

ALGORYTMY I STRUKTURY DANYCH

WYKŁAD 1/2 - Algorytmy i ich analiza

JANUSZ SZWABINSKI

Plan wykłada

1. Sprawy administracyjne
2. Algorytm - formalna definicja
3. Analiza doswiadczenia algorytmów
4. Analiza asymptotyczna

1. SPRAWY ADMINISTRACYJNE

1.1. Kontakt i godziny konsultacji

<http://prac.im.pwr.wroc.pl/~szwabin/>

1.2. Tresci programowane

- Algorytmy i ich analiza
- Rekurencja
- Podstawowe struktury danych
- Drzewa i algorytmy ich przetwarzania
- Sortowanie i wyszukiwanie
- Grafy i algorytmy grafowe

1.3 Literatura

- M.T. Goodrich, R. Tamassia, M.H. Goldwasser
„Data Structures and Algorithms in Python”
- T.H. Cormen, Ch.E. Leiserson, R.L. Rivest
„Wprowadzenie do algorytmów”
- B. Miller, D. Ranum
„Problem solving with algorithms and data structures using Python”

1.4. Warsztat

- Python 3.X + pakiety doCommandLine
- dowolny edytor / IDE z rozszerzeniem dla Pythona
- ewentualnie Jupyter

1.5 Zasady ocenienia

- listy zadań na laboratoriach
- dwa kolokwia (~7 i ~14 punktów)

$$\text{ocena} = \frac{\text{ocena}_{\text{lab}} + \text{ocena}_{\text{wkp}}}{2}$$

Punkty (na kolokwiach i laboratoriach) preliczane są na oceny według tabeli:

Liczba punktów	Ocena
$X < 65$	2.0
$65 \leq X < 70$	3.0
$70 \leq X < 75$	3.5
$75 \leq X < 85$	4.0
$85 \leq X < 95$	4.5
$95 \leq X$	5.0

2. ALGORYTHM

Def.

Algorytm to skonczone ciąg jasno zdefiniowanych czynności, kierujących do wykonania pewnego zadania zadani

⇒ przepis na rozwiązywanie problemów lub określanie jakiegoś celu

2.1 METODY ZAPISU

Zatem myśl, że naszym zadaniem jest obliczenie wartości funkcji

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

przy założeniu $f(0) = 0$

OPIS STÓWNY

1. Dla liczb ujemnych $|x| = -x$, więc $f(x) = -1$
2. Dla liczb dodatnich $|x| = x$, więc $f(x) = 1$
3. Dla $x = 0$ z założenia $f(0) = 0$

Można to wyrazić wzorem

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

LISTA KROKÓW

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

SCHEMAT BLOKOWY



Blok startowy lub końcowy



Operacja we/wy



Blok warunkowy



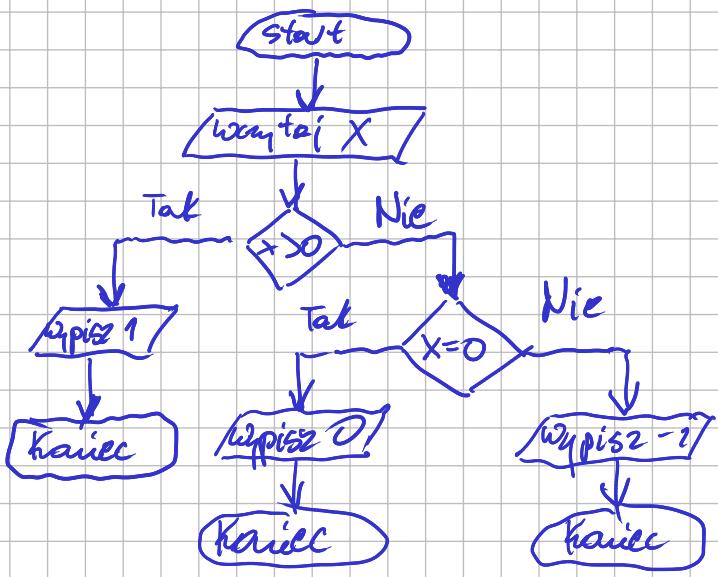
Blok wykonywany



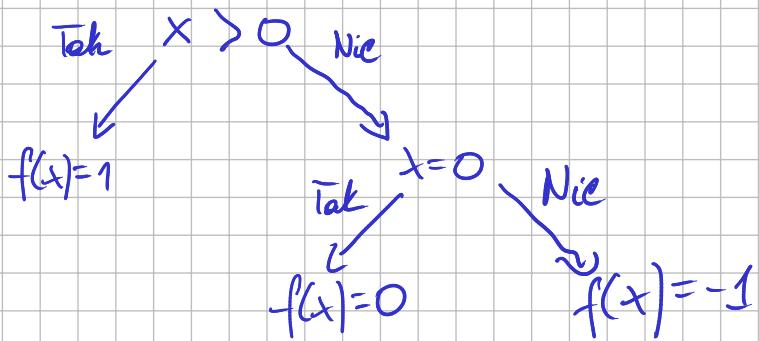
Zdefiniujemy
wcześniej proces



struktura Tycza



DRZEWKO ALGORYTMU



Program komputerowy to nic innego jak algorytm zapisany w wybranym języku programowania

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

2.2. DLACZEGO WARTO ZAJMOWAĆ SIĘ ALGORYTHAMI?

- cały obszar IT opiera się na algorytmach
- niektóre są paradygmatowe
 - Euklides z Aleksandrii (IV w p.n.e) → NWD
 - algorytm RSA do szyfrowania z kluczem publicznym
- pozwalają formułować zadania, które inaczej byłyby nieformalizowane → problem koniunkcyjny
- profesjonalni programiści używają algorytmów
- stymulują intelektualność
- generują rynek
 - GOOGLE

3. ANALIZA EKSPERYMENALNA ALGORYTMÓW

Analiza algorytmów zajmuje się ich porównywaniem pod względem niektórych obliczeniowych niebezpieczeństw do uzyskania rozwiązania

Möliwe kryteria porównania:

- częstotliwość kodów
- „rozdrobniość” (up. RAM)
- liczba operacji niebezpiecznych do wykonania
- czas wykonania

3.1 MOTYWACJA

Pozwajmy następującą funkcję:

```
. def foo(tom):  
    fred = 0  
    for bill in range(1,tom+1):  
        barney = bill  
        fred = fred + barney  
  
    return fred  
  
print(foo(10)) → 55
```

Co robi ta funkcja?

→ Sumuje liczby od 1 do tom!

Möliwy zapisać ją w bardziej częstotliwy sposób:

```
def sumOfN(n):  
    theSum = 0  
    for i in range(1,n+1):  
        theSum = theSum + i  
  
    return theSum  
  
print(sumOfN(10)) → 55
```

Obie funkcje robią to samo, jednak dobrą nazwę zmieniły w tej darskiej powodują, że od razu rozumieją jej przeznaczenie.

Zatem teraz, że interesuje nas czas wykonania tej funkcji dla różnych n. Zmodyfikujemy ostatnią definicję tak, aby mieliśmy ten czas:

```

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 kolejnych wywołań funkcji dla ustalonego N będące następujące:

```

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

```

```

Suma wynosi 50005000, czas wykonania: 0.0004852 sekund
Suma wynosi 50005000, czas wykonania: 0.0005150 sekund
Suma wynosi 50005000, czas wykonania: 0.0005434 sekund
Suma wynosi 50005000, czas wykonania: 0.0005865 sekund
Suma wynosi 50005000, czas wykonania: 0.0006051 sekund

```

Czas wykonywania się nieznacznie (bo obciążenie komputera zmienia się w czasie), ale nadal wielkość porasta się ten sam!

A teraz zbadajmy czas wykonyania dla różnych N:

```

for n in (10000,100000,1000000,10000000):
    print("Suma wynosi %d, czas wykonania: %10.7f sekund"%sumOfN2(n))

```

```

Suma wynosi 50005000, czas wykonania: 0.0016453 sekund
Suma wynosi 5000050000, czas wykonania: 0.0106499 sekund
Suma wynosi 500000500000, czas wykonania: 0.0573034 sekund
Suma wynosi 50000005000000, czas wykonania: 0.4355214 sekund

```

Czas wykonywania rośnie z n

Czy powyższy kod można poprawić? Okazuje się, że tak! Funkcja sumOfN2 wylicza sumę częściowego ciągu arytmetycznego o różnicę r=1 i wyrazie początkowym równym 1, korzystając z wartości ciągu, sumę modułu wyliczyc jakaś

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

Rozważmy funkcję implementującą powyższy wzór:

```
import time

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

    theSum = n*(n+1)/2

    end = time.time()

    return theSum, end-start
```

Wykonanie jej kolejnych wezwisków będzie następujący:

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

Suma wynosi 50005000, czas wykonania: 0.0000010 sekund
Suma wynosi 5000050000, czas wykonania: 0.0000005 sekund
Suma wynosi 500000500000, czas wykonania: 0.0000002 sekund
Suma wynosi 50000005000000, czas wykonania: 0.0000005 sekund

Analiza tych wezwisków prowadzi do dwóch wniosków:

- `sumOfN3` jest dużo szybsza od `sumOfN2`
- czas wykonania `sumOfN3` praktycznie nie zależy od N

→ dużo lepszy algorytm
→ cena: znajomość matematyczna :)

3.2 ANALIZA EKSPERYMENTALNA

Powyższy przykład może stanowić punkt wyjścia do przeprowadzenia bardziej złożonej analizy algorytmów:

1. Obserwacja pewnych właściwości świata naturalnego, np. czasu pracy programu na komputerze.

2. Opracowanie hipotezy (modelu), który jest zgodny z obserwacjami.
3. Przewidywanie zdarzeń za pomocą opracowanej hipotezy, np. czasu pracy programu dla danych wejściowych.
4. Weryfikacja przewidywań poprzez dalsze obserwacje,
5. Wielokrotna powtarzanie weryfikacji do momentu, kiedy hipotezy i obserwacje są zgodne.

3.3. STUDIUM PRZYPADKU - 3SUM

3SUM to znany problem w teorii złożoności obliczeniowej:

Czy dla danego zbioru N liczb całkowitych istnieją elementy a, b i c z tego zbioru takie, że

$$a + b + c = 0.$$

Jesli tak, to ile ich jest?

Dla zaинтересowanych — jeden z nieważnych znanych problemów w informatyce:

Czy istnieje algorytm rozwiązujący zadanie 3SUM w czasie $O(n^{2-\epsilon})$ dla pewnego $\epsilon > 0$?

Nas będzie jednak obecnie interesować rozwiązanie tego problemu METODĄ NAIWNĄ — będziemy sprawdzać wszystkie możliwe triplety:

```
import random
tab = []
for i in range(10):
    tab.append(random.randint(-50,50))
```

kod przygotowujący listę losujących elementów

```

count = 0
for i in range(len(tab)):
    for j in range(i+1, len(tab)):
        for k in range(j+1, len(tab)):
            if (tab[i]+tab[j]+tab[k]==0):
                count = count + 1
                print(tab[i],tab[j],tab[k])
print(f"W sumie: {count}")

```

hod sprawdzającej
triplety na powyższej
listie

W oparciu o powyższy algorytm możemy napisać program:

%%writefile 3sum_3.py → "magic command" w Jupyter
 """3SUM problem solved with brute-force algorithm"""

```

def countTriplets(tab):

    count = 0
    triplets = []

    for i in range(len(tab)):
        for j in range(i+1, len(tab)):
            for k in range(j+1, len(tab)):
                if (tab[i]+tab[j]+tab[k]==0):
                    count = count + 1
                    triplets.append((tab[i],tab[j],tab[k]))
    return triplets

if __name__ == "__main__":
    import sys
    import time
    import random

    n = int(sys.argv[1])
    tab = []
    for i in range(n):
        tab.append(random.randint(-50,50))

    t = time.process_time() #czas start
    triplets = countTriplets(tab)
    t = time.process_time() - t #ile trwało?
    print("%d %f"%(n,t))

```

Rozważamy listy jako
parametry wejściowe

Sprawdzamy jego działanie:

```
python3 3sum_3.py 10
```

10 0.000058

for N in 10 20 50 100 200 500 1000; do python3 3sum_3.py \$N; done

10 0.000031
20 0.000163
50 0.002183
100 0.017513
200 0.117994
500 1.706200
1000 13.871542

to jest pętla w BASH,
często popularnej polecenie pod Linuxem

Możemy również napisać wyniki do pliku:

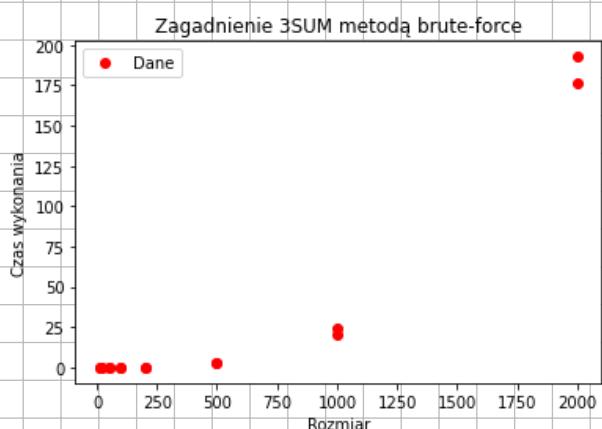
```
for N in 10 20 50 100 200 500 1000 2000; do python3 3sum_3.py $N >> wyniki.dat; done
```

WYKRES CZASU TRWANIA

```
import matplotlib.pyplot as plt
import numpy as np

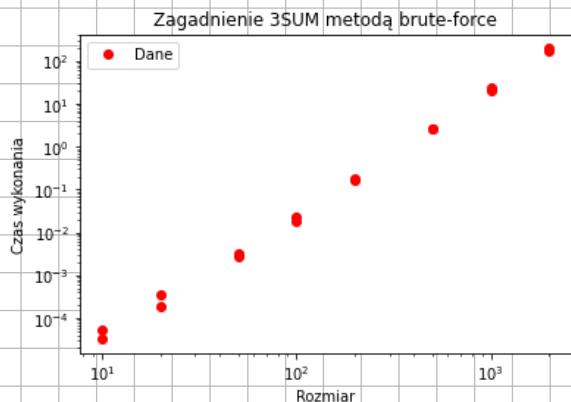
x,y = np.loadtxt("wyniki.dat", unpack=True)

plt.plot(x,y,'ro',label="Dane")
plt.xlabel("Rozmiar")
plt.ylabel("Czas wykonania")
plt.legend(loc='upper left')
plt.title("Zagadnienie 3SUM metodą brute-force")
plt.show()
```



Czasu bardziej czytelny jest wykres w skali logarytmicznej:

```
plt.loglog(x,y,'ro',label="Dane")
plt.xlabel("Rozmiar")
plt.ylabel("Czas wykonania")
plt.legend(loc='upper left')
plt.title("Zagadnienie 3SUM metodą brute-force")
plt.show()
```



HIPOTEZA

Spróbujmy teraz przybliżyć dane jakieś "proste" funkcje matematyczne. Ponieważ na wykresie w skali logarytmicznej dane ukladają się w linię prostą,

spodziewamy się zależności potęgowej:

$$y = \alpha x^b$$

Rzeczywiście, po obustronnym zlogarytmowaniu mamy

$$\log y = b \log x + \log \alpha$$

czyli funkcję liniową. Ponadto nasz algorytm wykorzystuje potęgowe rańniedzielenie, więc przypuszczaćmy, że czas będzie proporcjonalny do N^3 :

```
from scipy.optimize import curve_fit

def func(x, a, b):
    return a*x**3 + b

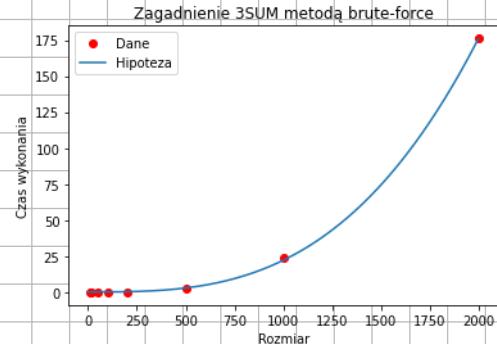
popt, pcov = curve_fit(func, x, y)

print("a = %s , b = %s" % (popt[0], popt[1]))
```

a. = 2.2015328416884753e-08 , b = 0.20601171867247256

x2 = np.arange(1,2000)

```
plt.plot(x,y,'ro',label="Dane")
plt.plot(x2, func(x2,*popt), label="Hipoteza")
plt.xlabel("Rozmiar")
plt.ylabel("Czas wykonania")
plt.legend(loc='upper left')
plt.title("Zagadnienie 3SUM metodą brute-force")
plt.show()
```



PRZEWIDYWANIE

Zatem, że interesuje nas, ile będzie wykonywać się program dla $N = 1200$:

func(1200,*popt)

37.240412886967256

WERYFIKACJA

python3 3sum_3.py 1200

1200 36.325262

clase empiryczne potwierdza naszą hipotezę!!!

3.4. OGRANIČENIA ANALIZY EKSPERYMENALNEJ

- ekspermentalne porównanie dwóch programów wymaga takich samych środowisk dla każdego z nich
- doświadczenie może być wykonyane na ograniczonej liczbie danych wejściowych (problem danych jest ważny)
- program musi być ukonczone, aby poddać je analizie!!!

4. ANALIZA ASYMPTOTYCZNA

4.1 WYJŚCIE POZA ANALIZĘ DOŚWIADCZALNĄ

Chcemy metody, które

- prowadzą ocenę względnego wydajnością dwoch algorytmów, niezależnie od sprzętu i oprogramowania
- chująca raczej na opisie algorytmu, a nie jego implementacji
- bierze pod uwagę wszystkie możliwe dane wejściowe

ZLICZANIE OPERACJI PODSTAWOWYCH

Analiza opisu, pseudokodu lub kodu algorytmu.

Do operacji podstawowych należą:

- przypisanie wartości do zmiennych
- operacje arytmetyczne
- porównanie dwóch wartości
- dostęp do pojętycznego elementu listy
- wywołanie funkcji (z wyjątkiem operacji wewnętrznych funkcji)
- zwrocenie wartości przez funkcję

Formalnie, operacje podstawowe

- odpowiadają instrukcjom niskiego poziomu ze statusem czasem wykonania
- części z nich tłumaczona jest na więcej niż jedną instrukcję procesora

Zamiast liczyć czas ich wykonania, będziemy zliczać ich liczbę i traktować ją jako miarę czasu wykonania algorytmu!

LICZBA OPERACJI W FUNKCJI DANYCH WEJŚCIOWYCH

Każdemu algorytmowi przyporządkowanej funkcji

$$f(u) \quad \text{zmienna danych wejściowych}$$

która charakteryzuje liczbę operacji w nizu różnych

PRZYPADKI PESYMISTYCZNE

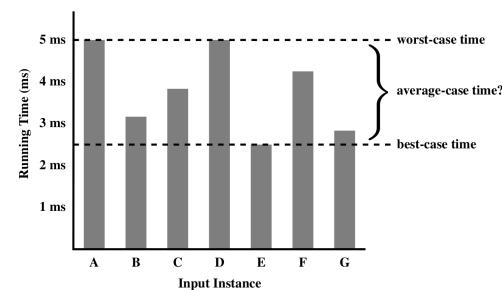


Figure 3.2: The difference between best-case and worst-case time. Each bar represents the running time of some algorithm on a different possible input.

Dla niektórych danych wejściowych algorytm działaż szybciej niż dla innych

→ teoretycznie, powinienśmy wyznaczyć liczbę operacji jako średniz ze wszystkich możliwych danych o tym samym rozmiarze

→ Tatujej skupić się na analizie najgorszych przypadków

→ wymaga jedynie zidentyfikowania danych, dla których algorytm będzie sprawdzał się najgorzej

4.2 ANALIZA ASYMPTOTYCZNA

Zauważenie: Nie liczy się tak naprawdę dokładna liczba operacji podstawowych, a tylko jej dominująca składowa, bo dla dużej liczby danych porastały składowki i tak są nieodbywalne.

Jedynie stwierdzić, interesuje nas tylko asymptotyczne tempo wzrostu czasu wykonania algorytmu

4.3. NOTACJA DUŻEGO O

DEFINICJA

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 .

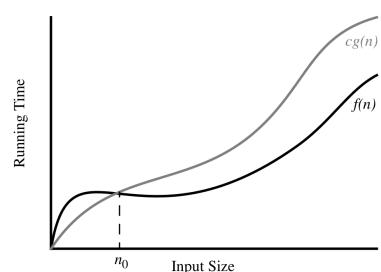


Figure 3.5: Illustrating the "big-Oh" notation. The function $f(n)$ is $O(g(n))$, since $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

- służy do zapisu szybkości wzrostu
- tzw. złożoność teoretyczna algorytmu

- w praktyce:

$f(n)$ - ciąg, którymu się zapisujemy (np. górne ograniczenie czasu działania algorytmu)

$g(n)$ - prosty ciąg o znanej szybkości wzrostu

- notacja **nie podaje** dokładnego czasu wykonania algorytmu
- pozwala odpowiedzieć na pytanie, jak ten czas będzie się zmieniał z formułacją danych wejściowych

Pozystawne właściwości:

1. w hierarchii ciągów

$$1, \log_2 n, \dots, \sqrt[3]{n}, \sqrt{n}, n, n^2, \dots, 2^n, n^{\frac{1}{n}}, n^{-1}$$

Każdy jest O od wszystkich ciągów na prawo od niego

2. $f(n) = O(g(n)) \Rightarrow c f(n) = O(g(n))$
 c jest stały

3. $f(n) = O(g(n))$
 $h(n) = O(g(n))$

$\Rightarrow f(n) + h(n) = O(g(n))$

4. $f(n) = O(a(n))$
 $g(n) = O(b(n))$

$\Rightarrow f(n) \cdot g(n) = O(a(n) \cdot b(n))$

5. $a(n) = O(b(n))$
 $b(n) = O(c(n))$

$\Rightarrow a(n) = O(c(n))$

6. $O(a(n)) + O(b(n)) = O(\max\{|a(n)|, |b(n)|\})$

7. $O(a(n)) \cdot O(b(n)) = O(a(n) \cdot b(n))$

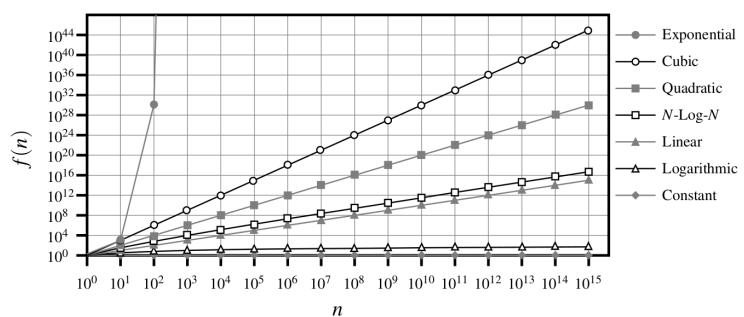


Figure 3.4: Growth rates for the seven fundamental functions used in algorithm analysis. We use base $a = 2$ for the exponential function. The functions are plotted on a log-log chart, to compare the growth rates primarily as slopes. Even so, the exponential function grows too fast to display all its values on the chart.

Zaktadaję, że dla $n = 1$ wykonanie każdej operacji trwa
zajmuje 10^{-6} s, w poniższej tabeli zestawione są
dla tempa wzrostu

n	10	20	30	40	50	60
$O(n)$	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
$O(n^2)$	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0025 s	0,0036 s
$O(n^3)$	0,001 s	0,008 s	0,027 s	0,064 s	0,125 s	0,216 s
$O(2^n)$	0,001 s	1,048 s	17,9 min	12,7 dni	35,7 lat	366 w
$O(3^n)$	0,059 s	58 min	6,5 lat	3855 w	$229 \cdot 10^6$ w	$1,3 \cdot 10^3$ w
$O(n!)$	3,6 s	771 w	$8,4 \cdot 10^{16}$ w	$2,6 \cdot 10^{32}$ w	$9,6 \cdot 10^{44}$ w	$2,6 \cdot 10^{66}$ w

w - wiek (100 lat)

4.4. KOSZT WYKONANIA POJEDYNCZEJ OPERACJI

- pierwsze komputery dostarczane były z instrukcjami zawierającymi dodatkowe czasy wykonania każdej operacji
- obecnie możemy stacować go **empirical**
 - np. minimum czas biliona dodawania i wyjmowania czas pojedynczego dodawania
- w praktyce zakładamy, że wszystkie operacje podstawowe zajmują pewien stały czas wykonania.

4.5 CALKOWITY CZAS PRACY

$$\sum (\text{koszt operacji}) \times (\text{częstotliwość wykonywania})$$

Niech:

```
import random  
lista = [random.randint(-10,10) for i in range(10)]
```

Rozważmy program

```
count = 0  
for i in lista:  
    if i==0:  
        count = count + 1  
  
print(count)
```

Częstotliwość wykonywania poszczególnych operacji

Operacja	Liczba wystąpień
przypisanie	1
poznanie	n
dostęp do listy	n
incrementacja i	n
incrementacja count	$\leq n$

- liczenie wszystkich operacji może być zasadne
- wybór operacji dominującej
 - najwięcej kosztuje i najczęściej występuje

Kolejny przykład:

```
#2SUM  
count = 0  
for i in range(len(lista)):  
    for j in range(i+1, len(lista)):  
        if lista[i]+lista[j]==0:  
            count = count + 1
```

Operacja dominująca	Liczba wystąpień
dostęp do listy	$n(n-1)$

czy każdy wieci, dlaczego?

Wartość i	Wartość j	Liczba dostępów do listy
0	1, 2, ..., n-1	$2(n-1)$
1	2, 3, ..., n-1	$2(n-2)$
2	3, 4, ..., n-1	$2(n-3)$
⋮	⋮	⋮
n-2	n-1	2
n-1	None	None

Sumując liczby operacji, otrzymamy

$$2(n-1 + n-2 + n-3 + \dots + 1) = 2\left(n \cdot n - \sum_{i=1}^n i\right) = 2n^2 - 2 \frac{n(n+1)}{2} \\ = n(n-1)$$

4.6 WYBRANE KLASY ALGORYTMÓW

4.6.1 Złożoność LOGARYTMICZNA

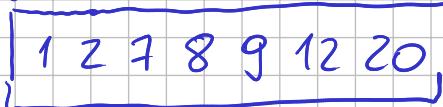
$$f(n) = O(\log(n))$$

- typowa dla algorytmów, w których problem przedstawiony dla danych o rozmiarze n da się sprowadzić do problemu z danymi o rozmiarze n/2
- klasa P problemów (tzn. problemy Tatwe, czas najwyżej wielomianowy)

PRZYKŁAD - WYSZUKIWANIE BINARNE

Dany jest posortowany (!) zbiór danych A oraz pewien element item. Chcemy odpowiedzieć na pytanie, czy item znajduje się w A?

item = 9



```
def binary_search(a,x,lo=0,hi=None):
    if hi is None:
        hi = len(a)
    while lo < hi:
        mid = (lo+hi)//2
        midval = a[mid]
        if midval == x:
            return mid
        elif midval < x:
            lo = mid + 1
        else:
            hi = mid
    return -1
```

A = [1,2,7,8,9,12,20]
binary_search(A,9)

↳ 4

binary_search(A,90)

↳ -1

Dla danej listy wejściowej algorytm wymaga jednego sprawdzenia elementu szukowanego, a następnie przesuwa je jedną z połowyk. Zatem

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

$$T(1) = 1$$

zauważmy, że $n = 2^m$ (tzn. $m = \log_2 n$). Wówczas

$$T(2^m) \leq T\left(\frac{2^m}{2}\right) + 1 \leq T\left(\frac{2^m}{4}\right) + 1 + 1 \leq T\left(\frac{2^m}{8}\right) + 1 + 1 + 1 \\ \leq \dots \leq T\left(\frac{2^m}{2^m}\right) + m = T(1) + m$$

czyli $T(n) \leq 1 + \log_2 n$

4.6.2 Złożoność liniowa

$$f(n) = O(n)$$

- typowa dla algorytmów, w których dla każdego elementu danych wejściowych wymagana jest pewna stała liczba operacji
- klasa P problemów łatwych

```
def linear_search(a,x):  
    for i in a:  
        if i == x:  
            return a.index(i)  
    return -1
```

4.6.3 Złożoność liniowo-logarytmiczna

$$f(n) = O(n \log n)$$

- typowa dla algorytmów, w których problem postawiony dla danych o rozmiarze n daje się sprawdzić w liniowej liczbie operacji do dwóch problemów o rozmiarach $n/2$
- klasa P problemów łatwych
- przykład: sortowanie przez scalanie

4.6.4 Złożoność kwadratowa

$$f(n) = O(n^2)$$

- dla każdej pary elementów wejściowych trzeba wykonać stałą liczbę operacji podstawowych
- klasa P problemów łatwych
- przykład: 2SUM metodą siłownią

4.6.5 Złożoność wielomianowa

$$f(n) = O(n^x)$$

- dla każdej krotki elementów wejściowych wykonywanie jest stała liczba operacji podstawowych
- klasa P
- przykłady :
 - 3SUM metodą siłowej
 - eliminacja Gaussa

4.6.6 Złożoność WYKADNICZA

$$f(n) = O(x^n)$$

- dla każdego podzbioru danych wejściowych należy wykonać stałą liczbę działań.
- klasa NP
- przykład - wieża z Hanoi

4.6.7 Złożoność SILNIE WYKADNICZA

$$f(n) = O(n!)$$

- stała liczba działań wykonywana dla każdej permutacji danych wejściowych
- klasa NP
- przykład - problem kominowojazera metodą siłową