W. 1

Rekurencja vs Iteracja

Czym jest rekurencja?

- „Żeby zrozumieć **rekurencję**, trzeba zrozumieć **rekurencję**”.
- Rekurencja jest techniką programowania polegającą na wywoływaniu funkcji przez samą siebie.
- Innymi słowy, program może być redukowany do tego samego problemu z ‘mniejszą liczbą danych do przetworzenia’.
- Większość nowoczesnych języków programowania wspiera rekurencję – funkcja wywołująca samą siebie.

Silnia

Silnia (ang. Factorial)

Rekurencyjna definicja funkcji silnia:

- 0!=1 (warunek stopu dla rekurencji)
- n!=n* (n-1)! (definicja rekurencyjna)

- Przykład:

3!

3*2!

3*2*1!

3*2*1*0!

3*2*1*1 (używamy warunku stopu: 0!=1)

3*2*1

3*2

6

1) Proszę zauważyć, że objętość danych w pamięci (liczb) najpierw rośnie, a później maleje. Jest to sytuacja problemowa, gdy mamy duży problem i mało pamięci!
2) Istnieje rekurencja (rekurencja ogonowa), która wykonuje się w stałej objętości pamięci – poznamy ją później.

Silnia

Proszę zaimplementować program obliczający silnię rekurencyjnie.

(Python i C++)

Silnia

Rozwiązanie (Python):

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Proszę zlokalizować

- warunek stopu;
- rekurencyjne wywołanie funkcji;
- Proszę obliczyć factorial(10), factorial(0); Funkcja silnia bardzo szybko rośnie i dla dużych wartości **n** komputer nie będzie w stanie obliczyć tej funkcji.
- Dlaczego dla **n<0** otrzymujemy Stack Overflow (Przepełnienie stosu)?

Silnia

Rozwiązanie (C++):

```
#include <iostream>
using namespace std;

int f(int n){
  if (n == 0)
    return 1;
  else
     return n*f(n-1);}

int main(){
  int num;
  cout<<"Enter a number: ";
  cin>>num;
  cout<<"Factorial of entered number: "<<f(num);
  return 0;}
```

1) Proszę skompilować i uruchomić.
2) Proszę usunąć problem przepełnienia stosu dla ujemnych argumentów.

Rekurencja vs iteracja

Podejście iteracyjne

Alternatywnym podejściem do rekurencji jest iteracja.

Iterację wykonujemy w pętli, np. for.

Iteracja wykonuje się w stałej objętości pamięci.

Obecnie iteracja i rekurencja wykonują się w podobnym czasie – kompilatory/interpretery są na tyle ‘inteligentne’, że potrafią zoptymalizować kod.

Podejście iteracyjne nie jest funkcyjne! Dlatego iterację w paradygmacie funkcyjnym przepisujemy na rekurencję.

Dobrze jest umieć przepisywać problemy rekurencyjne na iteracyjne i vice versa.

Silnia, rekurencyjnie

- Python:

```
n = 3
fact = 1
for i in range(1,n+1):
    fact = fact * i
print('fact(3)=', fact)
```

- Sekwencja wywołania:

fact = 1 (wejście do pętli)

fact=1*1

fact=1*2

fact=2*3

fact=6 (wyjście z pętli)

1) Proszę zauważyć, że kod wykonuje się w stałej objętości pamięci.
2) Proszę ten kod opakować w funkcję.
3) Proszę ten kod przepisać w C++.

Rekurencja ogonowa

(Tail recursion)

Rekurencja ogonowa (Tail recursion)

- Rekurencja ogonowa jest sposobem przepisania rekurencji, aby wykonywała się w stałej objętości pamięci.
- Często takie podejście wygląda podobnie do iteracji.
- Takie podejście wymaga zazwyczaj:
  - „Interfejsu” – funkcji która będzie puntem wejścia dla użytkownika.
  - Funkcji rekurencyjnej, która wykorzystuje specjalną zmienną (‘akumulator’) do przekazywania obliczanego wyrażenia z poprzedniego poziomu do następnego
  - Często interfejs może być też funkcją (z domyślnymi wartościami akumulatora).

Silnia ‘ogonowa’

- Rozwiązanie (Python):

```
#rekurencyjna funkcja
def tail_factorial(n, accumulator=1):
  if n == 0:
     return accumulator
  else:
     return tail_factorial(n-1, accumulator * n)
#interfejs
def factorial(n):
return tail_factorial(n, accumulator=1)
#wywołanie
print("3!=",factorial(3))
```

- Sekwencja wywołania:
  - **factorial(3)**
  - **tail_f(3, 1)**
  - **tail_f(2, 3)**
  - **tail_f(1, 6)**
  - **tail_f(0, 6)**
  - **return(6)** (warunek stopu n==0)

- Proszę zauważyć, że wszystko wywołuje się w stałej pamięci, a akumulator przekazuje kolejne wartości pomiędzy poziomami zagnieżdżenia.
- Proszę przepisać kod w C++.

Porównanie

Porównanie

```
#iteracja
n = 3
fact = 1
for i in range(1,n+1):
    fact = fact * i
print('fact(3)=', fact)
```

```
#Standardowe podejście
def factorial(n
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

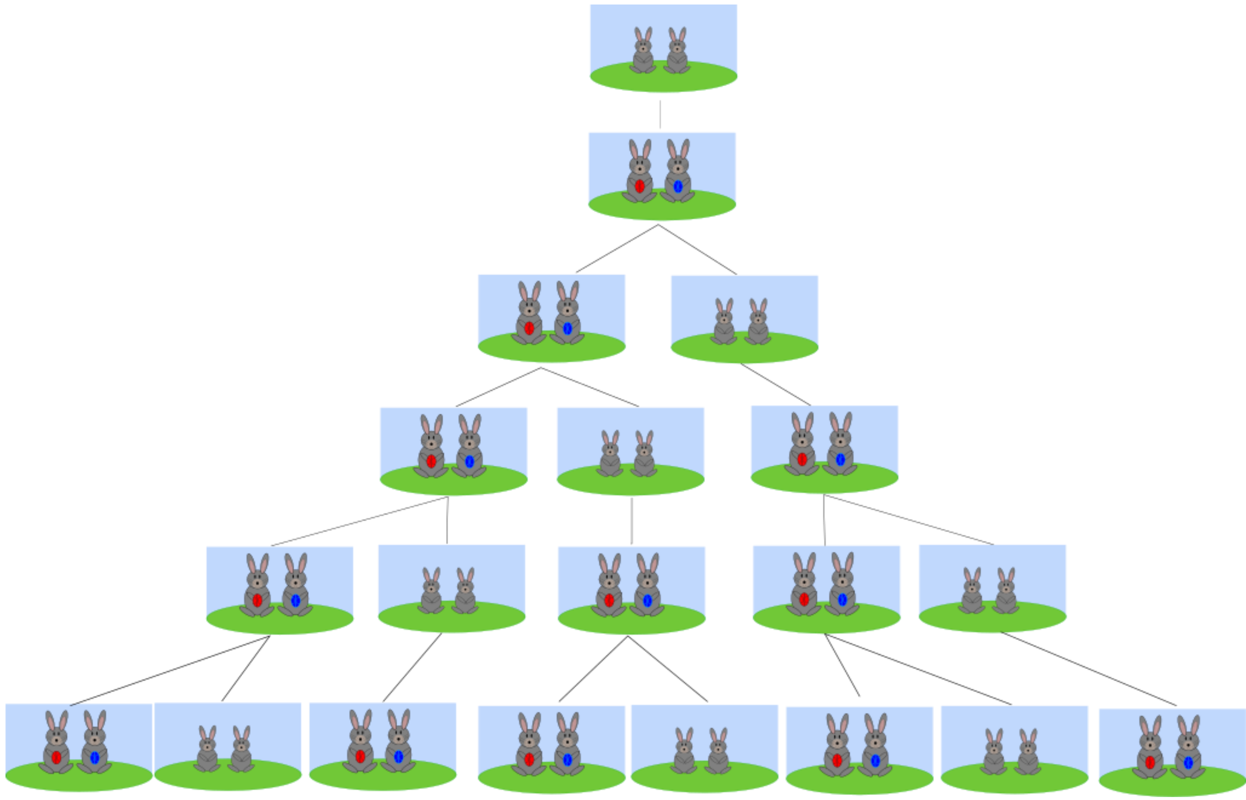
```
#rekurencja ogonowa
def tail_factorial(n, accumulator=1):
  if n == 0:
    return accumulator
  else:
    return tail_factorial(n-1, accumulator * n)
```

- Iteracja i rekurencja ogonowa praktycznie się nie różni.
- Standardowe podejście rekurencyjne jest proste ale zużywa dużo pamięci.

Przykład
Ciąg Fibonacciego

Historia

- Ciąg został omówiony w roku 1202 przez Leonarda z Pizy, zwanego Fibonaccim, w dziele Liber abaci jako rozwiązanie zadania o rozmnażaniu się królików. Nazwę „ciąg Fibonacciego” spopularyzował w XIX w. Édouard Lucas



Definicja

- Definicja:

  Warunek stopu:

    - F(0)=0
    - F(1)=1

  – Wywołanie rekurencyjne:

    - F(n)=F(n-1)+F(n-2)   (dla n>1)

- Początkowe wartości: 0,1,1,2,3,5,8,13,...

Implementacja (Python)

Python:

- **def f(n):**
  - **if n == 0:**
    - **return 0**
  - **elif n == 1:**
    - **return 1**
  - **else:**
    - **return f(n - 2) + f(n – 1)**

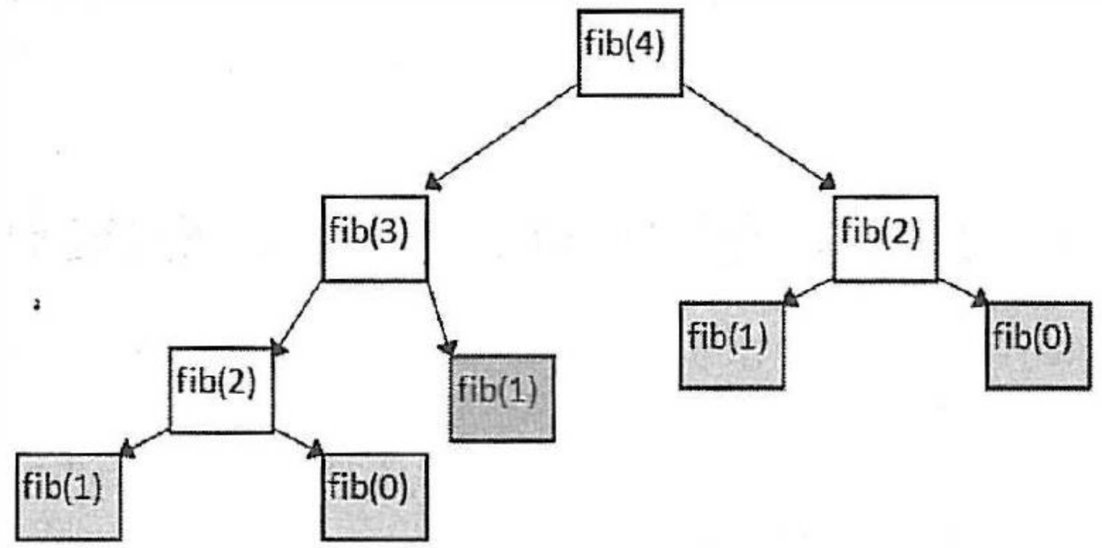
- Proszę uruchomić program.
- Proszę zabezpieczyć go przed przepełnieniem stosu (dla n<0).
- Proszę zaimplementować program w C++.

Implementacja (C++)

- (C++):
  - unsigned long int F(int x)
  - {
    - if (x < 2)
      - return x ;
    - else
      - return F(x - 1)+F(x- 2) ;

Proszę uruchomić i przetestować kod.

Sekwencja wywołań rekurencyjnych dla F(4)



- Proszę zobaczyć, że fib(2) występuje w lewym i prawym poddrzewie – algorytm nie jest zbyt efektywny, gdyż powtarza swoje obliczenia, nie wykorzystując już raz obliczonych wartości.
- Jak można poprawić takie nieefektywne algorytmy?

Programowanie dynamiczne

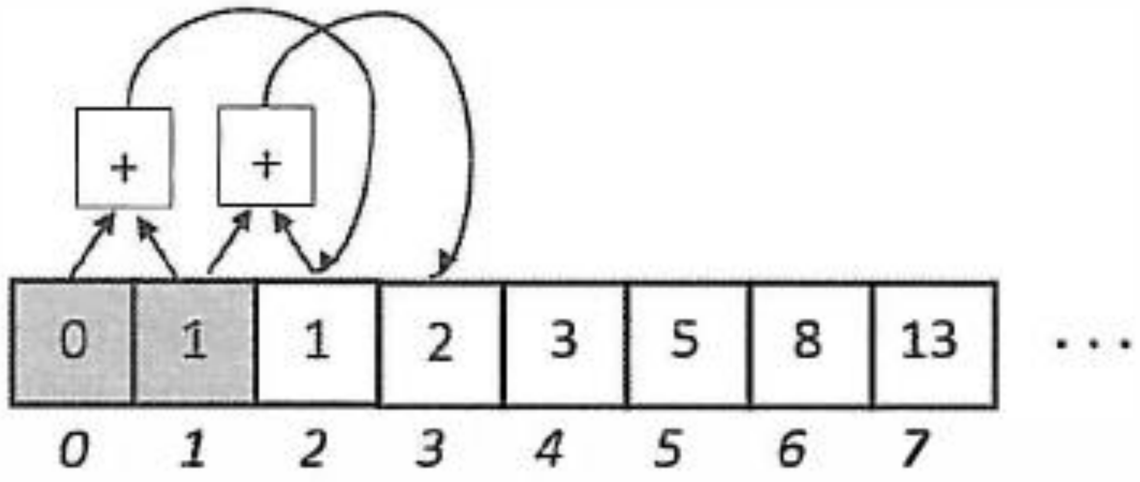
Programowanie dynamiczne

- Odkrywca: Richard E. Bellman (1979)
- Jest to technika optymalizacji programów rekurencyjnych w sposób iteracyjny.
- Idea: Zapamiętą wartości już obliczone i wykorzystuj je w momencie, gdy są potrzebne.
- Etapy:
  - Koncepcja:
    - Stwórz rekurencyjny algorytm.
    - Stwórz tablicę na obliczone wartości pośrednie.
  - Indukcja:
    - Wpisz do tablicy wartości elementarne (startowe) dla rekurencji.
  - Progresja:
    - Uzupełniaj kolejne wartości w tablicy używając wzoru rekurencyjnego.
    - Zatrzymaj się gdy dojdziesz do wartości która Cię interesuje.

Liczby Fibonacciego ‘dynamicznie’

```
#include<stdio.h>
int fib(int n)
{   int f[n+2];   // 1 extra to handle case, n = 0
  int i;
  /* 0th and 1st number of the series are 0 and 1*/
  f[0] = 0;
  f[1] = 1;
  for (i = 2; i <= n; i++)
  {      /* Add the previous 2 numbers in the series and store it */
     f[i] = f[i-1] + f[i-2]; }
  return f[n];
}
int main ()
{   int n = 9;
  printf("%d", fib(n));
  getchar();
  return 0; }
```

- Proszę zauważyć, że f jest tablicą, a obliczenie jest iteracyjne, a nie rekurencyjne
- Proszę zaimplementować program w Pythonie.



Fibonacci ‘ogonowy’

Fibonacci ‘ogonowo’

Proszę zaimplementować algorytm obliczania ciągu Fibonacciego używając rekurencji ogonowej w Pythonie.

Wskazówka: potrzebujesz dwóch akumulatorów, aby pamiętać dwa poprzednie wyrazy ciągu w celu obliczenia kolejnego.

Rozwiązanie (Python)

- **def fib_tail(n, a = 0, b = 1):**
-     if n == 0:
-         return a
-     if n == 1:
-         return b
-     #print("n=",n,"a=",a,"b=",b) #odkomentuj, aby zobaczyć sekwencję wywołań
-     return fib_tail(n - 1, b, a + b);
-
- **def fib(n):**
-     return fib_tail(n,a=0,b=1)

- **n=7**
- print("fib("+str(n)+") = "+str(fib(n)))

- Sekwencja wywołań:
  - ('n=', 7, 'a=', 0, 'b=', 1)
  - ('n=', 6, 'a=', 1, 'b=', 1)
  - ('n=', 5, 'a=', 1, 'b=', 2)
  - ('n=', 4, 'a=', 2, 'b=', 3)
  - ('n=', 3, 'a=', 3, 'b=', 5)
  - ('n=', 2, 'a=', 5, 'b=', 8)

- Zaimplementuj ten algorytm w C++.
- Zauważ, że startujemy od a=0, b=1, czyli od wartości początkowych ciągu Fibonacciego i obliczamy kolejne wyrazy. To jest analogiczne do podejścia iteracyjnego\dynamicznego i jest równie efektywne pamięciowo, ale nie musimy pamiętać tablicy.
- Zauważ też, że to podejście nie ma efektów ubocznych – wywołujemy funkcję bez takich efektów.

Przerywnik

Liczby Fibonacciego

Znany fakt

- Teraz możesz poznać najprostszy sposób.
  - F(n)= (phi^n -psi^n)/sqrt(5),
  - phi = (1+sqrt(5))/2 (złoty podział)
  - psi = (1-sqrt(5))/2

- Zaimplementuj ten sposób i sprawdź, czy wynik się zgadza. Jak poradzisz sobie z konwersją liczb zmiennoprzecinkowych na całkowite?

Złoty podział odcinka

- Odcinek o długości a+b, podziel w takim stosunku, aby
  - (a+b)/a = a/b = phi
- Zatem:
  - 1+1/phi = phi
  - phi=(1+sqrt(5))/2
- Złoty podział jest niewymierną liczbą.
- Złoty podział jest używany często do proporcji, jako najbardziej atrakcyjny dla ludzkiego oka, np. stosunek wysokości i szerokości obrazu/wykresu.

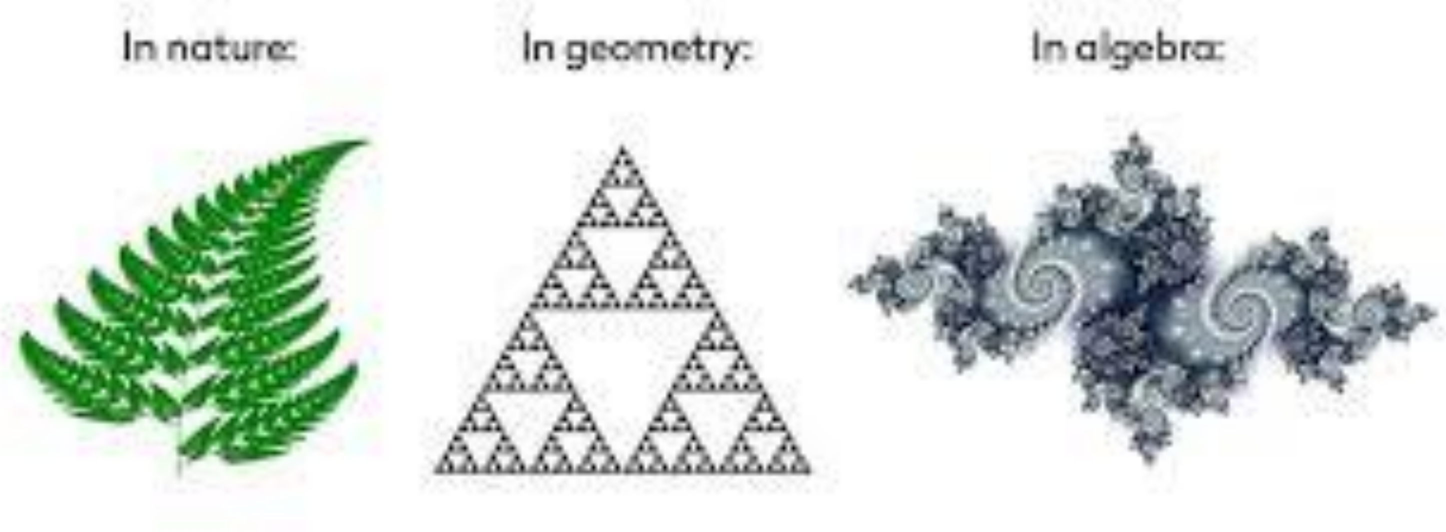
Algorytmy geometryczne i rekurencja

Idea

- Rekurencja dzieli problem na takie same problemy powtarzające (rekurencyjne wywołania) się na innych poziomach (skalach).
- Czy w grafice komputerowej i matematyce występują takie problemy?
- Czy mamy obiekty, które składają się z takich samych ‘części bazowych’ na różnych skalach?

Fraktale

- Fraktale to obiekty samopdobne – podobne do siebie samego na różnych skalach.
- Otrzymuje się je w skutek iteracji/rekurencji.
- Fraktale zostały wymyślone w celu ilustracji skomplikowanych i nieintuicyjnych pojęć matematycznych.
- Mają również regularną strukturę, która przyciąga uwagę człowieka, ale...

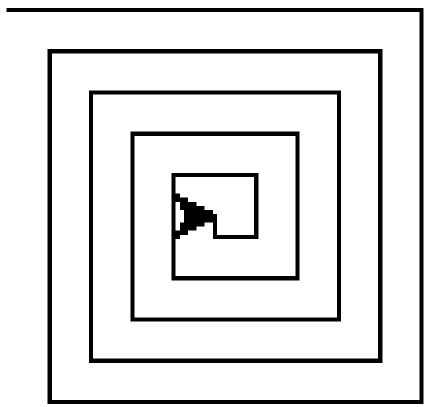


ale takie obiekty zawsze przyciągają uwagę…

Geometria żółwia w Pythonie

- Biblioteka Turtle umożliwia implementację grafiki żółwia.
- Żółw porusza się po płaszczyźnie (kartce) i ciągnie za sobą pisak, który może zostawiać ślad.
- Dokumentacja biblioteki:
  - https://docs.python.org/3.3/library/turtle.html?highlight=turtle

Spirala rekurencyjnie



- import turtle
- my_turtle = turtle.Turtle()
- my_win = turtle.Screen()

- def **draw_spiral**(my_turtle, line_len):
-     if line_len > 0:
-         my_turtle.forward(line_len)
-         my_turtle.right(90)
-         **draw_spiral**(my_turtle, line_len - 5)

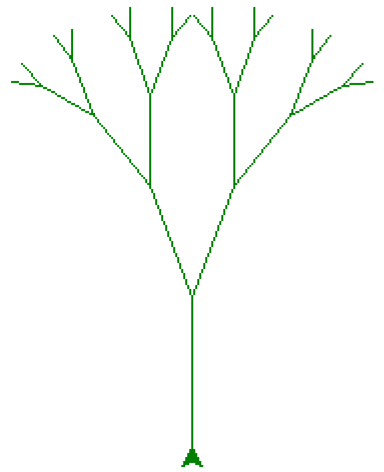
- draw_spiral(my_turtle, 100)
- my_win.exitonclick()

- Proszę zauważyć, że rekurencyjne wywołanie zmniejsza długość linii o 5.
- Proszę poeksperymentować z tym warunkiem.
- Co się stanie gdy będziemy dodawać 5 w każdym wywołaniu rekurencyjnym?

Drzewo rekurencyjnie

```
import turtle
def tree(branch_len, t):
    if branch_len > 5:
        t.forward(branch_len)
        t.right(20)
        tree(branch_len - 15, t)
        t.left(40)
        tree(branch_len - 15, t)
        t.right(20)
        t.backward(branch_len)
def main():
    t = turtle.Turtle()
    my_win = turtle.Screen()
    t.left(90)
    t.up()
    t.backward(100)
    t.down()
    t.color("green")
    tree(75, t)
    my_win.exitonclick()
main()
```

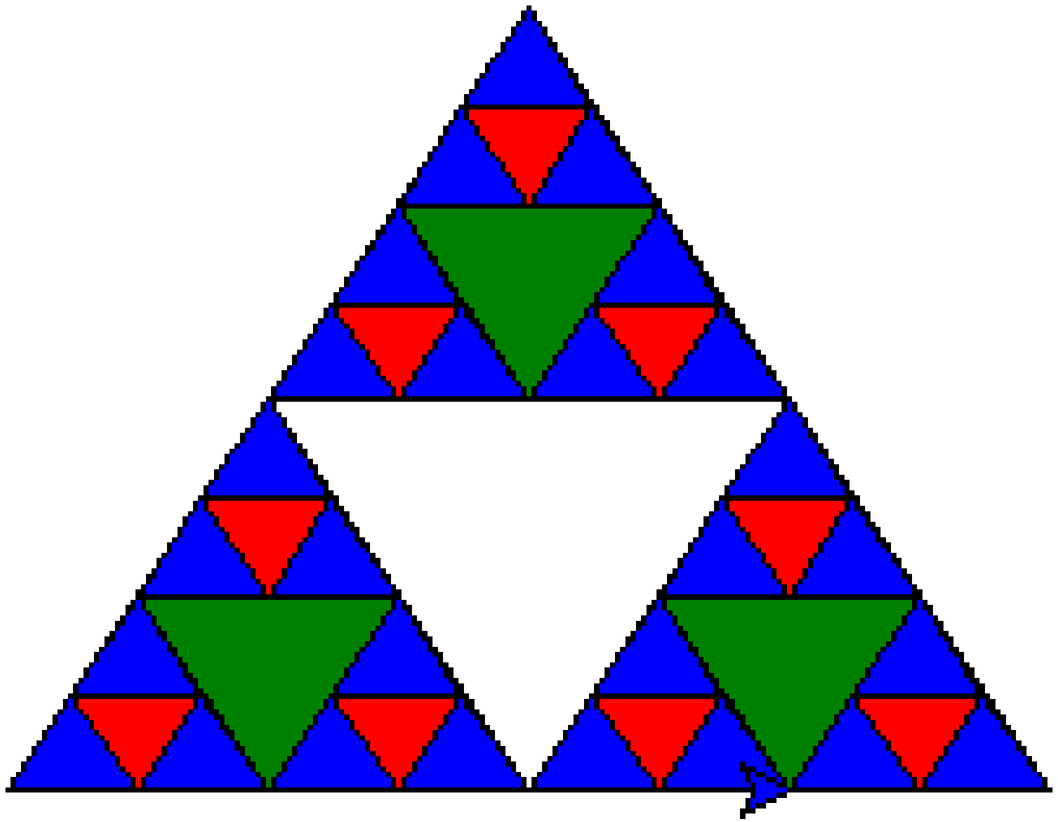
- W każdym wywołaniu rekrurencyjnym długość drzewa zmniejsza się o 15.
- Proszę poeksperymentować z długością, stworzyć niesymetryczne drzewo, zmienić kąt powrotu żółwia itd..
- Takie podejście jest stosowane w generacji drzew w grach na masową skalę.



Trójkąt Sierpińskiego

Wacław Sierpiński (1882-1969) – jeden z najbardziej rozpoznawalnych Polskich matematyków. Pracował w dziedzinie teorii mnogości, topologii, teorii funkcji i teorii liczb. Brał udział w pracach nad łamaniem szyfrów sowieckich podczas inwazji 1920r, dzięki czemu Polskie wojska mogły zwyciężyć z liczebniejszą armią sowiecką.

Trójkąt Sierpińskiego



- Konstrukcja jest rekurencyjna w nieskończoność – granicą jest Trójkąt Sierpińskiego.
  - Rozpocznij od dużego trójkąta.
  - Podziel na 3 trójkąty i usuń środkowy trójkąt
    - W każdym z pozostałych trójkątów usuwaj rekurencyjnie środek.
  - Wykonuj algorytm w nieskończoność, a otrzymasz Trójkąt Sierpińskiego.
- Układ współrzędnych: Przypisz każdemu trójkątowi na danym poziomie rekurencji liczby 1,2,3. Wówczas punkt w Trójkącie Sierpińskiego może być zaadresowany nieskończonym ciągiem liczb ze zbioru 1,2,3, np. 0,1233321…
- Jest to rozwinięcie liczby z przedziału [0;1] w systemie czwórkowym. Zatem Trójkąt Sierpińskiego ma tyle samo punktów co przedział [0;1], bo mamy bijekcję (wzajemnie odwracalne przekształcenie) zbiorów.

Trójkąt Sierpińskiego (Python)

- import turtle

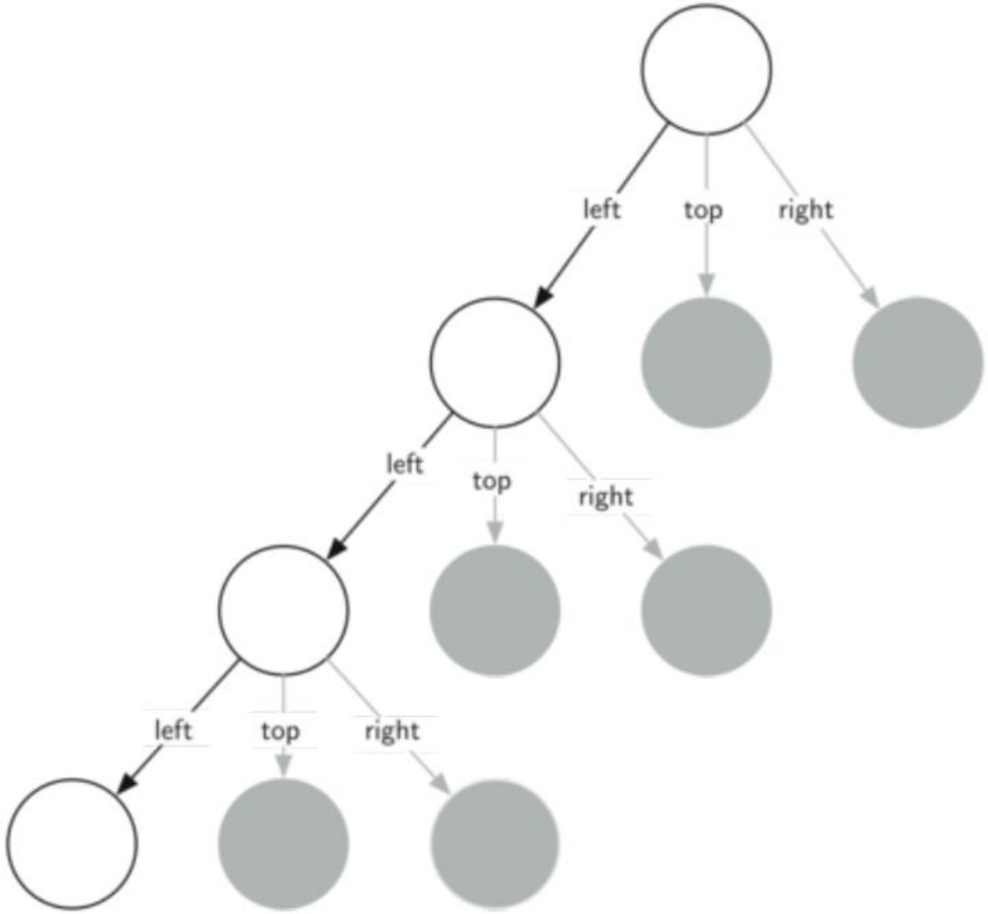
- def draw_triangle(points, color, my_turtle):
-     my_turtle.fillcolor(color)
-     my_turtle.up()
-     my_turtle.goto(points[0][0],points[0][1])
-     my_turtle.down()
-     my_turtle.begin_fill()
-     my_turtle.goto(points[1][0], points[1][1])
-     my_turtle.goto(points[2][0], points[2][1])
-     my_turtle.goto(points[0][0], points[0][1])
-     my_turtle.end_fill()

- def get_mid(p1, p2):
-     return ((p1[0] + p2[0]) / 2, (p1[1] + p2[1]) / 2)

- **def sierpinski**(points, degree, my_turtle):
-     color_map = ['blue', 'red', 'green', 'white', 'yellow','violet', 'orange']
-     draw_triangle(points, color_map[degree], my_turtle)
-     if degree > 0:
-         **sierpinski**([points[0], get_mid(points[0], points[1]), get_mid(points[0], points[2])],degree-1, my_turtle)
-         **sierpinski**([points[1],get_mid(points[0], points[1]), get_mid(points[1], points[2])],degree-1, my_turtle)
-         **sierpinski**([points[2],get_mid(points[2], points[1]), get_mid(points[0], points[2])], degree-1, my_turtle)

- def main():
-     my_turtle = turtle.Turtle()
-     my_win = turtle.Screen()
-     my_points = [[-100, -50], [0, 100], [100, -50]]
-     sierpinski(my_points, 3, my_turtle)
-     my_win.exitonclick()
-
- main()

Trójkąt Sierpińskiego



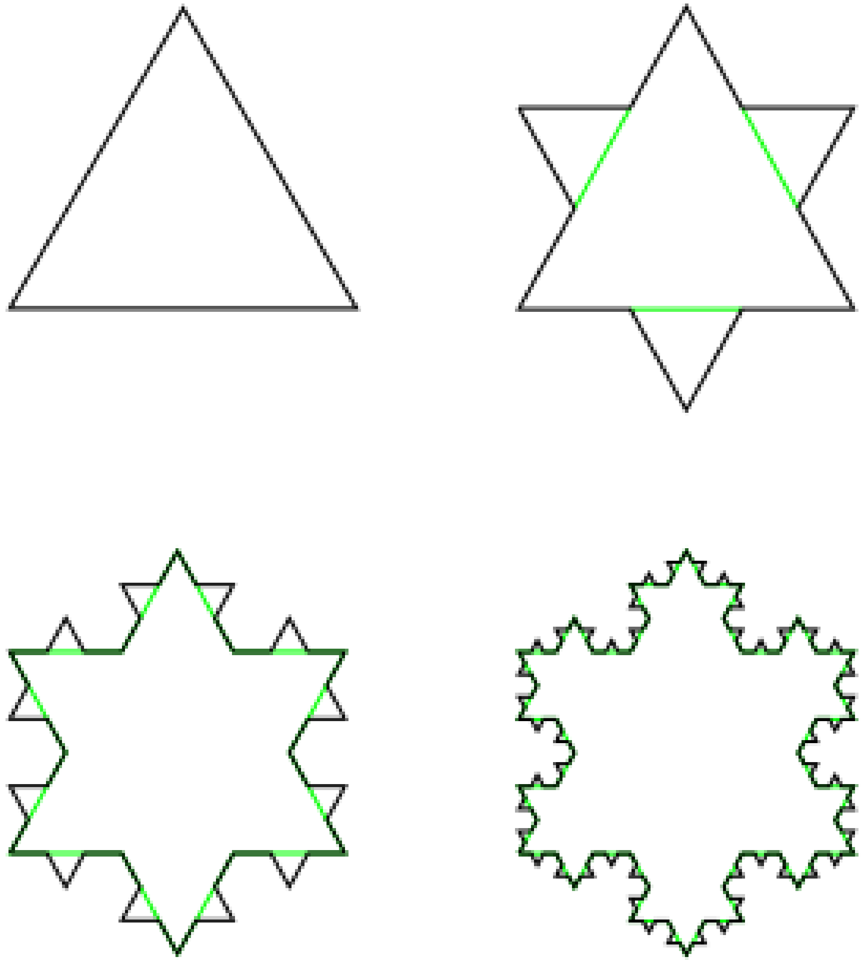
Schemat rekurencyjnego wywołania rysowania trójkątów dla żółwia przypomina drzewo „trójkowe", w którym najpierw rysujemy wzdłuż

- Taki sposób rysowania (przeszukiwania drzewa) nazywa się przeszukiwaniem w głąb (Depth First Search). Więcej o tym dowiemy się później.
- Spróbuj algorytm rozszerzyć na 4 części i kwadrat zamiast trójkąta. Otrzymasz uszczelkę Sierpińskiego.

Krzywa Kocha

Niels Fabian Helge von Koch (1870 – 1924) – Szwedzki matematyk, który podał jeden z pierwszych przykładów fraktala – krzywą Kocha lub płatek Kocha.

Krzywa Kocha



- Krzywa Kocha jest obiektem granicznym otrzymanym w iteracji.
- Iteracja polega na zastępowaniu środków trójkąta (odcinka dla dla krzywej).
- Krzywa Kocha jest przykładem obiektu, którego wymiar(fraktalny) jest równy d=ln(4)/ln(3) ≈ 1.26186, więc ‘jest czymś pomiędzy’ krzywą (d=1), a powierzchnią (d=2).

Krzywa Kocha – podejście długie

- import turtle

- **def koch**(t, order, size):
-     if order == 0:   # The base case is just a straight line
-         t.forward(size)
-     else:
-         **koch**(t, order-1, size/3)   # Go 1/3 of the way
-         t.left(60)
-         **koch**(t, order-1, size/3)
-         t.right(120)
-         **koch**(t, order-1, size/3)
-         t.left(60)
-         **koch**(t, order-1, size/3)

- t = turtle.Turtle()   # create the turtle
- wn = turtle.Screen()
- koch(t, 2, 200)
- wn.exitonclick()

Zauważ, że w każdym wywołaniu długość początkowa odcinka jest zmniejszana o 1/3.

Czy możemy to przepisać efektywniej? Tak.

Krzywa Kocha – podejście efektywniejsze

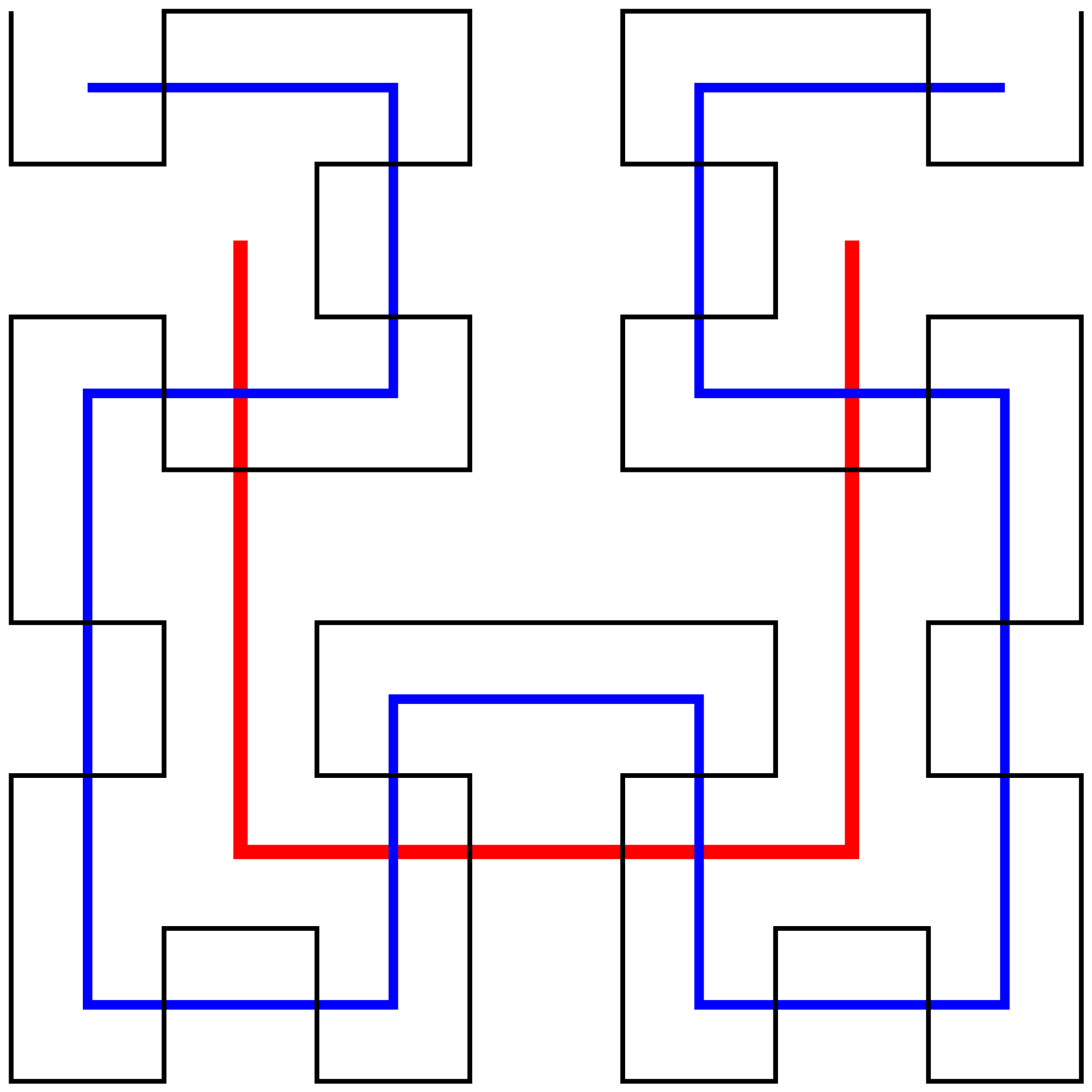
```
import turtle
def koch(t, order, size):
   if order == 0:
        t.forward(size)
    else:
       for angle in [60, -120, 60, 0]:
          koch(t, order-1, size/3)
         t.left(angle)
```

```
t = turtle.Turtle()           # create the turtle
wn = turtle.Screen()
koch(t, 3, 200)
wn.exitonclick()
```

Krzywa Hilberta

- David Hilbert (1862-1943) jeden z najbardziej wszechstronnych matematyków przełomu wieków 19-20.
- Podał matematyczną wersję ogólnej teorię względności (po rozmowie z Einsteinem) wcześniej niż Einstein.
- Na początku 20 wieku sformułował listę 23 (24) problemów matematycznych, które kształtowały losy matematyki w 20 wieku. Obecnie większa część z nich jest rozwiązana:
  - https://en.wikipedia.org/wiki/Hilbert%27s_problems

Krzywa Hilberta wypełniająca przestrzeń

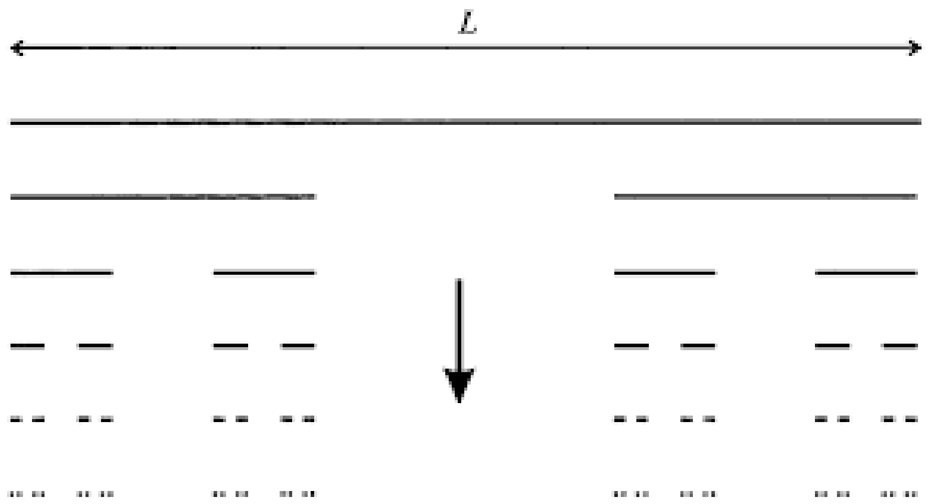


- Krzywa Hilberta jest krzywą otrzymaną w granicy nieskończonej liczby kroków.
  - Krok 1- czerwony
  - Krok 2 – niebieski
  - Krok 3 – szary
- Krzywa ta wypełnia cały kwadrat pokazując, że w dwóch wymiarach kwadratu jest TYLE SAMO punktów co na prostej (która jest wyprostowaną krzywą Hilberta).

Krzywa Hilberta

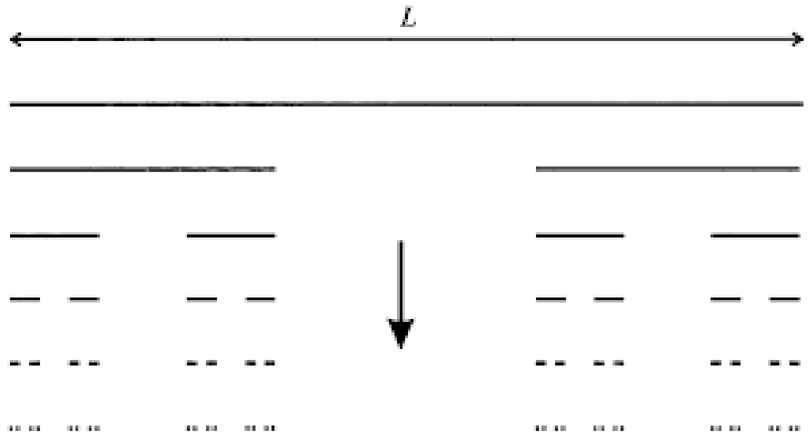
```
import turtle

def hilbert2(step, rule, angle, depth, t):
  if depth > 0:
    a = lambda: hilbert2(step, "a", angle, depth - 1, t)
    b = lambda: hilbert2(step, "b", angle, depth - 1, t)
    left = lambda: t.left(angle)
    right = lambda: t.right(angle)
    forward = lambda: t.forward(step)
    if rule == "a":
      left(); b(); forward(); right(); a(); forward(); a(); right(); forward(); b(); left();
    if rule == "b":
      right(); a(); forward(); left(); b(); forward(); b(); left(); forward(); a(); right();

myTurtle = turtle.Turtle()
myTurtle.speed(0)
hilbert2(5, "a", 90, 5, myTurtle)
```

Zbiór Cantora



George Cantor (1845-1918) – matematyk zajmujący się teorią mnogości i podstawy teorii przeliczalności zbiorów.

Zbiór Cantora



- Zbiór Cantora jest zbiorem granicznym polegającym na usuwaniu środkowej (1/3) odcinka z odcinków na każdym etapie.
  - Krok 1 – usuń środkowy odcinek długości 1/3
  - Krok 2- usuń z każdego z 2 odcinków środkowe odcinki równe 1/3 ich długości
  - …
- Zbiór Cantora jest zbiorem składającym się z punktów po nieskończonej liczbie takich usunięć – proszę udowodnić, że coś jeszcze zostaje przy użyciu zwartości odcinka [0,1].
- Ile punktów zostaje? Odp. Tyle ile jest na odcinku [0;1], czyli tak jakbyśmy nic nie usunęli. Dowód: proszę na każdym poziomie nadać lewemu odcinkowi etykietę 0, a prawemu 1. Wówczas idąc od góry punkt zbioru Cantora (w nieskończoności na dole) odpowiada ciągowi nieskończonemu liczb ze zbioru {0,1}, np. 0,101000.... Ponieważ otrzymamy dowolny ciąg, a każda liczba z przedziału [0;1] ma takie rozwinięcie, więc mamy tyle samo liczb, co elementów zbioru Cantora.
- Zbiór Cantora pokazuje, że nieskończoność nieintuicyjna – usuwamy nieskończenie wiele ‘dużych’ odcinków, a otrzymujemy zbiór z liczbą elementów, których jest ‘tyle samo’ (w sensie bijektywnej odpowiedniości punktów obydwu zbiorów) co przed procedurą usuwania. Proszę to przemyśleć, tylko nie za długo – proszę przeczytać życiorys Cantora.

Zbiór Cantora (Python)

```
WIDTH = 81
HEIGHT = 5
lines=[]
def cantor(start, len, index):
    seg = len / 3
    if seg == 0:
        return None
    for it in xrange(HEIGHT-index):
        i = index + it
        for jt in xrange(seg):
            j = start + seg + jt
            pos = i * WIDTH + j
            lines[pos] = ' '
    cantor(start,           seg, index + 1)
    cantor(start + seg * 2, seg, index + 1)
    return None
```

```
lines = ['*'] * (WIDTH*HEIGHT)
cantor(0, WIDTH, 1)

for i in xrange(HEIGHT):
    beg = WIDTH * i
    print ''.join(lines[beg : beg+WIDTH])
```

Ogólne spojrzenie na obiekty tworzone w procesie rekurencji
L-systemy

L-systemy

- L-systemy (od nazwiska Lindenmayera – biologa) są to systemy które powstają w wyniku iteracji pewnej sekwencji (która może coś oznaczać). Systemy takie naśladują wzrost roślin/glonów.
- Definicja:
  - L-system to zbiór (V,w,P):
    - V – alfabet (zbiór symboli, które mogą mieć dla Nas znaczenie); Składa się ze zmiennych (które będą zamieniane i stałych symboli, które są niezmienne;
    - W – symbol startowy od którego rozpoczynamy iterację;
    - P – reguły produkcyjne, wykorzystywane w iteracji;

Przykład – Wzrost alg

- V={A,B}, W=A, P={ (A->AB), (B->A)}
- A – komórka gotowa do rozmnażania przez podział A->AB; B- komórka niedojrzała, która dojrzewa B->A;
- Sekwencja wzrostu (nawias oznacza ‘podział’ A):
  - A
  - (A)B
  - (AB)A = ABA = (A)B(A)
  - (AB)A(AB) =ABAAB= (A)B(A)(A)B
  - (AB)A(AB)(AB)A=ABAABABA = ...
  - ...

Przykład - Zbiór Cantora

- V={A,B}, W=A, P={(A->ABA), (B->BBB)}
- A – narysuj odcinek; B – przesuń się o odcinek nic nie rysując;
- Sekwencja wzrostu (proszę narysować):
  - A
  - ABA= (A)B(A)
  - (ABA)BBB(ABA) = ABA BBB ABA = (A)B(A)BBB(A)B(A)
  - (ABA)BBB(ABA)BBBBBBBBB(ABA)BBB(ABA)
  - …

Krzywa Kocha

- V={F,+,-}, W=F, P={(F → F+F−F−F+F)}
- F- ruch do przodu o odcinek; + obrót o 90deg w lewo; - obrót o 90 deg w prawo;
- Sekwencja wzrostu (proszę narysować):
  - F
  - F+F−F−F+F
  - ….

Trójkąt Sierpińskiego

- V={F,G,+,-}, W=F-G-G, P={ (F → F−G+F+G−F), (G → GG)}
- F, G – narysuj odcinek do przodu; + stała określająca obrót o 120deg w lewo; - stała określająca obrót o 120deg w prawo; To symbole interpretowane przez żółwia.
- Sekwencja wzrostu(proszę narysować):
  - F-G-G (trójkąt)
  - (F−G+F+G−F)-(GG)-(GG) (trójkąt z wyciętym środkowym trójkątem)
  - ...

- Implementacja parsera L-Systemu w Pythonie dla grafiki żółwia
- https://runestone.academy/runestone/books/published/thinkcspy/Strings/TurtlesandStringsandLSystems.html

Elementy składowe

- Tworzenie L-systemu:
  - Reguły produkcji:
    - applyRules(lhch);
  - Produkcja kolejnej iteracji:
    - processString(oldStr)
  - Tworzenie systemu do wybranego poziomu z elementu startowego:
    - createLSystem(numIters,axiom)
- Parsowanie L-systemu (przez żółwia):
  - drawLsystem(aTurtle,instructions,angle,distance)

Wzrost alg

- def **applyRules(lhch):**
-     rhstr = ""
-     if lhch == 'A':
-         rhstr = 'B'   # Rule 1
-     elif lhch == 'B':
-         rhstr = 'AB' # Rule 2
-     else:
-         rhstr = lhch   # no rules apply so keep the character
-     return rhstr
- def **processString(oldStr):**
-     newstr = ""
-     for ch in oldStr:
-         newstr = newstr + applyRules(ch)
-     return newstr

- def **createLSystem(numIters,axiom)**:
-     startString = axiom
-     endString = ""
-     for i in range(numIters):
-         endString = processString(startString)
-         startString = endString
-     return endString
- #10 iteracji wzrostu alg startując z A
- print(createLSystem(10, "A"))

Krzywa Kocha - generacja

- def **applyRules(ch):**
-     newstr = ""
-     if ch == 'F':
-         newstr = 'F-F++F-F'   # Rule 1
-     else:
-         newstr = ch   # no rules apply so keep the character

-     return newstr

- def **processString(oldStr):**
-     newstr = ""
-     for ch in oldStr:
-         newstr = newstr + applyRules(ch)

-     return newstr

- def **createLSystem(numIters,axiom):**
-     startString = axiom
-     endString = ""
-     for i in range(numIters):
-         endString = processString(startString)
-         startString = endString

-     return endString

- print(createLSystem(3, "F"))

Krzywa Kocha – parsowanie/rysowanie

```
import turtle

def createLSystem(numIters,axiom):
    startString = axiom
    endString = ""
    for i in range(numIters):
        endString = processString(startString)
        startString = endString

    return endString

def processString(oldStr):
    newstr = ""
    for ch in oldStr:
        newstr = newstr + applyRules(ch)

    return newstr

def applyRules(ch):
    newstr = ""
    if ch == 'F':
        newstr = 'F-F++F-F'   # Rule 1
    else:
        newstr = ch    # no rules apply so keep the character

    return newstr
```

```
def drawLsystem(aTurtle, instructions, angle, distance):
    for cmd in instructions:
        if cmd == 'F':
            aTurtle.forward(distance)
        elif cmd == 'B':
            aTurtle.backward(distance)
        elif cmd == '+':
            aTurtle.right(angle)
        elif cmd == '-':
            aTurtle.left(angle)


def main():
    inst = createLSystem(4, "F")   # create the string
    print(inst)
    t = turtle.Turtle()           # create the turtle
    wn = turtle.Screen()


    t.up()
    t.back(200)
    t.down()
    t.speed(9)
    drawLsystem(t, inst, 60, 5)   # draw the picture
                                  # angle 60, segment length 5
    wn.exitonclick()
```

Zadanie

- Proszę przystosować powyższy program do rysowania trójkąta Sierpińskiego.
- Należy:
  - applyRules(ch) – zmienić reguły produkcji na trójkąt Sierpińskiego;
  - drawLsystem(aTurtle, instructions, angle, distance) - zmienić reguły ruchu żółwia aby reagował na symbole {F,G,+,-}.
- Proszę zmodyfikować program, aby parsował inne L-systemy:
  - https://en.wikipedia.org/wiki/L-system
- Może uda się Państwu stworzyć L-system o ciekawych własnościach. Proszę pomyśleć.

L-systemy rekurencyjne

- Proszę zapoznać się z funkcją **map** w Pythonie, np.
  - map(sqrt,[1,4,9])
- Funkcja map pobiera funkcję jednoargumentową jako pierwszy argument oraz tablicę do której tę funkcję aplikuje. Jest to zapis pętli for w paradygmacie funkcyjnym, gdzie iteracja nie występuje!
- Funkcje pobierające funkcje jako argumenty nazywamy funkcjami wyższego rzędu (higher order functions).
- Proszę przepisać funkcje
  - **createLSystem(numIters,axiom):**
  - **processString(oldStr)**
  - przy użyciu map zamiast pętli for. To powyższy generator-parser zamieni na napisany w paradygmacie funkcyjnym.

Iterowane Systemy Funkcji

(IFS – Iterated Function Systems)

IFS

- IFS jest sposobem generacji struktur samopodobnych/fraktalnych przy użyciu iteracji.
- Kroki:
  - Wybierz zbiór początkowy (z płaszczyzny, przestrzeni) – A.
  - Aplikuj odwzorowanie afiniczne w nieskończoność:
    - S=lim_{n->inf} F^n (A)
  - Granica S jest zbiorem fraktalnym (należy udowodnić, że granica nie jest zbiorem pustym).

Co to jest odwzorowanie afiniczne?

$$f(x,y) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

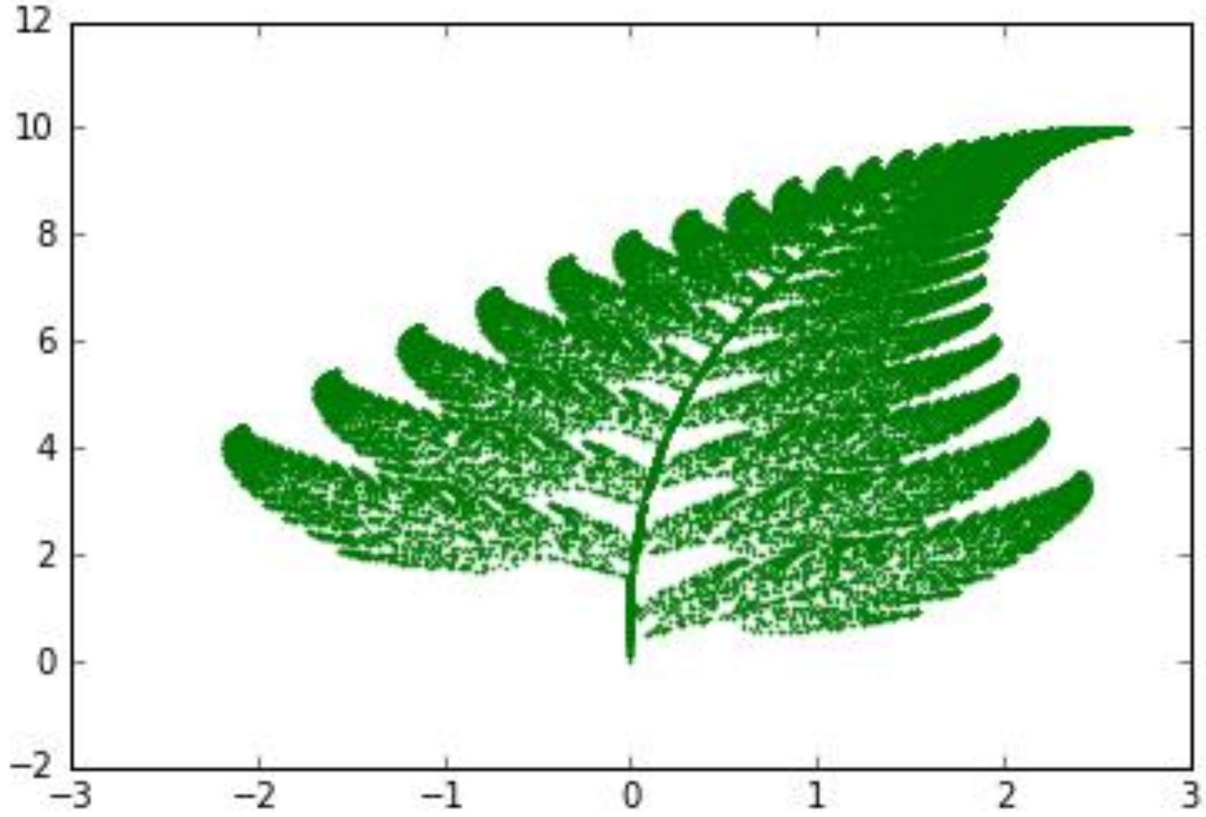
Paprotka Barnsleya

$$\mathrm{f}_1 :$$
$$x_{n+1} = 0$$
$$y_{n+1} = 0.16 y_n$$

$$f_2 :$$
$$x_{n+1} = 0.85 x_n + 0.04 y_n$$
$$y_{n+1} = -0.04 x_n + 0.85 y_n + 1.6$$

$$f_3 :$$
$$x_{n+1} = 0.2 x_n - 0.26 y_n$$
$$y_{n+1} = 0.23 x_n + 0.22 y_n + 1.6$$

$$f_4 :$$
$$x_{n+1} = -0.15 x_n + 0.28 y_n$$
$$y_{n+1} = 0.26 x_n + 0.24 y_n + 0.44$$



Paprotka Barnsleya

- # importing necessary modules
- import matplotlib.pyplot as plt
- from random import randint

- # initializing the list
- x = []
- y = []

- # setting first element to 0
- x.append(0)
- y.append(0)

- current = 0

- for i in range(1, 50000):

- # generates a random integer between 1 and 100
- z = randint(1, 100)

- # the x and y coordinates of the equations
- # are appended in the lists respectively.

- # for the probability 0.01
- if z == 1:
- x.append(0)
- y.append(0.16*(y[current]))

- # for the probability 0.85
- if z>= 2 and z<= 86:
- x.append(0.85*(x[current]) + 0.04*(y[current]))
- y.append(-0.04*(x[current]) + 0.85*(y[current])+1.6)

- # for the probability 0.07

- if z>= 87 and z<= 93:
- x.append(0.2*(x[current]) - 0.26*(y[current]))
- y.append(0.23*(x[current]) + 0.22*(y[current])+1.6)

- # for the probability 0.07
- if z>= 94 and z<= 100:
- x.append(-0.15*(x[current]) + 0.28*(y[current]))
- y.append(0.26*(x[current]) + 0.24*(y[current])+0.44)

- current = current + 1

- plt.scatter(x, y, s = 0.2, edgecolor ='green')

- plt.show()

Zadanie

Proszę spróbować zaimplementować algorytm tworzący paprotkę Bransleya w C/C++:

https://rosettacode.org/wiki/Barnsley_fern

Zbiory Julii

i

zbiór Mandelbrota

Zbiory Julii

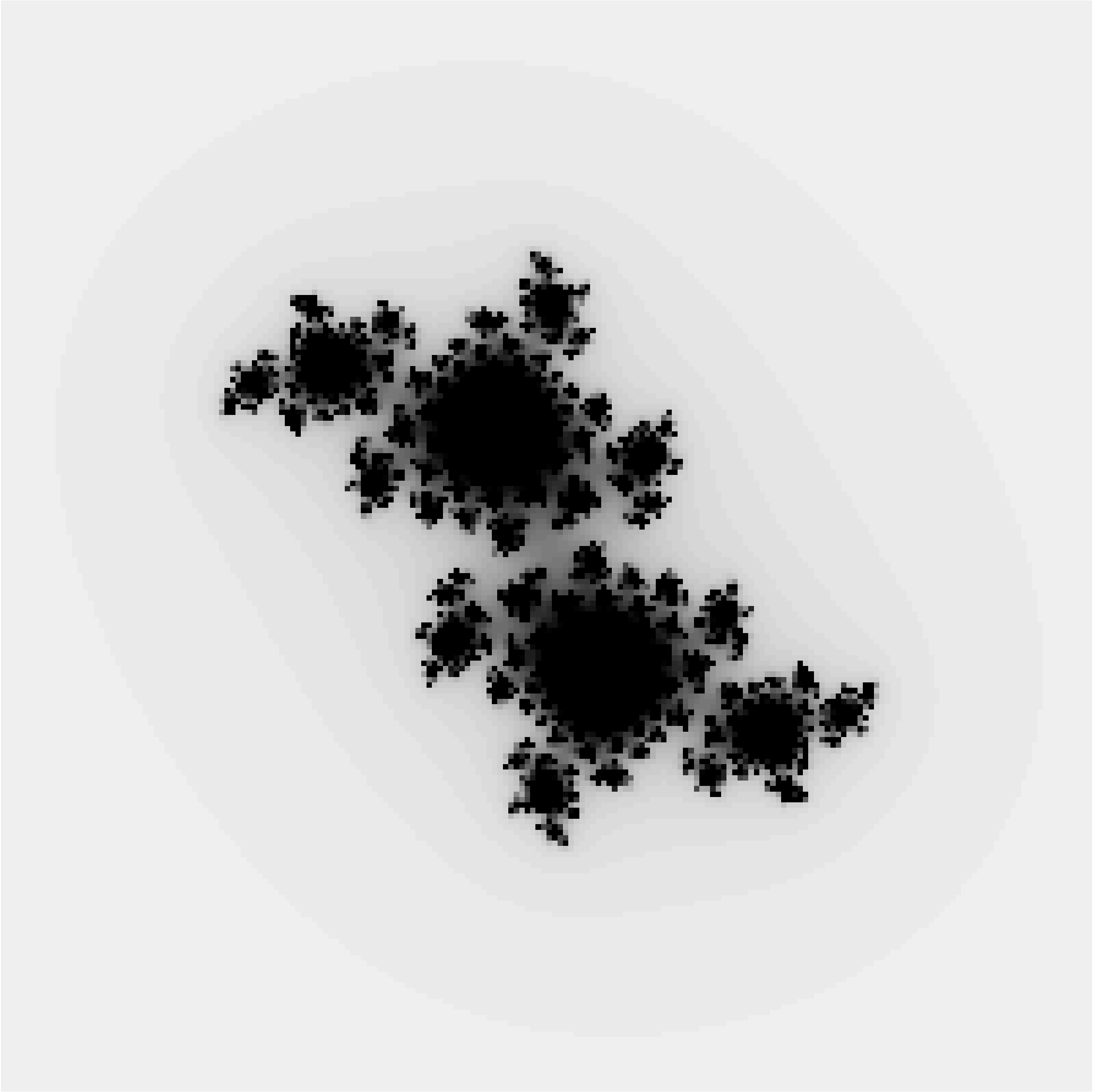
- Zbiór punktów nie rozbiegających się do nieskończoności (ograniczonych) pod wpływem iteracji z → z^2 +c płaszczyzny zespolonej. C jest stałą zespoloną.
- Algorytm:
  - Dla każdego punktu w zakresie |Re(z)|<2, |Im(z)|<2 wykonaj:
    - n_max iteracji z → z^2 +c lub do momentu, gdy z wyjdzie z kwadratu o boku 2x2 na płaszczyźnie zespolonej;
    - Oznacz piksel jasnym kolorem liczbę iteracji po której punkt wyjdzie z kwadratu;
    - Oznacz piksel czarnym kolorem jeżeli nie wyszedł z kwadratu po n_max iteracji;
- Zbiór Juli: c=0.65i

Zbiór Julii (Python)

```python
import numpy

# Specify image width and height
w, h = 200, 200

# Specify real and imaginary range of image
re_min, re_max = -2.0, 2.0
im_min, im_max = -2.0, 2.0

# Pick a value for c
c = complex(0.0,0.65)

# Generate evenly spaced values over real and imaginary ranges
real_range = numpy.arange(re_min, re_max, (re_max - re_min) / w)
imag_range = numpy.arange(im_max, im_min, (im_min - im_max) / h)

# Open output file and write PGM header info
fout = open('julia.pgm', 'w')
fout.write('P2\n# Julia Set image\n' + str(w) + ' ' + str(h) + '\n255\n')
```

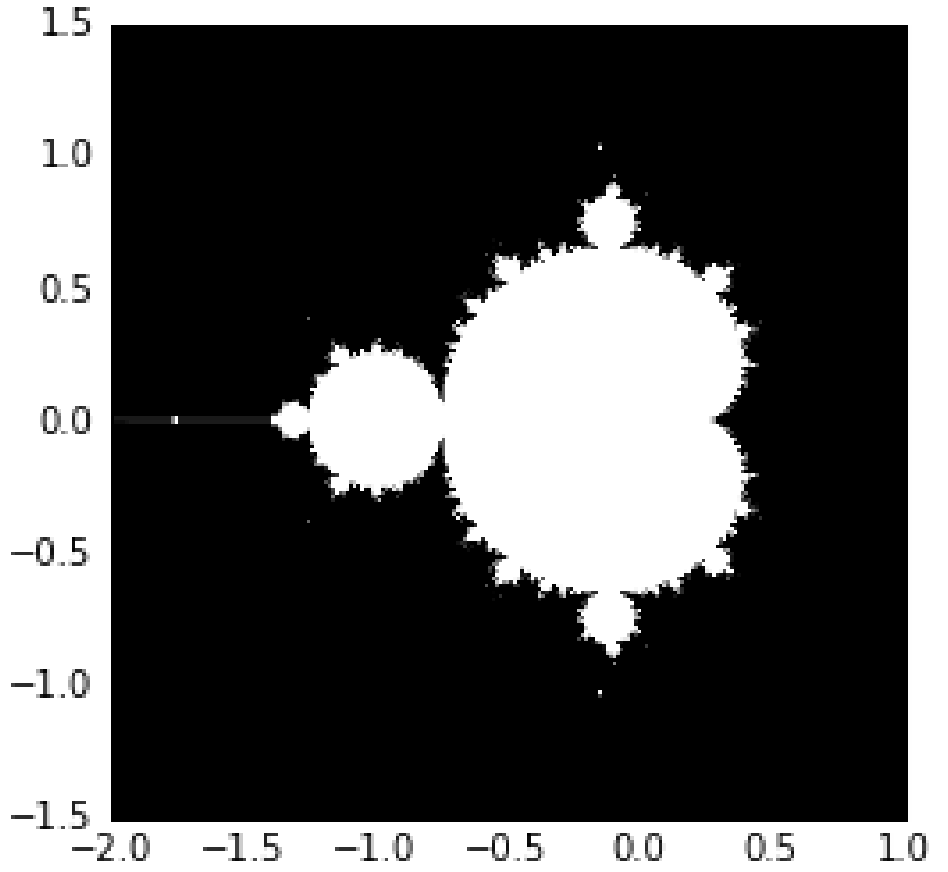
```python
# Generate pixel values and write to file
for im in imag_range:
    for re in real_range:
        z = complex(re, im)
        n = 250
        while abs(z) < 10 and n >= 5:
            z = z*z + c
            n -= 5
        # Write pixel to file
        fout.write(str(n) + ' ')
    # End of row
    fout.write('\n')

# Close file
fout.close()
```

Zbiór Julii



Proszę sprawdzić inne wartości c.

Zbiór Mandelbrota



- Algorytm:
  - Dla obszaru 2x2 weź punkt c
  - Iteruj n_max razy lub do czasu gdy punt z wyjdzie z kwadratu 2x2, z=0, z<- z^2+c
  - Jeżeli punkt nie wyjdzie poza obszar to należy do zbioru Mandelbrota - oznacz odpowiednio.
- Punkt c należy do zbioru Mandelbrota jeżeli dla tego punktu zbiór Julii jest spójny (połączony).

Zbiór Mandelbrota

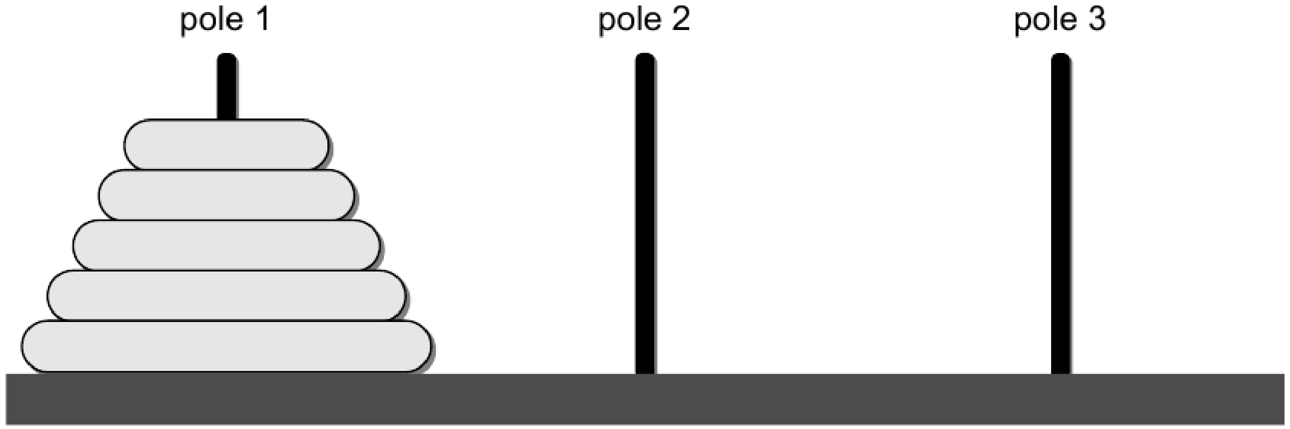
- import numpy as np
- import matplotlib.pyplot as plt
- from numpy import newaxis

- **def compute_mandelbrot(N_max, some_threshold, nx, ny):**
-     # A grid of c-values
-     x = np.linspace(-2, 1, nx)
-     y = np.linspace(-1.5, 1.5, ny)
-     c = x[:,newaxis] + 1j*y[newaxis,:]
-     # Mandelbrot iteration
-     z = c
-     for j in range(N_max):
-         z = z**2 + c
-     mandelbrot_set = (abs(z) < some_threshold)
-     return mandelbrot_set

- mandelbrot_set = **compute_mandelbrot**(50, 50., 601, 401)
- plt.imshow(mandelbrot_set.T, extent=[-2, 1, -1.5, 1.5])
- plt.gray()
- plt.show()

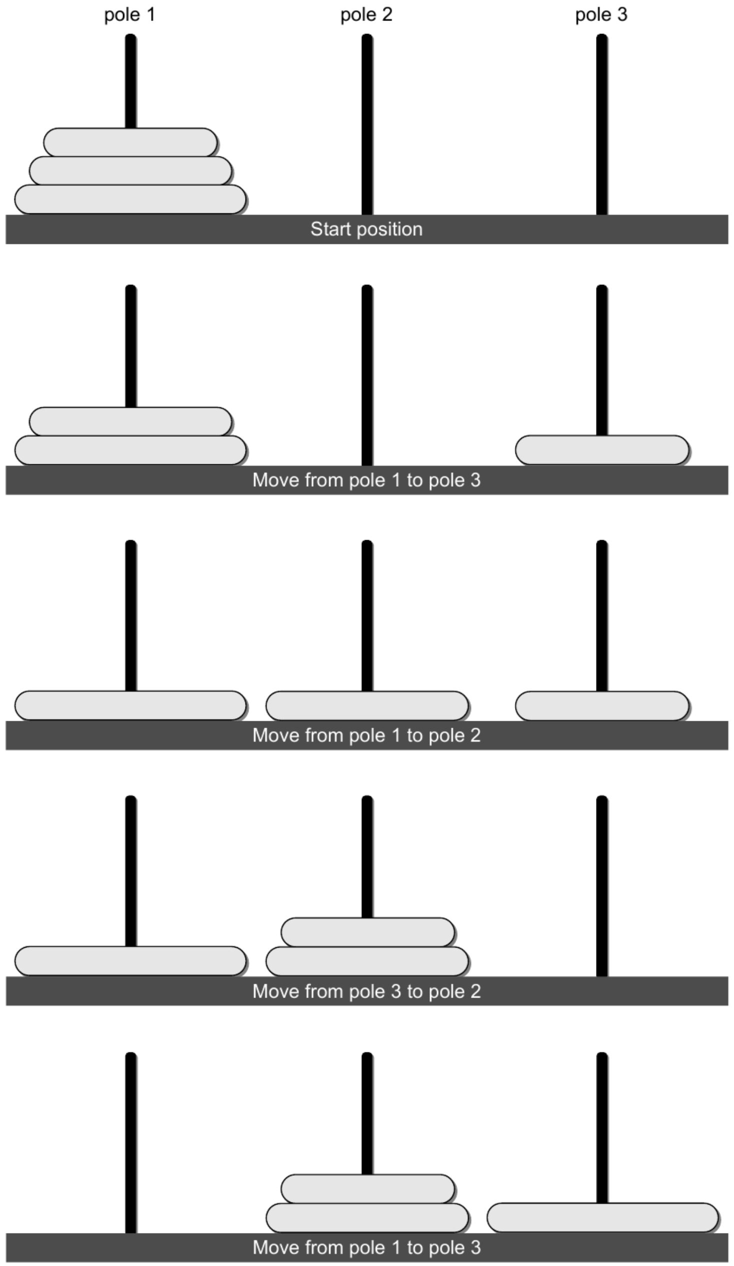
Wieże Hanoi

Wieże Hanoi



- Należy przenieść krążki z jednego drążka na drugi **jeden po drugim.**
- **Nie można** umieścić większego krążka na mniejszym.

Przykładowe cztery pierwsze kroki



Algorytm

- Algorytm:
  - Przesuń (rekurencyjnie) zestaw o wysokości n-1 na pośrednie miejsce używając docelowego miejsca.
  - Przesuń największy (ostatni) dysk na docelowe miejsce.
  - Przesuń (rekurencyjnie) zestaw o wysokości n-1 ze środkowego miejsca na docelowe używając początkowego (już pustego) miejsca.
- Proszę sprawdzić na kartce lub modelu, czy algorytm działa.

Implementacja (Python)

- def **move_tower**(height, from_pole, to_pole, with_pole):
-     if height >= 1:
-         **move_tower**(height - 1, from_pole, with_pole, to_pole)
-         move_disk(from_pole, to_pole)
-         **move_tower**(height - 1, with_pole, to_pole, from_pole)

- def move_disk(fp,tp):
-     print("moving disk from",fp,"to",tp)

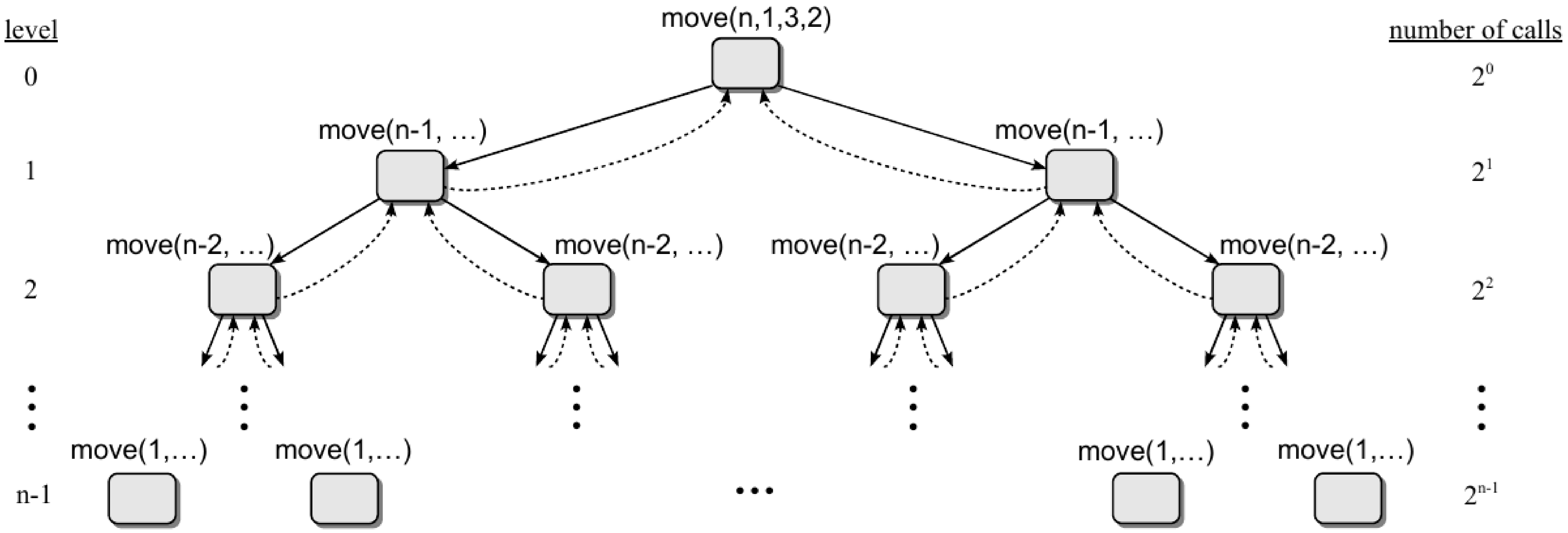
- move_tower(3, "A", "B", "C")

Sekwencja wywołań dla 3 krążków

Proszę sprawdzić czy to się zgadza:

- ('moving disk from', 'A', 'to', 'B')
- ('moving disk from', 'A', 'to', 'C')
- ('moving disk from', 'B', 'to', 'C')
- ('moving disk from', 'A', 'to', 'B')
- ('moving disk from', 'C', 'to', 'A')
- ('moving disk from', 'C', 'to', 'B')
- ('moving disk from', 'A', 'to', 'B')

Sekwencja wywołań dla n



Dlaczego to ważne?

- Legenda:
  - W wielkiej świątyni Benares w Hanoi, pod kopułą, która zaznacza środek świata, znajduje się płytka z brązu, na której umocowane są trzy diamentowe igły, wysokie na łokieć i cienkie jak talia osy. Na jednej z tych igieł, w momencie stworzenia świata, Bóg umieścił 64 krążki ze szczerego złota. Największy z nich leży na płytce z brązu, a pozostałe jeden na drugim, idąc malejąco od największego do najmniejszego. Bez przerwy we dnie i w nocy kapłani przekładają krążki z jednej diamentowej igły na drugą, przestrzegając niewzruszonych praw Brahma. Prawa te chcą, aby kapłan na służbie brał tylko jeden krążek na raz i aby umieszczał go na jednej z igieł w ten sposób, by nigdy nie znalazł się pod nim krążek mniejszy. Wówczas, gdy 64 krążki zostaną przełożone z igły, na której umieścił je Bóg w momencie stworzenia świata, na jedną z dwóch pozostałych igieł, wieża, świątynia, bramini rozsypią się w proch i w jednym oka mgnieniu nastąpi koniec świata.
- Ile to potrwa?
  - 2^64 -1 = 18446744073709551615 jednostek czasu, gdzie jednostka czasu to czas na przesunięcie jednego krążka, np. 1 sekunda.
- Dlaczego?
  - Czasowa złożoność obliczeniowa algorytmu. Obliczymy to już wkrótce.

Literatura do analizy

- Rozdział 2 - Algorytmy. Struktury danych i techniki programowania.
- Rozdział 10 - Data Structures and Algorithms using Python.

Koniec