

ALGORYTMY I STRUKTURY DANYCH

WYKŁAD 3 i 4 - REKURENCJA

JANUSZ
SZWABIŃSKI

Plan wykładu

1. Co to jest rekurencja?
2. Przykład - silnia
3. Rekurencja kontra iteracja
4. Przykłady
 - 4.1 Wyśrodkowanie binarne
 - 4.2 System plików
 - 4.3 Symbol Newtona
 - 4.4 Cechy podzielności przez 3
 - 4.5 Konwersja liczby naturalnej do dziesięciorazowej
 - 4.6 Wieża z Hanoi
5. Analiza algorytmów rekurencyjnych
6. Typowe problemy z rekurencjami
7. Kolejne przykłady
 - 7.1 Rekurencja liniowa
 - 7.2 Rekurencja binarna
8. Projektowanie algorytmów rekurencyjnych

1. CO TO JEST REKURENCJA?

- metoda polegająca na rozwiązywaniu problemu w oparciu o rozwiązywanie tego samego problemu dla danych o mniejszych rozmiarach
- technika programistyczna polegająca na wywołaniu funkcji przez samą siebie
- alternatywny do pętli sposób wykonywania powtarzanych działań
- bardzo efektywna technika
 - zwięzły opis algorytmu

- Tatuja w implementacji

2. SILNIA

Silnia liczby naturalnej n , oznaczana jako $n!$, to iloczyn kolejnych liczb naturalnych od 1 do n ,

$$n! = \begin{cases} 1 \cdot 2 \cdot 3 \cdots \cdot (n-1) \cdot n = \prod_{k=1}^n k, & n \geq 1 \\ 1, & n=0 \end{cases}$$

wartość $0!$ okresla się osobno

Zauważmy, że np.

$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 4! \cdot 5$$

Zatem silnia ma bardziej naturalną rekurencyjną wersję swojej definicji

$$n! = \begin{cases} (n-1)! \cdot n, & n \geq 1 \\ 1, & n=0 \end{cases}$$

Implementacja tej definicji jest bardzo prostą:

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

- brak żarzących pętli
- powtarzanie realizowane są przez kolejne wywołanie funkcji

Ciąg wywołań nazywa się łańcuchem przyjmując tzw. **slacku rekurencyjnego**, który odróżnia się od wywołania funkcji na komputerze:

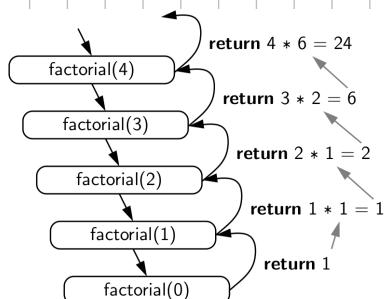


Figure 4.1: A recursion trace for the call `factorial(5)`.

- przy każdym wywołaniu funkcji tworząca jest nowa ramienna nadająca przestroni nazw dla parametrów funkcji, jej miejscowe lokalne i aktualnie wykonywanie polecenie

- jeśli funkcja wywołuje samą siebie, kolejne wywołanie jest zawsze ta sama, w której przekazywana jest informacja o miejscu, z którego należy później kontynuować
- nowa ramka dla kolejnego wywołania

UWAGA! W bibliotece math znajdują się gotowe implementacje silni

```
import math
math.factorial(5)
```

3. REKURENCJA KONTRA ITERACJA

- programy rekurencyjne są z reguły bardzo przyjazne
- dla pewnych problemów rekurencja stwarza natychmiastowy wybór
- podstawowa technika programowania w językach funkcyjnych (Haskell, Lisp)
- ma jednak pewne ograniczenia
 - czas wykonania
 - zapotrzebowanie na pamięć operacyjną
 - możliwość może dawać越来越大的值, jeśli problem nie ma rekurencyjnego charakteru

Rozważmy iteracyjną wersję funkcji silnia:

```
def fac_iter(n):
    sil = 1
    if n > 1:
        for i in range(2, n+1):
            sil = sil*i
    return sil
```

fac (120)	12,7 μs
fac_iter (120)	5,37 μs



Biorąc pod uwagę tylko wydajność, nie ma w zasadzie powodów, dla którego warto stosować rekurencję

4. PRZYKŁADY

4.1. WYSZUKIWANIE BINARNE

Wyszukiwanie binarne to jeden z najciekawszych algorytmów w informatyce. Głównie ze względu na tego przedstawiające się dane w postaci sekwencji uporządkowanych.

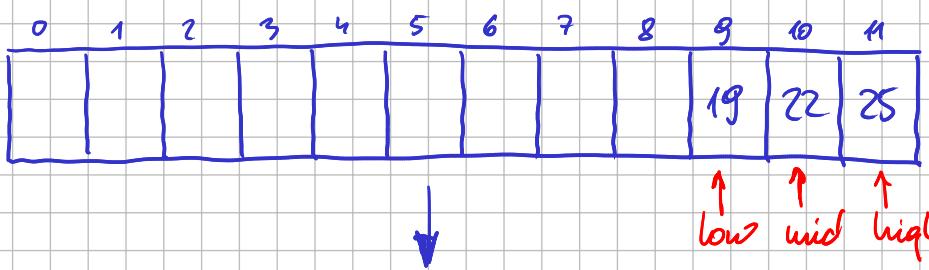
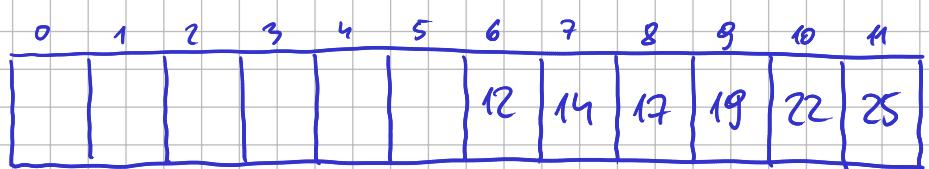
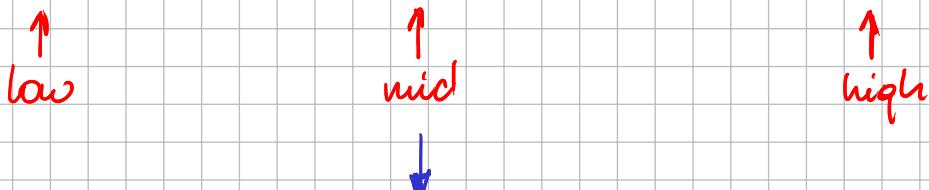
Jeśli sekwencja nie jest uporządkowana, konieczny jest wyszukiwanie liniowego \rightarrow sprawdzamy w postaci każdej element sekwencji. Złożoność takiego wyszukiwania to $O(n)$.

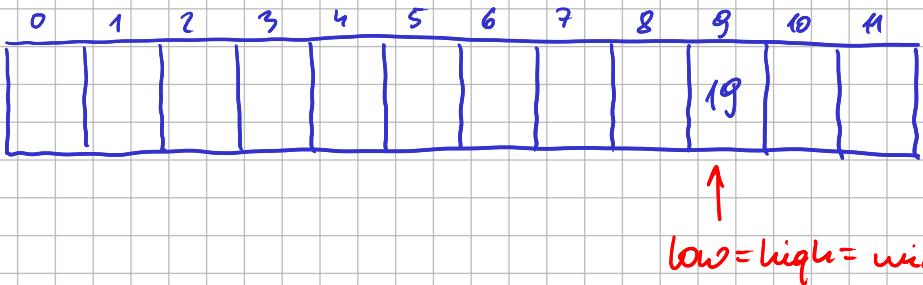
W przypadku wykazuje się, że wyszukiwanie binarne ma złożoność $O(\log n)$.

Priporumijmy sobie, jak działa ten algorytm.

W poniższej sekwencji będziemy szukać wartości 19

0	1	2	3	4	5	6	7	8	9	10	11
2	4	5	7	8	9	12	14	17	19	22	25





Rekurencyjna wersja tego algorytmu będzie miała postać:

```
def binary_search(data, target, low, high):
    if low > high:
        return False # interval is empty; no match
    else:
        mid = (low + high) // 2
        if target == data[mid]: # found a match
            return True # or mid
        elif target < data[mid]:
            # recur on the portion left of the middle
            return binary_search(data, target, low, mid - 1)
        else:
            # recur on the portion right of the middle
            return binary_search(data, target, mid + 1, high)
```

4.2. SYSTEMY PLIKÓW

W nowoczesnych systemach plików katalogi rozgałęzione są rekurencyjnie

→ katalog główny zawiera pliki i katalogi, które zawierają pliki i katalogi itd.

W związku z tym, wiele operacji w systemie plików może być realizowane rekurencyjnie.

PRZYKŁAD : wyznaczenie całkowitej powierzchni na dysku zajmowanej przez pliki i katalogi znajdujące się w wybranym katalogu

- całkowita powierzchnia zajmowana przez katalog to
 - własne powierzchni zajmowane przez ten katalog
 - całkowita powierzchni jego podkatalogów

→ algorytm powiniene być zdecidowanie prosty:

Algorytm DiskUsage (path):

Input: Tarczki maków wskazujące katalog startowy

Output: całkowita przestrzeń dyskowa zajmowana przez ten katalog

total = size (path)

if path represents a directory then

for each child entry stored within directory path do

 total = total + DiskUsage (child)

return total

Do implementacji tego algorytmu wykorzystujemy model OS

```
import os
```

```
def disk_usage(path):
    total = os.path.getsize(path)
    if os.path.isdir(path):
        for filename in os.listdir(path):
            childpath = os.path.join(path, filename)
            total += disk_usage(childpath)
    print('{0:<7}'.format(total), path)
    return total
```

ten sposób pozwala na głęboką rekurencję automatycznie

4.3. SYMBOL NEWTONA

Symbol Newtona redefiniowany jest jako

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

Symbol ten pojawia się we wzorze określającym Newtona jako współczynnik w k-tym wyrazie:

$$(x+y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k$$

Powojenna definicja symbolu binomialnego jest następującym wzorem rekurencyjnym

$$\binom{n}{k} = \begin{cases} 1, & k \in \{0, n\} \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & 0 < k < n \end{cases}$$

Zadanie

- wie mamy wyliczac silni, ktore powoduje szybkie wejscie pora rzes w niektorych dziedzinach programowania

```
def binom(n,k):
    if n<0 or k < 0:
        print("Błędne argumenty na wejściu")
        return None
    if k==0:
        return 1
    if n==k:
        return 1
    else:
        return binom(n-1,k-1) + binom(n-1,k)
```

binom(7,2)

↳ 21

4.4. CECHA PODZIELNOSCI PRZEZ 3

Cechka podzielnosci pozwala na stwierdzenie, czy dana liczba jest podzielna bez reszy przez inną bez uckania się do dzielenia.

Liczba jest podzielna przez 3, jeśli suma cyfr tej liczby jest podzielna przez 3.

zauważmy, iż możemy tego moim sposobem skorzystać z
dzielenia liczb jednocyfrowych, której
podzielność jest bardzo łatwo określić:

$$104\ 628 \rightarrow 1+0+4+6+2+8 = 21 \rightarrow 2+1 = \underline{\underline{3}}$$

Aby naimplementować sprawdzenie podzielności
przez 3, musimy umieć zamienić dowolną liczbę
na cyfry i sumować je.

W Pythonie możemy to zrobić, przekształcając
liczby na listy znaków

```
number = 2456
s = str(number)
print(s)
```

↳ '2456'

Następnie z listy tworzymy listę i konwertujemy
ją na listę liczb całkowitych

```
I = list(s)
figs = [int(i) for i in I]
print(figs)
```

↳ [2, 4, 5, 6]

Ostatni krok to suma elementów listy

```
sum(figs)
```

↳ 17

Konczącące z polecenia `map`, można napisać
powyższe kroki w jednym wierszu:

```
sum(map(int,str(number)))
```

↳ 17

Teraz jesteśmy gotowi do implementacji właściwej funkcji

```

def divisible_by_3(number):
    ret = False
    if number in (3,6,9):
        ret = True
    if number > 9:
        ret = divisible_by_3(sum(map(int, str(number))))
    return ret

```

4.5 KONWERSJA LICZBY NA TANIECHUZNAKÓW W DOWOLNEJ REPREZENTACJI

Zatóżmy teraz, że nie mamy funkcji `str` i musimy 'ręcznie' przekonwertować liczby do taniechuznaków. Dla ustalenia uważmy dowolną reprezentację dziesięciu.

Zdefiniujmy taniechuznaków

`convString = "0123456789"`

W przypadku liczb od 0 do 9 konwersja jest prosta:

`convString[9]`

↳ '9'

Aby przekonwertować większą liczbę, np. 769, musimy rozbić ją na cyfry, każdą z cyfr zamienić na znaki i połączyc znaki ze sobą.

Zauważmy, że

$$769 \text{ // } 10 = 76 + 9$$

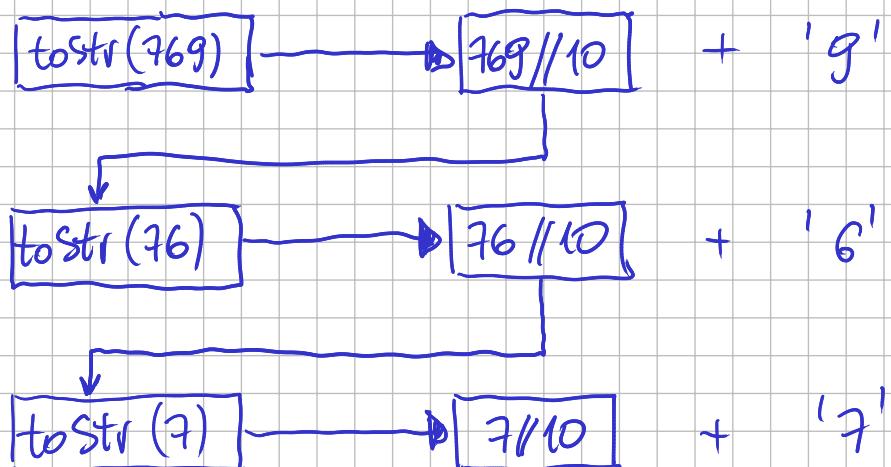
↑ ↑
basis representation
dzielenie całkowite

reszta (jedna z cyfr)

$$76 // 10 = 7 + \textcircled{6}$$

$$7 // 10 = 0 + \textcircled{7}$$

Schemat algorytmu rekurencyjnego przedstawia się następująco



Implementacja (dla dowolnej bazy ≤ 16)

```
def toStr(n,base):  
    convertString = "0123456789ABCDEF"  
    if n < base:  
        return convertString[n]  
    else:  
        return toStr(n//base,base) + convertString[n%base]
```

`toStr(1453,10)`

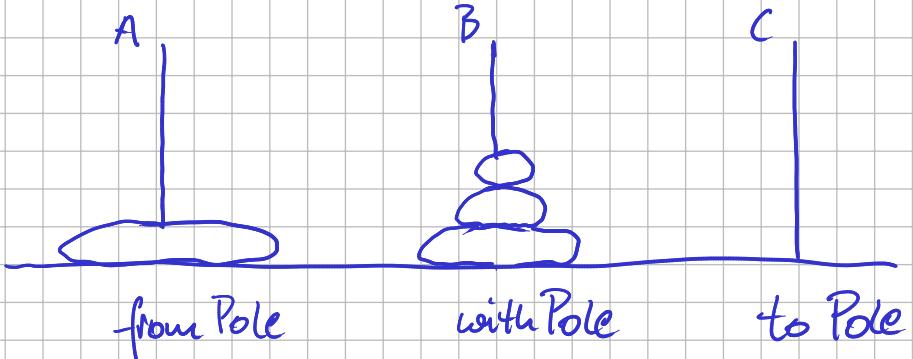
↳ '1453'

`toStr(1453,2)`

↳ '10110101101'

4.6. WIEZA HANOI

Wichtigosć dotyczy czasowej przykładowi użycia definicji rekurencyjnej. Dlatego zastosowanie rekurencji w ich przypadku było naturalnym wyborem. Metoda ta sprawdza się jednak również w przypadku problemów, które we pierwszy rzut oka nie wydają się rekurencyjne.



- zadanie uznawane w Azji
- do Europy sprowadzone przez franc. mat. Édouarda Lucasa w 1883 roku

Dla n kożików najmniejsza liczba wymaganych ruchów wynosi

$$L(n) = 2^n - 1$$

Dla $n = 64$ daje to

$$2^{64} - 1 = 18446744073709551615$$

niechc. zakładając 1 ruch na sekundę, rozwiązanie taniotówkii potrwa

$$58\,494\,241\,7355 \text{ lat}$$

Czyż nie można rozwiązać zadania na komputerze, rozwiązać, iż problem da się napisać w postaci stocznikowego prostego algorytmu rekurencyjnego:

1. Przenieś (rekurencyjnie) $n-1$ kożików ze stepa A na step B wykorzystując step C.
2. Przenieś kożika ze stepa A na step C.
3. Przenieś (rekurencyjnie) $n-1$ kożików ze stepa B na step C wykorzystując step A.

Przykładowa implementacja

```
from   with  
def moveTower(n,A,C,B):  
    if n >= 1:  
        moveTower(n-1,A,B,C) to  
        moveDisk(A,C)  
        moveTower(n-1,B,C,A)  
  
def moveDisk(fp,tp):  
    print("moving disk from",fp,"to",tp)
```

5. ANALIZA ALGORYTMÓW REKURENCYJNYCH

- w przypadku rekurencji, dla każdej wywołania funkcji będącej zliczać tylko te operacje, które wykonujemy są w obiekcie tego wywołania
- ślad rekurencji daje nam informacji na temat liczb wywołań

5.1 SILNIA

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

- $n+1$ wywołań
 - w każdej wywołaniu state liczb operacji
- $\Rightarrow O(n)$

Mówiąc do analizy pojęć wieco innego.

Niech:

- $T(n)$ - czas potrzebny do wykonania funkcji rekurencyjnej dla dowolnego n
- $T(0)=1$ - jednoznaczny czas pracy dla wejścia $n=0$ lub $n=1$, z rozszerzeniem o kolejny 1

Mamy

$$T(n) = T(n-1) + 1 \quad , \quad n > 0$$

$$T(0) = 1$$

$$T(1) = T(0) + 1 = 2$$

$$T(2) = T(1) + 1 = 3$$

:

:

:

$$T(n) = T(n-1) + 1 = n + 1$$

$$\Leftrightarrow T(n) = O(n)$$

5.2 WYSZUKIWANIE BINARNE

Dla danej listy wejściowej algorytm wymaga jednego sprawdzenia elementu środkowego, a następnie przekształca jedynego z potówek:

$$T(n) \leq T\left(\frac{n}{2}\right) + 1$$

$$T(1) = 1$$

Niech $n = 2^m$ ($\Rightarrow m = \log_2 n$).

Wówczas:

$$\begin{aligned} T(2^m) &\leq T\left(\frac{2^m}{2}\right) + 1 \\ &\leq T\left(\frac{2^m}{2^2}\right) + 1 + 1 \\ &\vdots \\ &\leq T\left(\frac{2^m}{2^m}\right) + m = T(1) + m \end{aligned}$$

Czyli

$$T(n) = O(\log_2 n)$$

6. TYPOWE PROBLEMY Z REKURENCJĄ

6.1. WYDAJNOŚĆ

Rozważmy ciąg fibonacciego:

$$F_0 = 0$$

$$F_1 = 1$$

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

Bezpośrednia implementacja tej definicji będzie miała wydajność:

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

Obliczenie n -tego wyrazu ciągu będzie wymagało wykonywania liczby wywołań funkcji. Mamy:

$$C_0 = 1$$

$$C_5 = 1 + C_3 + C_4 = 15 \quad \text{w ogólności}$$

$$C_1 = 1$$

$$C_6 = 1 + C_4 + C_5 = 25$$

$$C_2 = 1 + C_0 + C_1 = 3$$

$$C_7 = 1 + C_5 + C_6 = 41$$

$$C_3 = 1 + C_1 + C_2 = 5$$

$$\textcircled{C_8} = 1 + C_6 + C_7 = 67$$

$$C_4 = 1 + C_2 + C_3 = 9$$

liczba wywołań

$$C_n > 2^{n/2}$$

Co więcej, wynikające z tych wywołań będą wykazywać F_{n-2} , zatem bardzo dużo wykonywań będzie się powtarzać.

Ależ zredukować liczbę wywołań, mówiąc zrobić coś takiego:

```
def good_fibonacci(n):
    if n <= 1:
        return (n,0)
    (a,b) = good_fibonacci(n-1)
    return (a+b,a)
```

wyrazy F_n i F_{n-1}

konwencja $F_{-1} = 0$

- n wywołań

- $O(n)$

6.2 GŁĘBOKOŚĆ REKURENCJI (W PYTHONIE)

Zatem, że naszym celem jest obliczenie liczby harmonicznej H_n :

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = \sum_{k=1}^n \frac{1}{k}$$

Definicję moim zapisać rekurencyjnie:

$$H_n = \begin{cases} 1 & , n=1 \\ H_{n-1} + \frac{1}{n} & , n \geq 1 \end{cases}$$

```
def H(n):
    if n==1:
        return 1.0
    return H(n-1) + (1/n)
```

$H(2)$

↳ 1.5

$H(10)$

↳ 2,92896825

$H(4000)$

↳ RecursionError: maximum recursion depth exceeded in comparison

- zabezpieczenie przed „niekontrolowaną” rekurencją
- moim zamiarów domyslnie (gęb. ~3000) ustalenie

```
import sys
old = sys.getrecursionlimit()
print(old)
sys.setrecursionlimit(5000)
```

↳ 3000

- tymczasowy przepchnięcie paczki !!

7. DALSZE PRZYKŁADY REKURENCJI

7.1 REKURENCJA LINIOWA

- każde wywołanie funkcji prowadzi do jednego wywołania rekurencyjnego
- warunek nie jest zawsze takiż zauważalny i złożoność

7.1.1. SUMOWANIE ELEMENTÓW SEKWENCJI

Czujemy obliczyć sumę elementów sekwencji (up. listy)

- o długości n

seq =	0	1	2	3	4	5	6	7	8	9
	4	3	6	2	8	9	3	2	8	5

$$S(n) = \begin{cases} 0, & n=0 \quad \leftarrow \text{sekwencja pusta, stąd } 0 \\ S(n-1) + \text{seq}[n-1], & n > 0 \end{cases}$$

```
def linear_sum(S,n):  
    if n==0:  
        return 0  
    return linear_sum(S,n-1) + S[n-1]
```

$$T(n) = O(n)$$

7.1.2. ODWRACANIE SEKWENCJI

Algorytm:

1. Zamieniamy element pierwszy z ostatnim.
2. Rekurencyjnie przetwarzamy pozostałe elementy.

0	1	2	3	4	5	6
4	3	6	2	8	9	5

start ↑ stop ↑

0	1	2	3	4	5	6
5	3	6	2	8	9	4

start ↑ stop ↑

0	1	2	3	4	5	6
5	9	6	2	8	3	4

start ↑ stop ↑

0	1	2	3	4	5	6
5	9	8	2	6	3	4

start = stop ↑

Dwie przypadki biorące

- $\text{start} == \text{stop}$ \rightarrow kolejna sekwencja jest pusta, nie ma co robić (nieparzyste liczby elementów)
- $\text{start} == \text{stop} - 1$ \rightarrow poroztac jeden element.

def reverse(S,start,stop):

if start <= stop-1:

. S[start], S[stop] = S[stop], S[start]
reverse(S,start+1,stop-1)

- $1 + \frac{n}{2}$ wegaotanii sekwencjinych
 - stała liczba operacji niesekencjinych w kaicym wegaotaniu
- $\Rightarrow T(n) = O(n)$

7.1.3 POTĘGOWANIE

Korzystając z faktu, że $X^n = X \cdot X^{n-1}$ ($n \geq 0$), możemy zapisać definicję sekwencjinych

$$\text{power}(x,n) = \begin{cases} 1 & , n=0 \\ x \cdot \text{power}(x,n-1) & , n > 0 \end{cases}$$

def power(x,n):

if n==0:

return 1

return x*power(x,n-1)

$$T(n) = O(n)$$

Mogąca tą funkcję uzupełnić. Niech

$$k = \left\lfloor \frac{n}{2} \right\rfloor$$

oznacza funkcję "podwoje" (czyli $k = n/2$). Rozważamy wegaotenice

$$(x^k)^2$$

Jeśli n jest parzyste, wtedy

$$\left\lfloor \frac{n}{2} \right\rfloor = \frac{n}{2} \Rightarrow (x^k)^2 = \left(x^{\frac{n}{2}}\right)^2 = x^n$$

Zatem

$$\text{power}(x, n) = \begin{cases} 1, & n=0 \\ x \cdot (\text{power}(x, \lfloor \frac{n}{2} \rfloor))^2, & n>0 \text{ i nieparzyste} \\ (\text{power}(x, \lfloor \frac{n}{2} \rfloor))^2, & n>0 \text{ i parzyste} \end{cases}$$

```
def power2(x,n):
    if n==0:
        return 1
    partial = power(x,n//2)
    result = partial*partial
    if n%2 == 1:
        result *= x
    return result
```

zapisującą rekurencję poprawia wydajność, usiądzie $\text{power}(x, n/2) * \text{power}(x, n/2)$ dostały $O(n)$

$$T(n) = O(\log n) \quad (\text{jak w binary-search})$$

7.2. REKURENCJA BINARNA

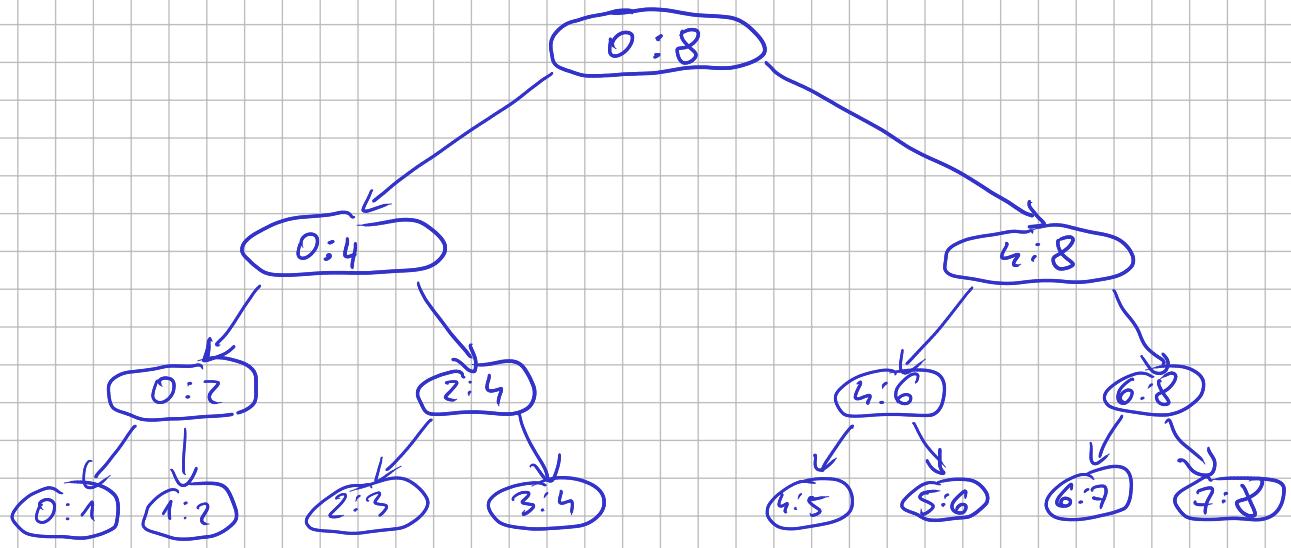
- funkcja łączy dwukrotnie siebie

Przykład - sumowanie elementów sekwencji

- możemy rekurencyjnie wyliczyć sumę obu połówkó
- dodajemy wyniki do siebie

```
def binary_sum(S,start,stop):
    if start >= stop:
        return 0
    if start == stop-1: #tylko jeden element
        return S[start]
    mid = (start + stop)//2
    return binary_sum(S,start,mid) + binary_sum(S,mid,stop)
```

Zauważmy, że $n = 2^m$. Ślad rekurencji ma w tym wypadku postać



- głębokość rekurencji :

$$1 + \log_2 n$$

→ wymagania dodatkowej pamięci
 $O(\log_2 n)$

- $2n - 1$ wywołani



$$T(n) = O(n)$$

8. PROJEKTOWANIE ALGORYTMÓW REKURENCYJNYCH

Wszystkie funkcje powinny mieć postać:

- przypadek bazowy
- pewne operacje nierekurencyjne
- rekurencyjne wywołanie samej siebie