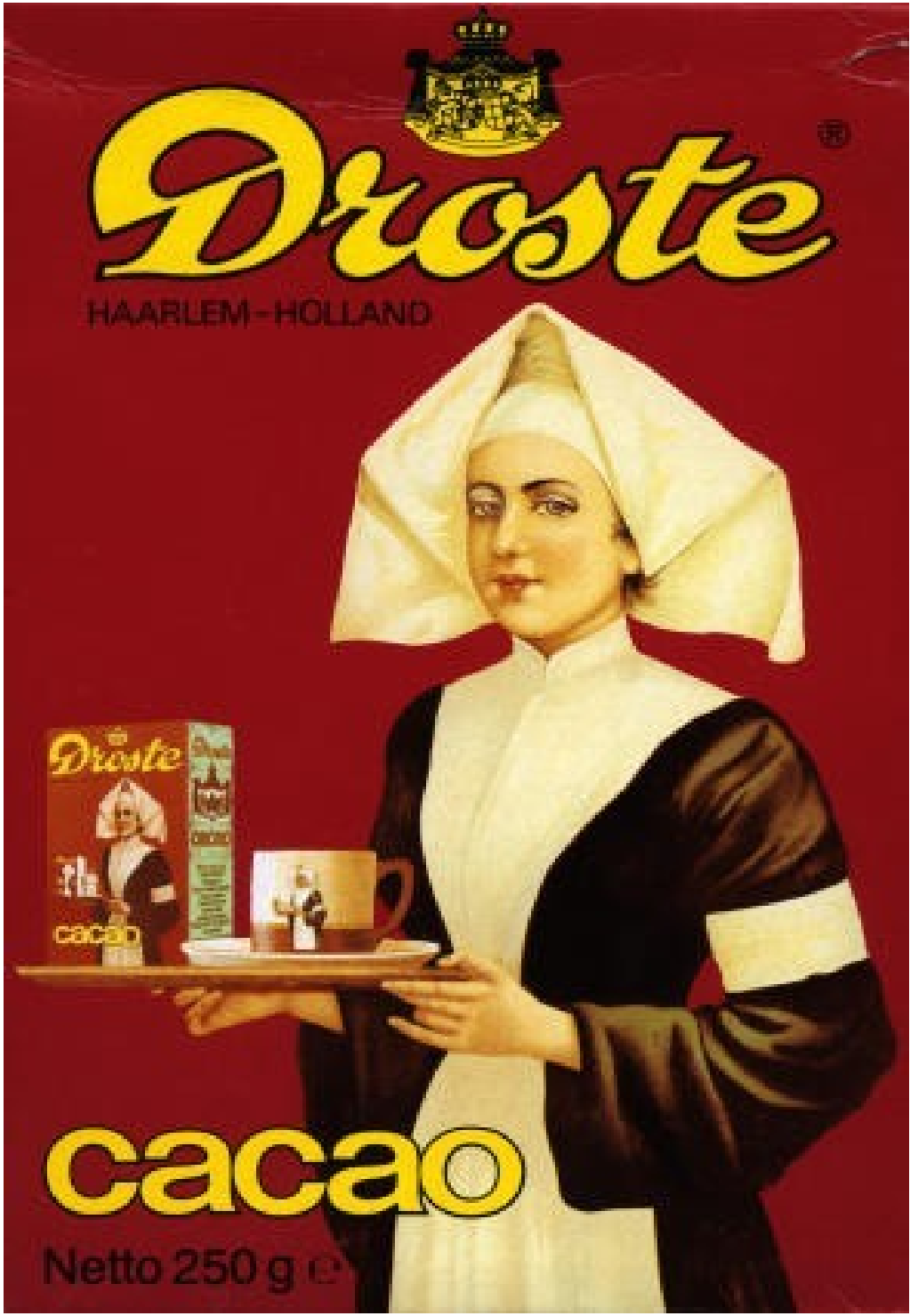
Wstęp do Informatyki dla bioinformatyków

Wykład 3
Rekurencja

Bartek Wilczyński
19 X 2015

Rekurencja:

- Samopowtarzalność,
- Rozbicie problemu na “mniejsze” instancje tego samego problemu
- Zdolność funkcji do wywoływania samej siebie



Rekurencja w matematyce:

$$1! = 1$$
$$N! = N \cdot (N-1)!$$

```
def silnia(n):
    w=1
    for i in range(1,n+1):
        w*=i
    return w
```

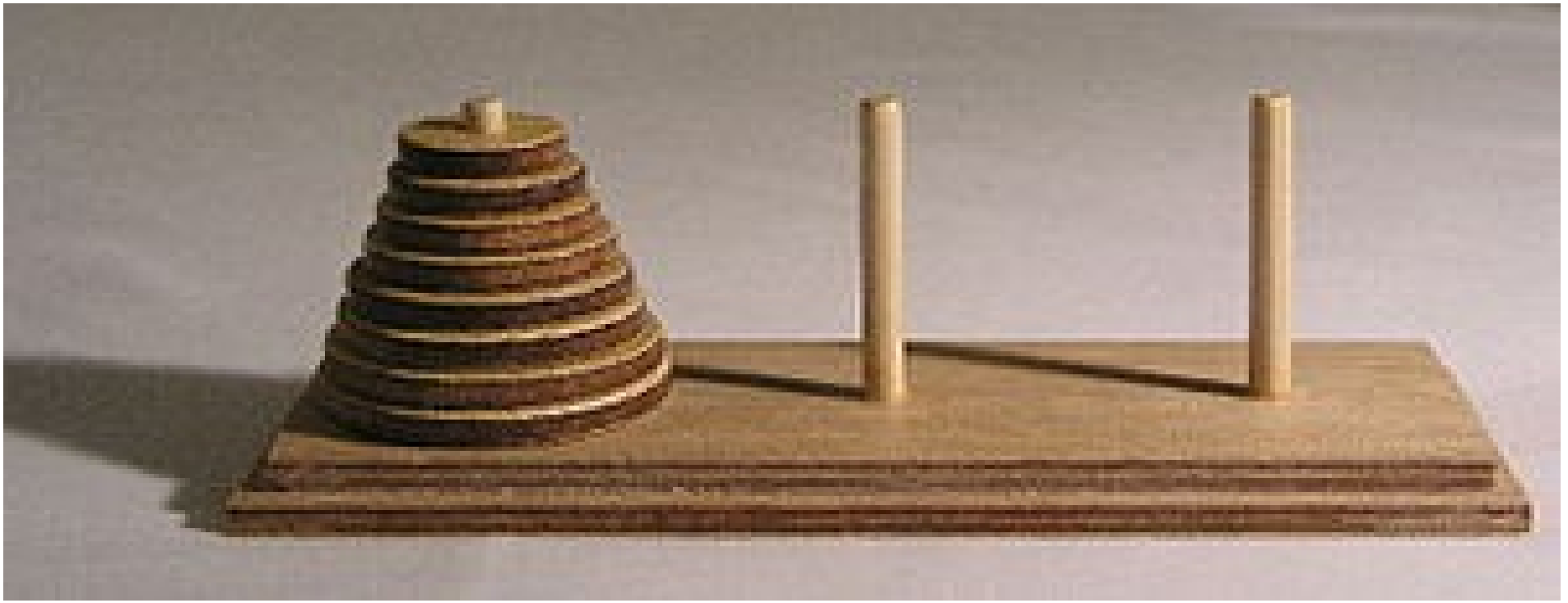
```
def silnia_r(n):
    if n==1:
        return 1
    else:
        return n*silnia_r(n-1)
```

Przykład nieco mniej oczywisty

```
def nwd(m,n):
    """ Największy wspólny dzielnik m i n"""
    if m==n:
        return m
    elif m>n:
        return nwd(m-n,n)
    else: # n>m
        return nwd(n-m,m)
```

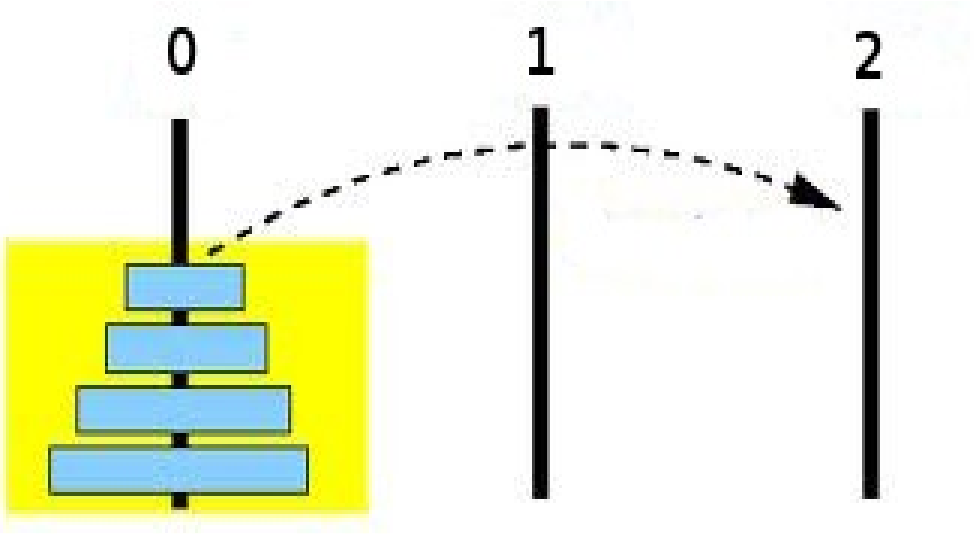
Rekursja to nie tylko liczby

- **Wieże Hanoi:** jak przenieść wszystkie krążki na 3 drążek przenosząc 1 krążek na raz bez naruszania porządku krążków (mniejszy zawsze na większym)



Jak opisać problem Wież Hanoi?

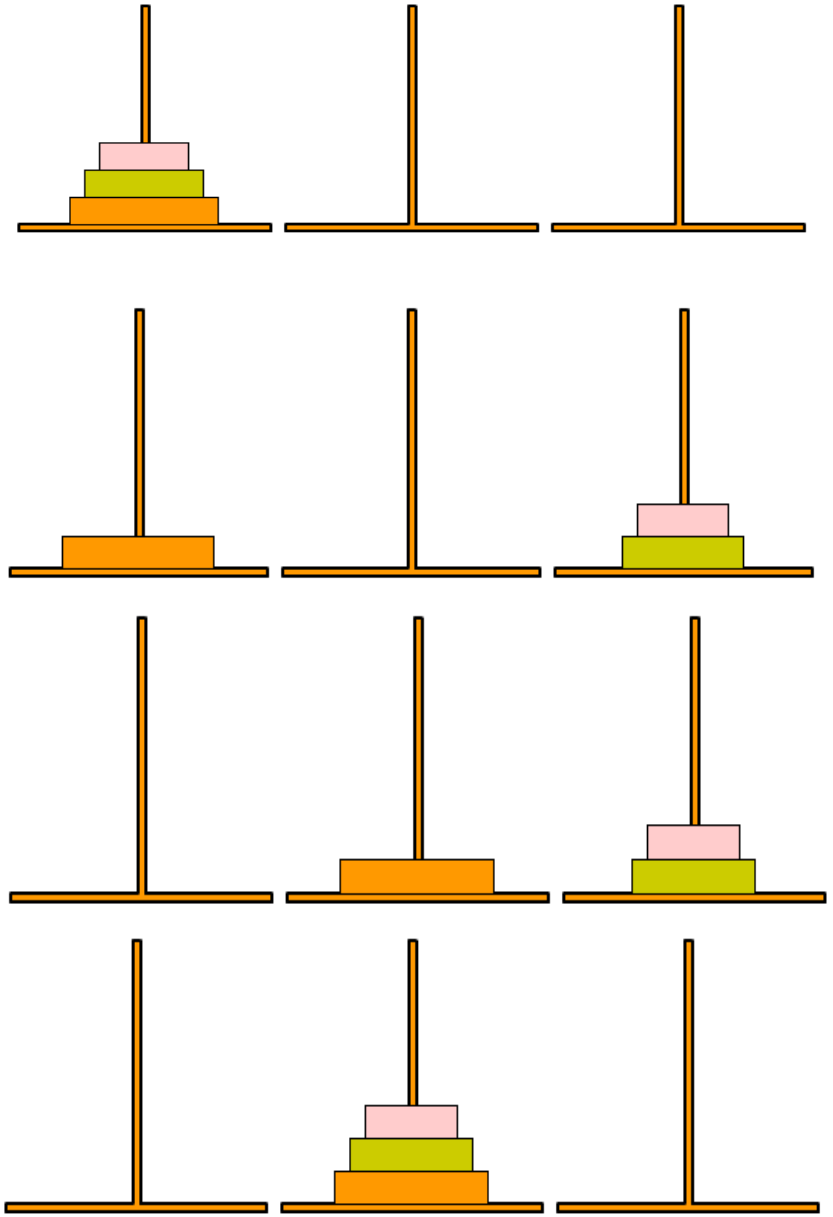
- Ponumerujmy drążki : 0, 1, 2
- Każdy ruch będzie opisany jako para numerów drążków np (0,2)



- Szukamy najkrótszej sekwencji ruchów przenoszącej N krążków z drążka 0 na 1.

Wieże Hanoi: rekurencja

- Aby przenieść N krążków muszę
  - najpierw przenieść N-1 “górnych” krążków na “trzeci” drążek,
  - Potem przenoszę największy krążek na docelowy
  - Potem przenoszę znowu N-1 krążków z “trzeciego” na docelowy



Wieże Hanoi - rozwiązanie

```
def hanoi(n,skad=0, dokad=1):
    if n==1:
        return [(skad,dokad)]
    else:
        if skad+dokad==1:
            trzeci = 2
        elif skad+dokad==2:
            trzeci = 1
        else: #skad+dokad==3
            trzeci = 0
        rozwiazanie = hanoi(n-1,skad,trzeci)
        rozwiazanie +=[(skad,dokad)]
        rozwiazanie +=hanoi(n-1,trzeci,dokad)
        return rozwiazanie
```

O sortowaniu przez scalanie

- Rekurencja jest często stosowana w metodzie “dziel i zwyciężaj” (ang. divide and conquer)
- Ta metoda pozwala efektywnie skrócić działanie algorytmów
- Jak posortować listę:
  - Podziel ją na pół
  - **Posortuj** każdą połówkę **rekurencyjnie**
  - **Scal** posortowane połówki w jedną posortowaną listę

Scalanie też może być rekurencyjne

- Aby scalić dwie posortowane listy wystarczy, że sprawdzę która z nich ma mniejszy pierwszy element.
- Mniejszy z nich na pewno jest najmniejszy ze wszystkich,

  a więc znajdzie się na początku ostatecznej listy

```
def merge(l1,l2):
    if l1==[]:
        return l2
    elif l2==[]:
        return l1
    elif l1[0]<l2[0]:
        return [l1[0]]+merge(l1[1:],l2)
    else:
        return [l2[0]]+merge(l1,l2[1:])
```

Kiedy już umiemy scalać, możemy sortować

```
def m_sort(l):
    if len(l)<=1:
        return l # posortowana
    else: #len(l) >= 2
        l1=l[:len(l)/2] #pierwsza połówka
        l2=l[len(l)/2:] # druga połówka
        return merge(m_sort(l1),m_sort(l2))
```

Rekursja nie zawsze jest najlepszym rozwiązaniem

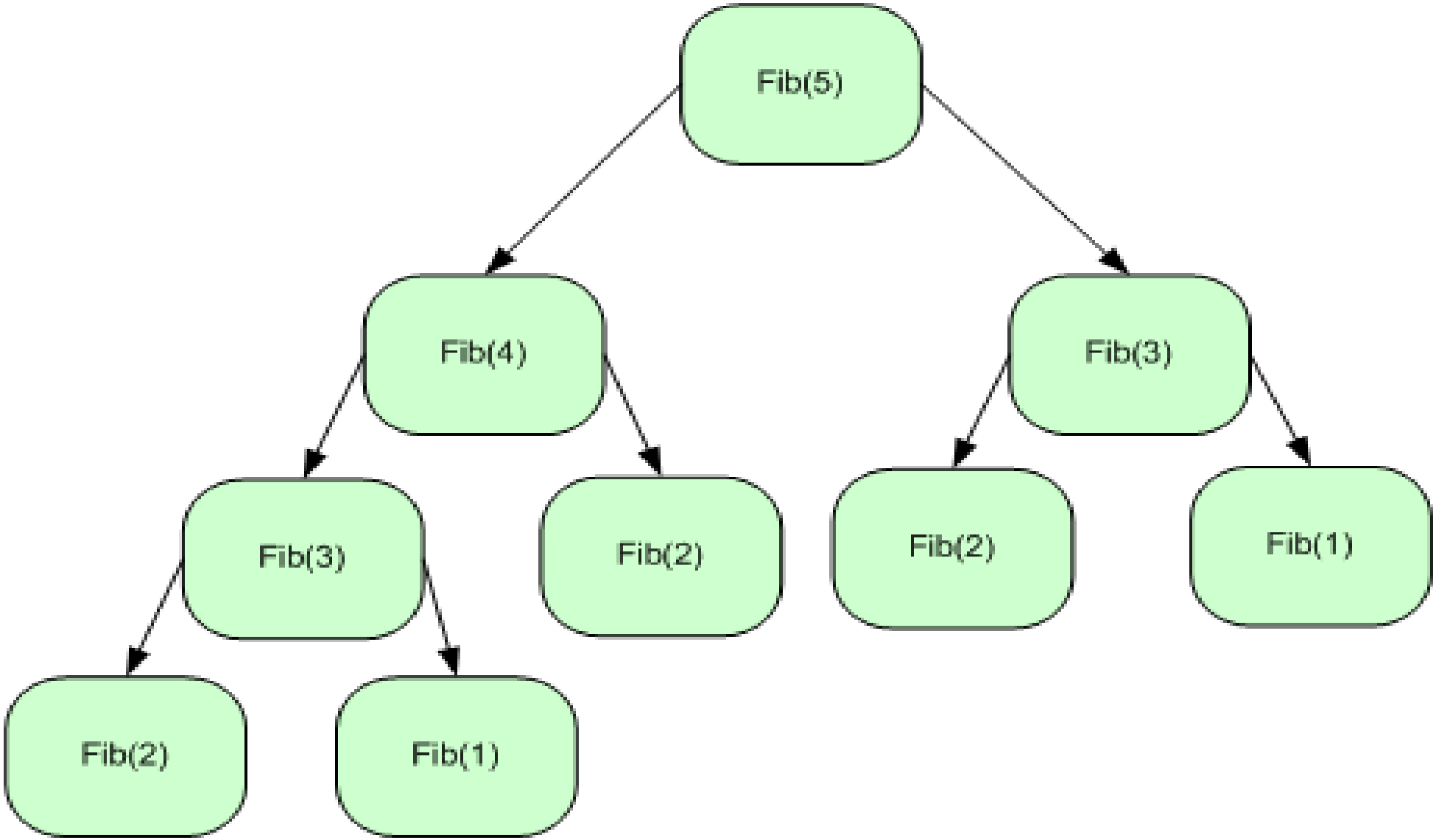
Liczby Fibonacciego:

$$F_n = F_{n-1} + F_{n-2}$$

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

```
def fib_iter(n):
    f=[1,1]
    for i in range(2,n+1):
        f.append(f[-1]+f[-2])
        print f
    return f[n]
```

Gdzie jest problem?



O głębokości rekursji

```
In [45]: silnia_r(1000)
```

.....

```
/home/bartek/<ipython-input-3-e02c11fe5a07> in silnia_r(n)
      3         return 1
      4     else:
----> 5         return n*silnia_r(n-1)
      6

/home/bartek/<ipython-input-3-e02c11fe5a07> in silnia_r(n)
      3         return 1
      4     else:
----> 5         return n*silnia_r(n-1)
      6

RuntimeError: maximum recursion depth exceeded
```

Podsumowanie

- Rekurencja jest często bardzo dobrym i zwięzłym sposobem wyrażania algorytmów
- Można przy jej pomocy łątwo opisywać rozwiązania, które trudno byłoby opisać inaczej
- Rekurencja leży u podstaw wielu technik takich jak “dziel i zwyciężaj”
- Trzeba uważać na możliwe wielokrotne wykonanie tego samego kodu i głębokość rekurencji