# Algorytmy i struktury danych

## Wykład 1 - Wprowadzenie

### Janusz Szwabiński ¶

Plan wykładu:

# Sprawy administracyjne

### Dane kontaktowe

- email: janusz.szwabinski@pwr.edu.pl
- konsultacje: poniedziałek 13:00-15:00, środa 14:00 - 16:00, pokój 5.16, C-11
- http://prac.im.pwr.edu.pl/~szwabin/ (http://prac.im.pwr.edu.pl/~szwabin/)

### Treści programowe

- Podstawowe struktury danych
- Algorytmy i ich analiza
- Rekursja
- Sortowanie i wyszukiwanie
- Drzewa i algorytmy ich przetwarzania
- Grafy i podstawowe algorytmy
- Techniki projektowania algorytmów
- Wybrane zagadnienia (m.in. operacje na macierzach, wielomiany, FFT, wyszukiwanie wzorców w tekstach)

### Literatura

1. T. H. Cormen, Ch. E. Leiserson, R. L. Rivest. Wprowadzenie do algorytmów. WNT, Warszawa, 1997.
2. B. Miller, D. Ranum. Problem Solving with algorithms and data structures using Python. http://interactivepython.org/runestone/static/pythonds/index.html (http://interactivepython.org/runestone/static/pythonds/index.html) (**w ramach kursu będę często korzystał z materiałów tam zawartych**).
3. R. Sedgewick. Algorytmy w C++. RM, Warszawa, 1999.
4. R. Sedgewick. Algorytmy w C++.Grafy. RM, Warszawa, 2003.
5. L. Banachowski, K. Diks, W. Rytter. Algorytmy i struktury danych. WNT, Warszawa, 1996.
6. A. V. Aho, J. E. Hopcroft, J. D. Ullman. Projektowanie i analiza algorytmów komputerowych. PWN, Warszawa, 1983; Helion, Gliwice, 2003.
7. E. M. Reingold, J. Nievergelt, N. Deo. Algorytmy kombinatoryczne. PWN, Warszawa, 1985.
8. B. Eckel. Thinking in C++. Edycja polska. Helion, Gliwice, 2002.
9. B. Eckel. Thinking in Java. Edycja polska. Helion, Gliwice, 2001, 2003.

### Materiały on-line

1. Erik Demaine, and Srinivas Devadas. 6.006 Introduction to Algorithms, Fall 2011. (Massachusetts Institute of Technology: MIT OpenCourseWare), http://ocw.mit.edu (http://ocw.mit.edu) (Accessed 19 Sep, 2016). License: Creative Commons BY-NC-SA.

### Warsztat

1. Preferowanym językiem programowania jest Python 2.7.X/3.X.
2. Dowolny edytor tekstowy ze wsparciem składni Pythona (np. Notepad++ pod systemami MS Windows, geany/vi pod systemami z rodziny Linux/Unix).
3. Ewentualnie IPython/Jupyter.

### Zaliczenie

- listy zadań na ćwiczeniach

- dwa testy: na 7 i ostatnim wykładzie

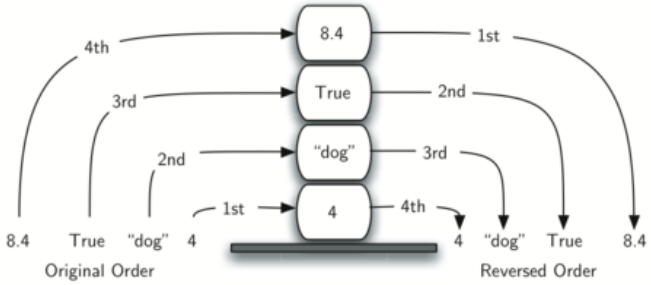
$$ocena = \frac{3}{4}lab + \frac{1}{4}wyk$$

## Do czego służą struktury danych?

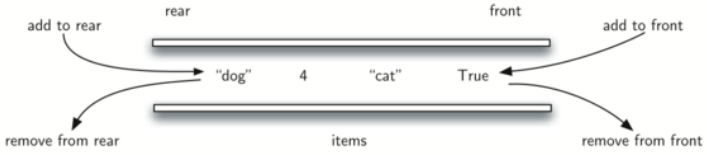
**Abstrakcyjna struktura danych** (ASD) to zbiór danych elementarnych wraz z dobrze zdefiniowanym na nich zbiorem operacji:

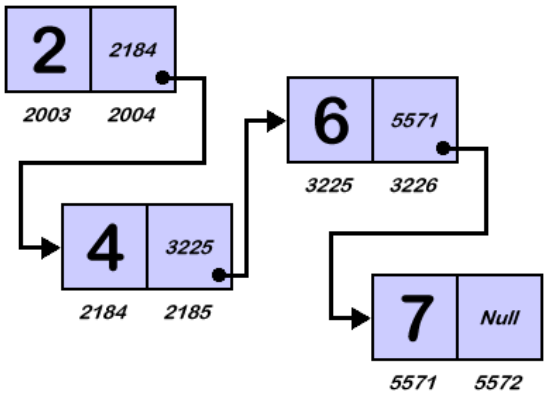
- zaawansowane pojemniki na dane
- pomagają gromadzić różnorodne dane i układać je w odpowiedni sposób
- dopasowują się rozmiarem do wielkości danych
- pojawiają się w wielu algorytmach
- pozwalają na eleganckie rozwiązanie wielu ważnych zagadnień obliczeniowych

### Liniowe struktury danych

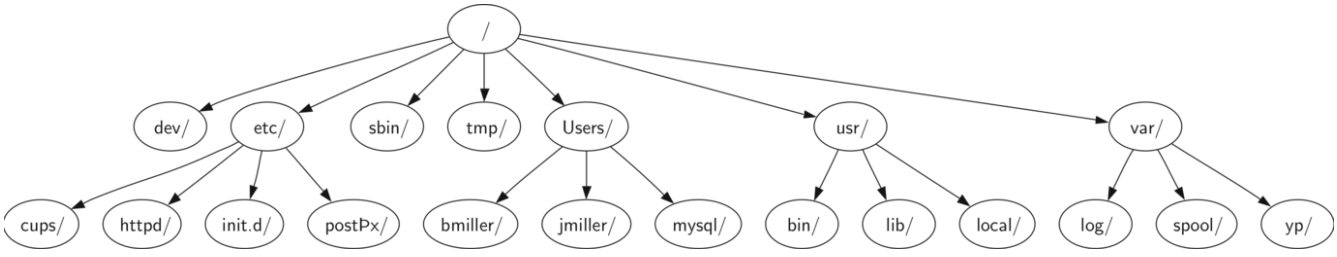




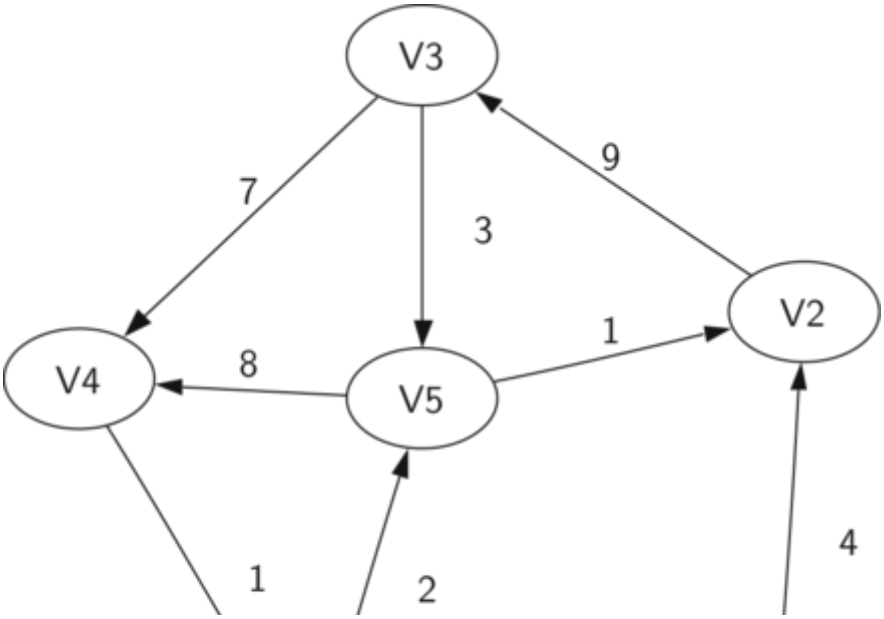



### Drzewa




### Grafy




## Co to jest algorytm?

**Algorytm** to skończony ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego rodzaju zadań. Innymi słowy to przepis na rozwiązanie problemu lub osiągnięcie jakiegoś celu.

Istnieje kilka różnych metod zapisu algorytmu. Załóżmy, że naszym zadaniem jest obliczenie funkcji

$$f(x) = \frac{x}{|x|}$$

przy założeniu, że $f(0) = 0$.

### Słowny opis algorytmu

1. dla liczb ujemnych $|x| = -x$, więc $f(x) = x/(-x) = -1$,
2. dla liczb dodatnich $|x| = x$, więc $f(x) = x/x = 1$,
3. jeśli $x = 0$, to z definicji $f(0) = 0$.

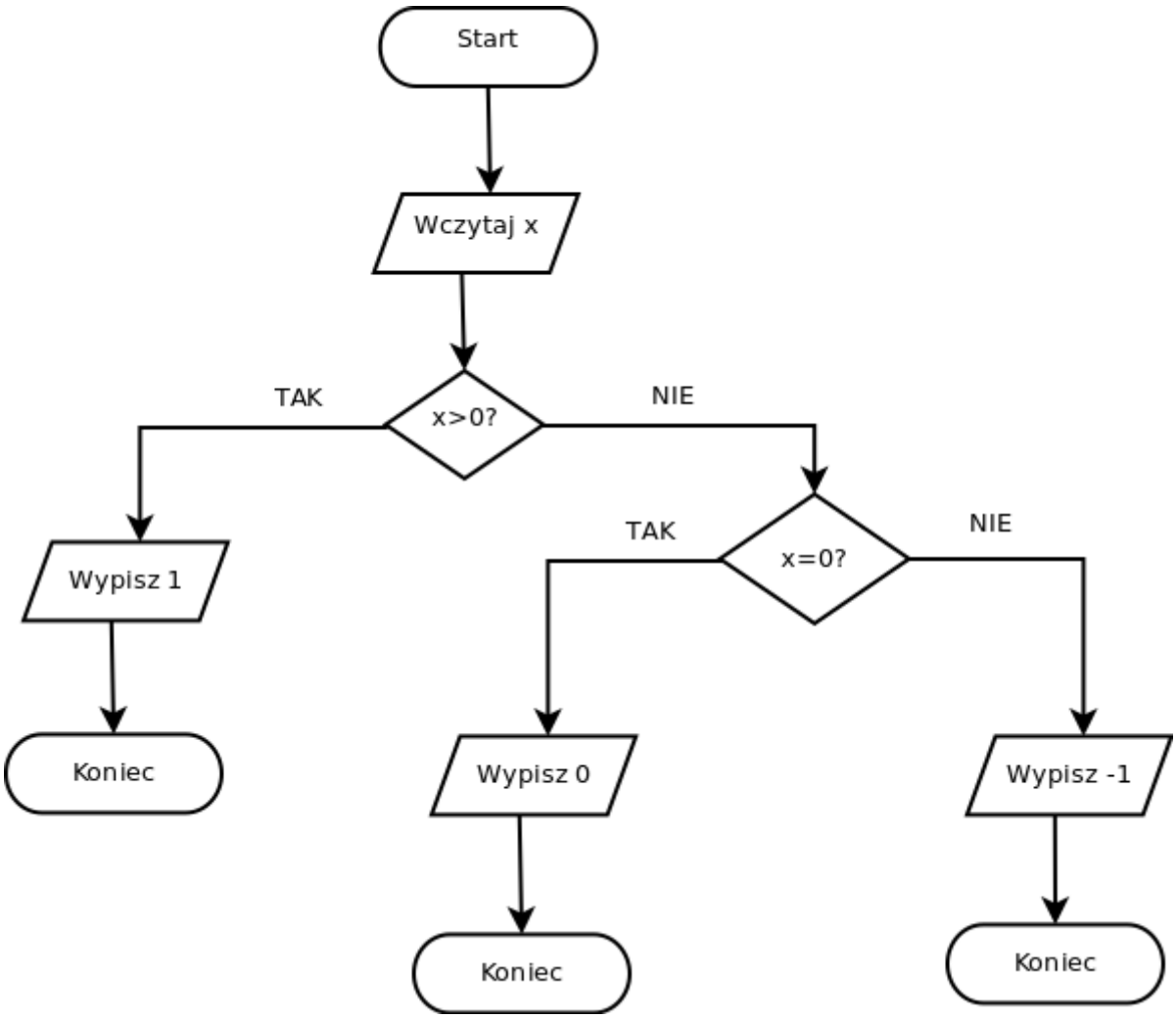
Opis słowny czasami da się w prosty sposób wyrazić wzorem matematycznym:

$$f(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}$$

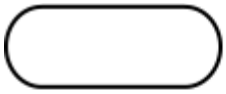
### Lista kroków

1. Wczytaj wartość danej $x$.
2. Jeśli $x > 0$, to $f(x) = 1$. Zakończ algorytm.
3. Jeśli $x = 0$, to $f(x) = 0$. Zakończ algorytm.
4. Jeśli $x < 0$, to $f(x) = -1$. Zakończ algorytm.

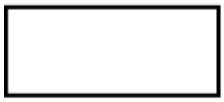
### Schemat blokowy




Poszczególne elementy na powyższym schemacie mają następujące znaczenie:



Blok startowy/końcowy algorytmu



Blok wykonawczy

Blok warunkowy

Uprzednio zdefiniowany proces

In [1]:

```
x = float(input("Podaj x: "))
if x > 0:
    print("f(x)=1")
elif x < 0:
    print("f(x)=-1")
else:
    print("f(x)=0")
```

```
Podaj x: 7
f(x)=1
```

**Dlaczego warto poznać algorytmy?**

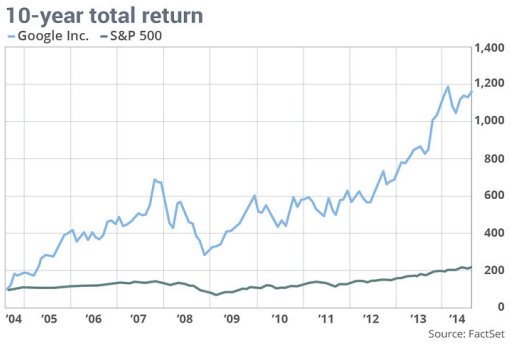
- cały obszar IT opiera się na algorytmach
- niektóre z nich są ponadczasowe:
  - Euklides z Aleksandrii zajmował się badaniem algorytmów już w IV w p.n.e.

(Źródło: Wikipedia)

  - jego algorytm wyznaczania największego wspólnego dzielnika dwóch liczb (NWD) jest znany do dziś
  - NWD jest stosowany w algorytmie RSA - najpopularniejszym obecnie asymetrycznym algorytmie kryptograficznym z kluczem publicznym (https://pl.wikipedia.org/wiki/Kryptografia_klucza_publicznego (https://pl.wikipedia.org/wiki/Kryptografia_klucza_publicznego))
- pozwalają rozwiązać zagadnienia, które inaczej pozostałyby nierozwiązane:
  - **problem komiwojażera**, np. wyznaczenie najkrótszej trasy pozwalającej na zwiedzenie wszystkich stolic województw
- pozwalają stać się profesjonalnym programistą

> I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships. (Linus Torvalds, twórca Linuksa)

- stymulują intelektualnie
- przynoszą zysk



(Źródło: http://www.marketwatch.com)

## Analiza algorytmów

Często bywa tak, że ten sam algorytm zaimplementowany jest na wiele różnych sposobów. Nasuwa się wtedy pytanie, która z implementacji jest lepsza od pozostałych. Dla przykładu porównajmy dwa programy:

In [2]:

```
def sumOfN(n):
   theSum = 0
   for i in range(1,n+1):
       theSum = theSum + i

   return theSum

print(sumOfN(10))
```

```
55
```

In [3]:

```
def foo(tom):
    fred = 0
    for bill in range(1,tom+1):
       barney = bill
       fred = fred + barney

    return fred

print(foo(10))
```

```
55
```

Mimo, że na pierwszy rzut oka tego nie widać, oba robią to samo - sumują liczby od $1$ do $n$, i na dodatek robią to w ten sam sposób. Mimo to powiemy, że pierwszy program jest lepszy ze względu na czytelność.

Ogólnie rzecz biorąc, aby dokonać porównania między programami, musimy zdefiniować odpowiednie kryteria. Oprócz czytelności mogą to być:

- liczba operacji niezbędnych do wykonania
- "zasobożerność"
- wydajność
- czas wykonania

**Analiza algorytmów** zajmuje się właśnie ich porównywaniem pod względem nakładów obliczeniowych niezbędnych do uzyskania rozwiązania. Wróćmy jeszcze raz do pierwszego z powyższych programów i zmodyfikujmy go tak, aby wyliczał jeszcze czas sumowania:

In [4]:

```
import time

def sumOfN2(n):
   start = time.time()

   theSum = 0
   for i in range(1,n+1):
      theSum = theSum + i

   end = time.time()

   return theSum,end-start
```

Wyniki pięciu wywołań funkcji `sumOfN2` dla $n = 10000$ są następujące:

In [5]:

```
for i in range(5):
       print("Suma wynosi %d, czas wykonania: %10.7f sekund"%sumOfN2(10000))
```

```
Suma wynosi 50005000, czas wykonania:  0.0075047 sekund
Suma wynosi 50005000, czas wykonania:  0.0018022 sekund
Suma wynosi 50005000, czas wykonania:  0.0013335 sekund
Suma wynosi 50005000, czas wykonania:  0.0012643 sekund
Suma wynosi 50005000, czas wykonania:  0.0014412 sekund
```

Czasy wykonania poszczególnych wywołań różnią się nieznacznie od siebie (zależą od chwilowego obciążenia komputera), jednak rząd wielkości pozostaje ten sam. Zobaczmy, co stanie się, jeżeli zwiększymy $n$ o jeden rząd:

In [6]:

```
for i in range(5):
       print("Suma wynosi %d, czas wykonania: %10.7f sekund"%sumOfN2(100000))
```

```
Suma wynosi 5000050000, czas wykonania:  0.0199480 sekund
Suma wynosi 5000050000, czas wykonania:  0.0153518 sekund
Suma wynosi 5000050000, czas wykonania:  0.0166814 sekund
Suma wynosi 5000050000, czas wykonania:  0.0098536 sekund
Suma wynosi 5000050000, czas wykonania:  0.0087562 sekund
```

I znowu, czasy wykonania są dość podobne do siebie, jednak w porównaniu z poprzednim przykładem wzrosły mniej więcej dziesięciokrotnie. Dla jeszcze większego $n$ otrzymamy:

In [7]:

```
for i in range(5):
       print("Suma wynosi %d, czas wykonania: %10.7f sekund"%sumOfN2(1000000))
```

```
Suma wynosi 500000500000, czas wykonania:  0.0854867 sekund
Suma wynosi 500000500000, czas wykonania:  0.0557914 sekund
Suma wynosi 500000500000, czas wykonania:  0.0546665 sekund
Suma wynosi 500000500000, czas wykonania:  0.0569293 sekund
Suma wynosi 500000500000, czas wykonania:  0.0577178 sekund
```

Widzimy, że czas wykonania ponownie wzrósł.

Zauważmy teraz, że program `sumOfN2` wylicza sumę częściową ciągu arytmetycznego o różnicy $r = 1$ i wyrazie początkowym $1$. Korzystając z własności ciągu sumę tę możemy wyliczyć przy pomocy wyrażenia:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

In [8]:

```
def sumOfN3(n):
    start = time.time()

    theSum = n*(n+1)/2

    end = time.time()

    return theSum,end-start
```

In [9]:

```
for n in (10000,100000,1000000,10000000):
    print("Suma wynosi %d, czas wykonania: %10.7f sekund"%sumOfN3(n))
```

```
Suma wynosi 50005000, czas wykonania:  0.0000036 sekund
Suma wynosi 5000050000, czas wykonania:  0.0000019 sekund
Suma wynosi 500000500000, czas wykonania:  0.0000014 sekund
Suma wynosi 50000005000000, czas wykonania:  0.0000012 sekund
```

Analizując te wyniki dochodzimy do dwóch wniosków:

1. `sumOfN3` wykonuje się dużo szybciej niż `sumOfN2`
2. w przypadku `sumOfN3` czas wykonania funkcji nieznacznie zależy od $n$

## Notacja asymptotyczna

Aby sformalizować powyższe wnioski, potrzebujemy miary, która pozwoli na scharakteryzować czas wykonywania algorytmu w zależności od wielkości danych wejściowych, i niezależnie od typu komputera i użytego języka programowania.

Do scharakteryzowania czasu wykonania algorytmu ważne jest określenie liczby operacji i/lub kroków niezbędnych do jego zakończenia. Jeżeli na każdą z tych operacji potraktujemy jako podstawową jednostkę obliczeniową, wówczas możemy wyrazić czas wykonania algorytmu poprzez liczbę niezbędnych operacji.

Wybór podstawowej jednostki nie jest łatwy i zależy od tego, jak algorytm jest zaimplementowany. W przypadku powyższych funkcji dobrym kandydatem na jednostę podstawową jest liczba przypisań niezbędnych do wyliczenia sumy. I tak w przypadku funkcji `sum0fN2` mamy $T(n) = n + 1$ przypisań, natomiast w przypadku `sum0fN3` - tylko jedno. Można pójść krok dalej i stwierdzić, że dokładna liczba operacji nie jest ważna, a to, co się liczy, to dominująca składowa $T(n)$, bo dla dużych $n$ pozostałe składniki i tak są zaniedbywalne. Innymi słowy, interesuje nas tylko asymptotyczne tempo wzrostu czasu wykonywania algorytmów.

Do zapisu tego tempa służy tzw. notacja dużego O:

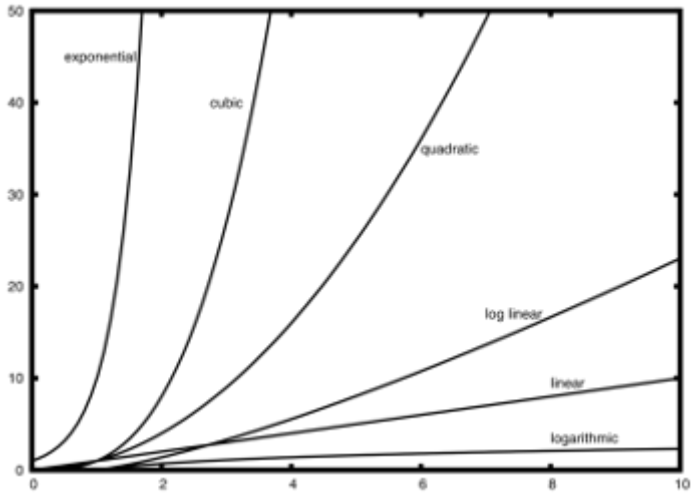
> Niech $f$ i $g$ będą ciągami liczb rzeczywistych. Piszemy $f(n) = O(g(n))$ wtedy, gdy istnieje stała dodatnia $C$ taka, że $|f(n)| \leq C|g(n)|$ dla dostatecznie dużych wartości $n$.

- służy do zapisu szybkości wzrostu
- nazywana jest ona czasami **złożonością teoretyczną** algorytmu
- definicja dopuszcza, aby nierówność nie była spełniona dla pewnej liczby małych wartości $n$
- w praktyce $f(n)$ oznacza ciąg, którym właśnie się zajmujemy (np. górne ograniczenie czasu działania algorytmu), a $g(n)$ jest prostym ciągiem o znanej szybkości wzrostu
- notacja nie podaje dokładnego czasu wykonania algorytmu
- pozwala odpowiedzieć na pytanie, jak ten czas będzie rósł z rozmiarem danych wejściowych

Przydatne własności:

- W hierarchii ciągów $1$ , $\log_2 n$,. . ., $\sqrt[4]{n}$, $\sqrt[3]{n}$, $\sqrt{n}$, $n$, $n \ln n$, $n\sqrt{n}$, $n^2$, $n^3$, . . ., $2^n$, $n!$, $n^n$ każdy z nich jest $O$ od wszystkich ciągów na prawo od niego
- Jeśli $f(n) = O(g(n))$ i $c$ jest stałą, to $c * f(n) = O(g(n))$
- Jeśli $f(n) = O(g(n))$ i $h(n) = O(g(n))$, to $f(n) + h(n) = O(g(n))$
- Jeśli $f(n) = O(a(n))$ i $g(n) = O(b(n))$, to $f(n) * g(n) = O(a(n) * b(n))$
- Jeśli $a(n) = O(b(n))$ i $b(n) = O(c(n))$, to $a(n) = O(c(n))$
- $O(a(n)) + O(b(n)) = O(\max\{|a(n)|, |b(n)|\})$
- $O(a(n)) * O(b(n)) = O(a(n) * b(n))$

W praktyce stwierdzenie, że algorytm jest klasy $O(n^3)$ oznacza, że dla danych wejściowych o wielkości $n$ (np. układ równań liniowych $n$ zmiennych) czas wykonania algorytmu jest proporcjonalny do $n^3$.




Zobaczmy, jak wypada porównanie czasów wykonania algorytmów różnych typów przy założeniu, że dla $n = 1$ każdy z nich wykonuje się $10^{-6}$ s:

| $n$ | 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|
| $O(n)$ | 0,00001s | 0,00002s | 0,00003s | 0,00004s | 0,00005s | 0,00006s |
| $O(n^2)$ | 0,0001s | 0,0004s | 0,0009s | 0,0016s | 0,0025s | 0,0036 |
| $O(n^3)$ | 0,001s | 0,008s | 0,027s | 0,064s | 0,125s | 0,216s |
| $O(2^n)$ | 0,001s | 1,048s | 17,9min | 12,7dni | 35,7lat | 366w |
| $O(3^n)$ | 0,059s | 58min | 6,5lat | 3855w | 227*$10^6$w | 1,3*$10^{13}$w |

## Przykład - anagramy

**Anagram** oznacza wyraz, wyrażenie lub całe zdanie powstałe przez przestawienie liter bądź sylab innego wyrazu lub zdania, wykorzystujące wszystkie litery (głoski bądź sylaby) materiału wyjściowego:

- kebab $\leftrightarrow$ babek
- Gregory House $\leftrightarrow$ Huge ego, sorry
- "Quid est veritas?" (*Co to jest prawda?*, Piłat) $\leftrightarrow$ "Vir est qui adest" (*Człowiek, który stoi przed tobą*, Jezus)

Testowanie, czy łańcuchy znaków są anagramami, to przykład zagadnienia, które można rozwiązać algorytmami o różnym tempie asymptotycznego wzrostu, a jednocześnie na tyle prostego, aby można było o tym opowiedzieć w ramach kursu ze wstępu do programowania.

### Rozwiązanie 1: "odhaczanie" liter

Jednym z możliwych rozwiązań naszego problemu jest sprawdzenie, czy litera z jednego łańcucha znajduje się w drugim. Jeżeli tak, "odhaczamy" ją i powtarzamy czynność dla pozostałych liter. Jeżeli po zakończeniu tej operacji wszystkie litery w drugim łańcuchu zostaną "odhaczone", łańcuchy muszą być anagramami. Odhaczanie możemy zrealizować poprzez zastąpienie odnalezionej litery wartością specjalną `None`.

In [10]:

```
def anagramSolution1(s1,s2):
    alist = list(s2) #zamień drugi łańcuch na listę

    pos1 = 0
    stillOK = True

    while pos1 < len(s1) and stillOK:
        pos2 = 0
        found = False
        while pos2 < len(alist) and not found:
            if s1[pos1] == alist[pos2]:    #sprawdzamy literę s1[pos1]
                found = True
            else:
                pos2 = pos2 + 1

        if found:
            alist[pos2] = None
        else:
            stillOK = False #przerwij, jeśli litery nie ma

        pos1 = pos1 + 1 #pozycja następnej litery w łańcuchu s1

    return stillOK

print(anagramSolution1('abcd','dcba'))
```

```
True
```

Dla każdego znaku z łańcucha $s1$ musimy wykonać iterację po maksymalnie $n$ elementach listy $s2$. Każda pozycja w liście $s2$ musi zostać odwiedzona raz w celu odhaczenia. Dlatego całkowita liczba wizyt elementów listy $s2$ jest sumą liczb naturalnych od $1$ do $n$:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

Dla dużych $n$ wyraz $\frac{1}{2}n^2$ będzie dominował nad $\frac{1}{2}n$. Dlatego algorytm jest klasy $O(n^2)$.

**Rozwiązanie 2: sortowanie i porównywanie**

Zauważmy, że jeżeli $s1$ i $s2$ są anagramami, muszą składać się z tych samych liter, występujących taką samą liczbę razy. Jeżeli więc posortujemy każdy z łańcuchów alfabetycznie od 'a' do 'z', powinniśmy otrzymać dwa takie same łańcuchy.

Do posortowania wykorzystamy metodę `sort` :

In [2]:

```
def anagramSolution2(s1,s2):
    alist1 = list(s1) #konieczne, żeby skorzystać z sort
    alist2 = list(s2)

    alist1.sort()
    alist2.sort()

    pos = 0
    matches = True

    while pos < len(s1) and matches:
        if alist1[pos]==alist2[pos]:
            pos = pos + 1
        else:
            matches = False

    return matches

print(anagramSolution2('ajcde','edcba'))
```

```
False
```

Na pierwszy rzut oka może się wydawać, że algorytm jest klasy $O(n)$, ponieważ wykonujemy tylko jedną iterację po elementach łańcuchów. Jednak wywołanie metody `sort` również "kosztuje", najczęściej $O(n^2)$ lub $O(n \log n)$. Dlatego czas wykonania będzie zdominowany przez operację sortowania.

**Rozwiązanie 3: algorytm siłowy (ang. *brute force*)**

Metoda siłowa rozwiązania jakiegoś zadania polega na wyczerpaniu wszystkich możliwości. Dla anagramów oznacza to wygenerowanie listy wszystkich możliwych łańcuchów ze znaków łańcucha $s1$ i sprawdzenie, czy $s2$ znajduje się na tej liście. Nie jest to jednak zalecane podejście, przynajmniej w tym przypadku. Zauważmy mianowicie, że dla ciągu znaków $s1$ o długości $n$ mamy $n$ wyborów pierwszego znaku, $(n-1)$ możliwości dla znaku na drugiej pozycji, $(n-2)$ na trzeciej pozycji itd. Musimy zatem wygenerować $n!$ łańcuchów znaków.

Dla ustalenia uwagi przyjmijmy, że $s1$ składa się z 20 znaków. Oznacza to konieczność wygenerowania

In [12]:

```
import math
math.factorial(20) #silnia
```

Out[12]:

```
2432902008176640000
```

łańcuchów znaków, a następnie odszukanie wśród nich ciągu $s2$. Widzieliśmy już wcześniej, ile czasu zajmuje algorytm klasy $O(n!)$, dlatego nie jest to polecane podejście do zagadnienia anagramów.

**Rozwiązanie 4: zliczaj i porównuj**

Jeżeli $s1$ i $s2$ są anagramami, będą miały tą samą liczbę wystąpień litery 'a', tą samą litery 'b' itd. Dlatego możemy zliczyć liczbę wystąpień poszczególnych znaków w łańcuchach i porównać te liczby ze sobą:

In [18]:

```
def anagramSolution4(s1,s2):
    c1 = [0]*26     #dla ułatwienia ograniczamy się do języka angielskiego
    c2 = [0]*26

    for i in range(len(s1)):
        pos = ord(s1[i])-ord('a') #pozycja znaku w alfabecie
        c1[pos] = c1[pos] + 1

    for i in range(len(s2)):
        pos = ord(s2[i])-ord('a')
        c2[pos] = c2[pos] + 1

    j = 0
    stillOK = True
    while j<26 and stillOK:
        if c1[j]==c2[j]:
            j = j + 1
        else:
            stillOK = False

    return stillOK

print(anagramSolution4('apple','pleap'))
```

```
True
```

Również w przypadku tej metody mamy do czynienia z iteracjami, jednak teraz żadna z nich nie jest zagnieżdżona. Dodając kroki konieczne do wykonania w tych iteracjach do siebie otrzymamy

$$T(n) = 2n + 26$$

Jest to zatem algorytm klasy $O(n)$, czyli najszybszy ze wszystkich prezentowanych. Zauważmy jednak, że lepszą wydajność uzyskaliśmy kosztem większego obciążenia pamięci (dwie dodatkowe listy $c1$ i $c2$). To sytuacja bardzo często spotykana w praktyce. Dlatego programując, nie raz staniemy przed koniecznością wyboru, który z zasobów (czas procesora czy pamięć) należy poświęcić.

## Struktury danych w Pythonie i ich wydajność

Złożone struktury danych dostępne w Pythonie oszczędzają programiście sporo pracy. Dobrze jest wiedzieć, jakie są zarówno ich mocne strony jak i ograniczenia, ponieważ pozwoli to w przyszłości pisać lepsze programy.

**Listy**

Przyjrzyjmy się najpierw różnym sposobom generowania list:

In [13]:

```
def test1(): #łączenie list
    l = []
    for i in range(1000):
        l = l + [i]

def test2(): #dodawanie elementu
    l = []
    for i in range(1000):
        l.append(i)

def test3(): #list comprehension
    l = [i for i in range(1000)]

def test4(): #z zakresu
    l = list(range(1000))
```

In [14]:

```
%%timeit
test1()
```

```
The slowest run took 4.51 times longer than the fastest. This could
mean that an intermediate result is being cached.
1000 loops, best of 3: 1.34 ms per loop
```

In [15]:

```
%%timeit
test2()
```

```
The slowest run took 7.11 times longer than the fastest. This could
mean that an intermediate result is being cached.
10000 loops, best of 3: 81.3 µs per loop
```

In [16]:

```
%%timeit
test3()
```

```
The slowest run took 5.44 times longer than the fastest. This could
mean that an intermediate result is being cached.
10000 loops, best of 3: 33.3 µs per loop
```

In [17]:

```
%%timeit
test4()
```

```
The slowest run took 4.18 times longer than the fastest. This could
mean that an intermediate result is being cached.
10000 loops, best of 3: 14.1 µs per loop
```

Z powyższego eksperymentu wynika, że najszybszą metodą jest tworzenie listy z zakresu, natomiast najdłużej trwa łączenie list.

Jeśli chodzi o różne operacje na listach, ich złożoność jest następująca:

| Operacja | Złożoność |
|---:|---:|
| index [] | O(1) |
| zmiana wartości elementu | O(1) |
| append | O(1) |
| pop() | O(1) |
| pop(i) | O(n) |
| insert(i,item) | O(n) |
| del | O(n) |
| iterowanie | O(n) |
| zawiera (in) | O(n) |
| odczyt wycinka [x:y] | O(k) |
| usuwanie wycinka | O(n) |
| przypisanie wycinka | O(n+k) |
| reverse | O(n) |
| łączenie | O(k) |
| sort | O(n log n) |
| iloczyn | O(nk) |

Dla przykładu zbadajmy wydajność metody `pop` :

In [1]:

```
import timeit
popzero = timeit.Timer("x.pop(0)",
                       "from __main__ import x")
popend = timeit.Timer("x.pop()",
                      "from __main__ import x")
```

In [2]:

```
x = list(range(2000000))
popzero.timeit(number=1000)
```

Out[2]:

```
1.8516352830001779
```

In [3]:

```
x = list(range(2000000))
popend.timeit(number=1000)
```

Out[3]:

```
0.00010356900020269677
```

Eksperyment ten można powtórzyć dla różnych $n$:

In [15]:

```
popzero = timeit.Timer("x.pop(0)",
                       "from __main__ import x")
popend = timeit.Timer("x.pop()",
                      "from __main__ import x")

pi = []
pe = []

for i in range(1000000,10000001,1000000):
    x = list(range(i))
    pt = popend.timeit(number=1000)
    pe.append(pt)
    x = list(range(i))
    pz = popzero.timeit(number=1000)
    pi.append(pz)
```

In [16]:

```
%matplotlib inline
```

In [17]:

```
import matplotlib
import matplotlib.pyplot as plt
```

In [22]:

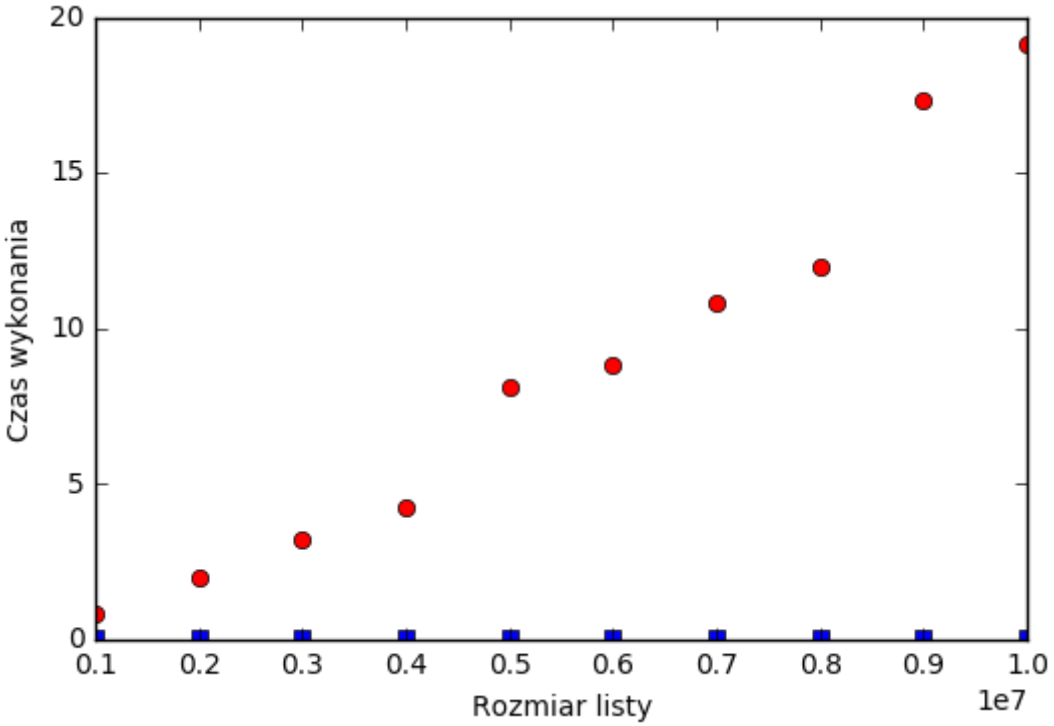
```
sizes = list(range(1000000,10000001,1000000))
```

In [26]:

```
plt.xlabel("Rozmiar listy")
plt.ylabel("Czas wykonania")
plt.plot(sizes,pi,'ro',label="pop(0)")
plt.plot(sizes,pe,'bs',label="pop()")
```

Out[26]:

```
[<matplotlib.lines.Line2D at 0x7f8a4a370400>]
```



### Słowniki

Słowniki to drugi główny typ danych w Pythonie. Wydajności wybranych operacji są następujące:

| Operacja | Złożoność |
|---|---|
| copy | O(n) |
| get item | O(1) |
| set item | O(1) |
| delete item | O(1) |
| contains (in) | O(1) |
| iteration | O(n) |