne wywołanie funkcji znajduje się w jednym miejscu jej definicji.
  2. Po powrocie z rekurencyjnego wywołania nie mogą być już wykonywane żadne operacje na zmiennych lokalnych funkcji.

Rekurencja

Przekształcanie do postaci ogonowej

```
1:  function Silnia2(n, akumulator)
2:      if n = 0 then
3:          Silnia2 ← akumulator
4:      else
5:          Silnia2 ← Silnia2(n − 1, n ∗ akumulator)
6:      end if
7:  end function
8:
9:  function Silnia(n)
10:     Silnia ← Silnia2(n, 1)
11: end function
```

Rekurencja

Przekształcanie do postaci ogonowej

Przebieg obliczeń:

$$
\begin{aligned}
\text{Silnia}(5) &= \text{Silnia2}(5,1) \\
&= \text{Silnia2}(4,5) \\
&= \text{Silnia2}(3,20) \\
&= \text{Silnia2}(2,60) \\
&= \text{Silnia2}(1,120) \\
&= \text{Silnia2}(0,120) \\
&= 120
\end{aligned}
$$

Rekurencja

Zamiana postaci ogonowej na iterację

```
1: function Silnia(n)
2:     akumulator ← 1;
3:     while n ≠ 0 do
4:         akumulator ← akumulator ∗ n;
5:         n ← n − 1
6:     end while
7:     Silnia ← akumulator
8: end function
```