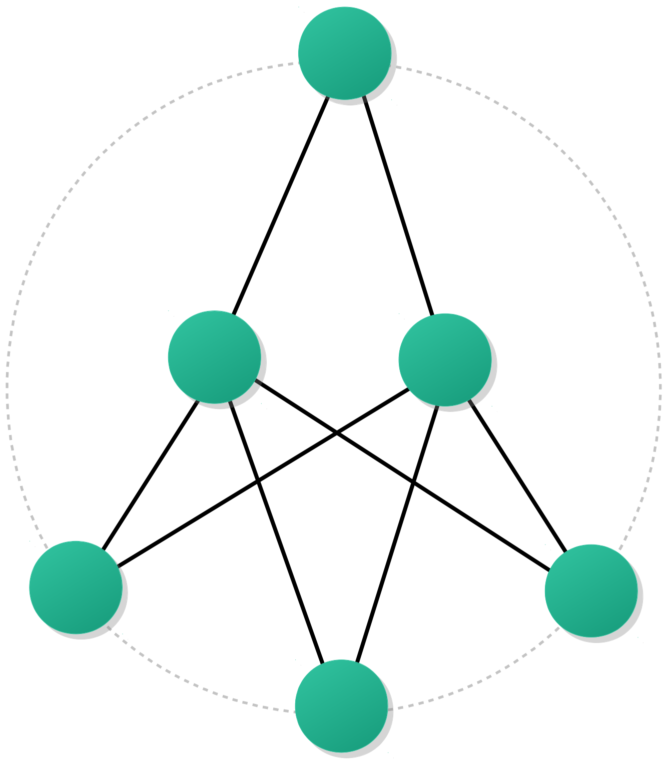
Wprowadzenie do algorytmów i struktur danych w Pythonie.

Część pierwsza:
Notacja asymptotyczna



Michał Wiśniewski
12-13 październik 2021

Materiały uzupełniające oraz przyjęte konwencje.

Odnośniki do materiałów wykorzystanych podczas warsztatu.

Prezentacja oraz kod źródłowy ćwiczeń.
github.com/Michael-Wisniewski/algorithmic-workshop

Pakiet narzędzi do profilowania kodu.
pypi.org/project/algo-profiler

Dokumentacja pakietu algo-profiler.
aroundpython.com/2021/06/13/tool-for-writing-algorithms

Ścieżka do pliku zawierającego kod źródłowy.

Ścieżka do skryptu, który należy uruchomić aby otrzymać przedstawiony pomiar.

Adnotacja warta zapamiętania.

Czego nauczę się podczas warsztatu?

- Rozwiązywać szersze spektrum problemów.
- Profilować i udoskonalać istniejące rozwiązania.
- Projektować bardziej niezawodne systemy.

Czym jest algorytm?

Algorytm - ciąg jednoznacznie zdefiniowanych czynności, przekształcających pewne dane wejściowe do pewnych danych wyjściowych, w celu rozwiązania określonego problemu.

Dwie kluczowe cechy algorytmu:

- zwraca poprawny wynik
- efektywnie wykorzystuje zasoby obliczeniowe

Podział algorytmów ze względu na “*poprawności wyniku*”.

dokładne – zwracają optymalne rozwiązanie (sortowanie listy, mnożenie macieży)

heurystyczne - nie gwarantują znalezienia optymalnego rozwiązania. Zwracane przez niego rozwiązanie może być dowolnie odległe od optimum. Służą do znajdowania rozwiązań problemów dla których klasyczne metody są zbyt wolne lub nie są w stanie osiągnąć optymalnego wyniku (prognozowanie pogody, problem komiwojażera).

aproksymacyjne – tak jak algorytmy heurystyczne nie gwarantują znalezienia optymalnego rozwiązania. Zwracane rozwiązanie różni się znanym czynnikiem od optymalnej miary ilościowej rozwiązania (określenie czy liczba jest pierwsza ze wskaźnikiem błędu wynoszącym np. 2^50).

Przykładowe metryki “*efektywnego wykorzystywania zasobów obliczeniowych*”.

	Miara wydajnośći	Przykładowe zwiększenie efektywności
czas	**Zwrócenie wyniku w akceptowalnym przedziale czasowym.**	Posortowanie danych przed wielokrotnym przeszukaniem zbioru dancyh.
pamięć	Zużycie minimalnej ilości pamięci podczas obliczeń.	Wykonywanie operacji zamian “na miejscu” podczas sortowania.
pamięć	Wykorzystanie całej dostępnej pamięci, w celu przyspieszenia działania algorytmu.	Zmiana algorytmu sortowania opierającego się na porównywaniu kluczy, na algorytm sortowania przez zliczanie.
przestrzeń dyskowa	Minimalna wielkość pliku.	Zastosowanie algorytmu o wyższym stopniu bezstratnej kompresji danych.
przepustowości sieci	Optymalne wykorzystanie przepustowości sieci.	Zastosowanie algorytmu dopasowującego szybkości transmisji danych do rejestrowanego poziomu liczby utraconych pakietów.
wydajność energetyczna	Minimalne zużycie energii elektrycznej potrzebnej do wykonania obliczeń.	Zwiększenie stopnia zwektoryzowania obliczeń wykonywanych na GPU.
stabilność systemu	**Równomierne obciążenie węzłów klastra sieciowego.**	**Zaimplementowanie algorytmu równoważenia obciążenia, który dostosowuje ilość wysyłanych zapytań dla każdego węzła.**

- W wielu przypadkach obliczenie przybliżonego wyniku jest jedynym akceptowalnym rozwiązaniem.
- Czas wykonania jest tylko jedną z wielu metryk wydajności.

Metodologia pomiaru czasu.

Czynniki wpływające na czas wykonania algorytmu:

- konfiguracja sprzętowa
- język programowania
- wersja kompilatora / interpretera
- rodzaj przetwarzanych danych
- …
- efektywna implementacja
- system operacyjny
- wersja sterowników
- ...

Problemy związane z podawaniem dokładnych wartości czasu wykonania:

- **niemiarodajne** - “*Mój algorytm wykonał się dwa tysiące razy szybszej na AMD Threadripper, niż twój na Commodore 64*”.
- **nieintuicyjne** - “*Ten algorytm potrzebuje czterech godzin do posortowania dwóch milionów rekordów*”.

Notacja Big-O

Czym jest – Funkcja z notacji asymptotycznej, która w przybliżony sposób opisuje w jaki sposób, wraz ze wsrotem wielkości dancych wejściowych, rośnie zużycie zasobów.

Co określa – Skalowalność algorytmu.

Jak znaleźć – Znajdujemy dokładną funkcję która sumuje czasy wykonania wszystkich operacji, a następnie pomijamy składniki niskiego rzędu oraz stałe czynniki. Uzależniamy funkcję jedynie od jednej zmiennej, będącej liczbą naturalną, która determinuje wielkość danych wejściowych.

* Dopuszczamy się pewnych uproszczeń.

Definicja

f(n) = O(g(n)), jeżeli dla odpowiednio dużego n, f(n) różni się od g(n) o pewien stały współczynnik - f(n) = constant * g(n).

$g(n) = 10n^3 + 5n + 100$
$f(n) = O(g(n)) = O(10n^3 + 5n + 100)$

n	$10n^3$	5n	100	suma
1	10	5	100	115
10	10000	50	100	10150
100	100000	500	100	100600
...	...	...	...	...
10000	100000000	50000	100	100050100

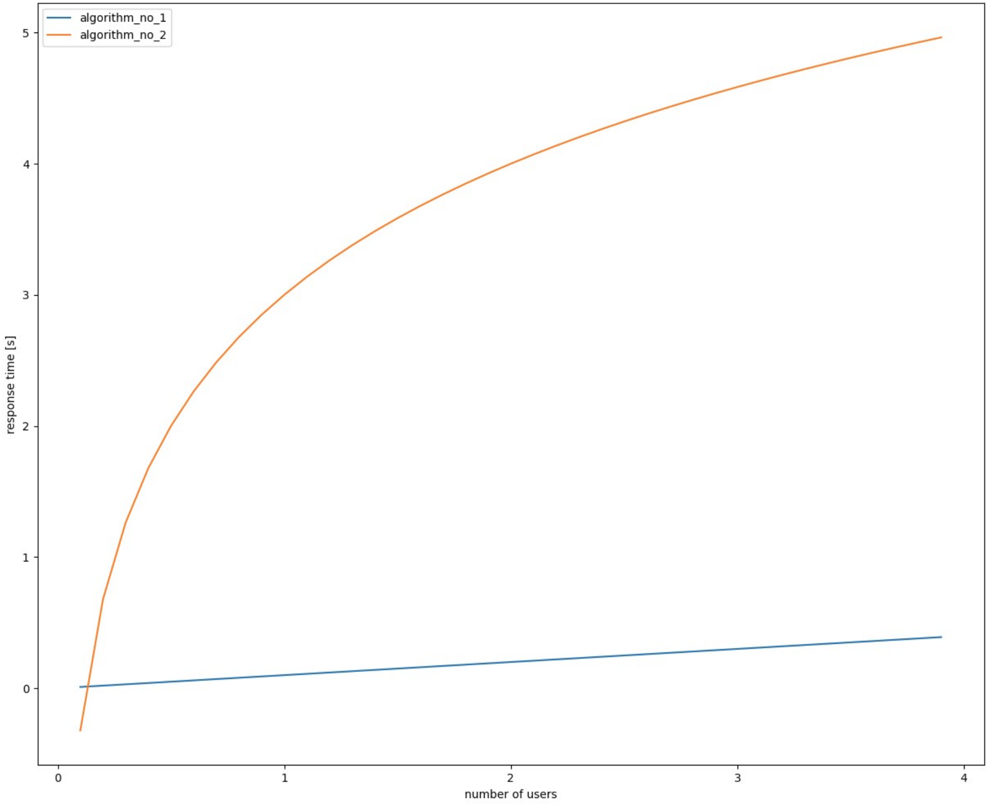
Dla n = 10000, f(n) = 0.00009995 * g(n).

$f(n) = O(10n^3 + 5n + 100) = n^3$

Dlaczego określenie złożoności obliczeniowej algorytmu ma znaczenie?

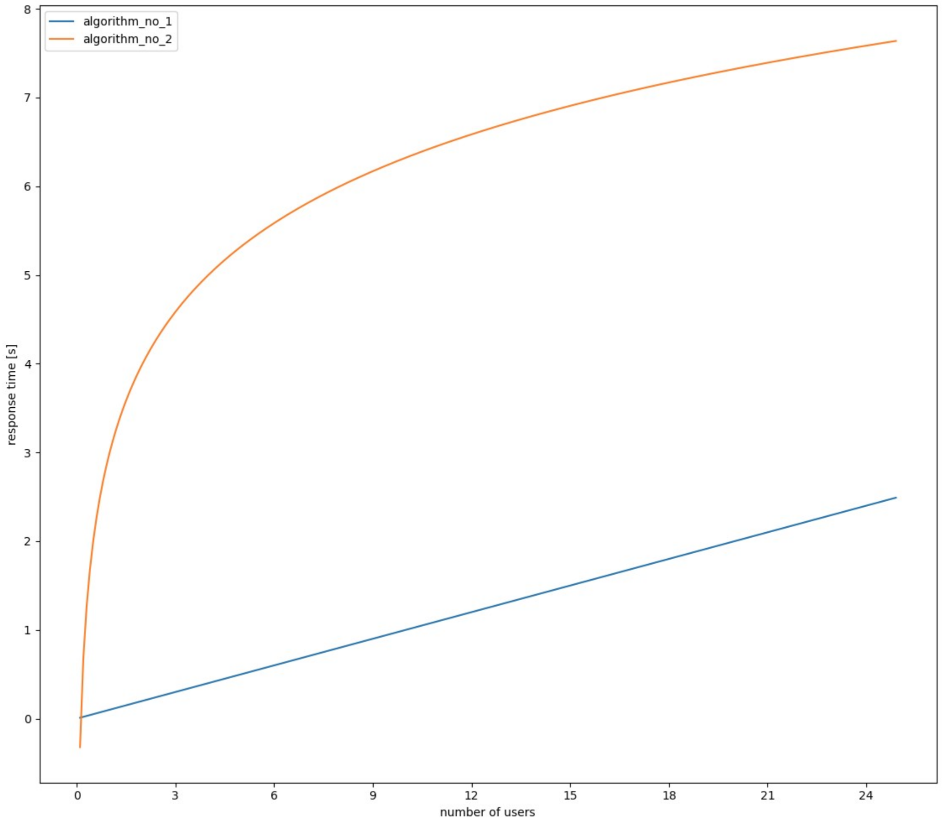
- Pozwala w szybki sposób porównywać różne algorytmy.
- Odrzucać rozwiązania, które na pewno nie sprawdzą się w środowisku produkcyjnym.
- Odnajdywać potencjalne bottleneck'i zanim staną się one problemem.

Przykład I



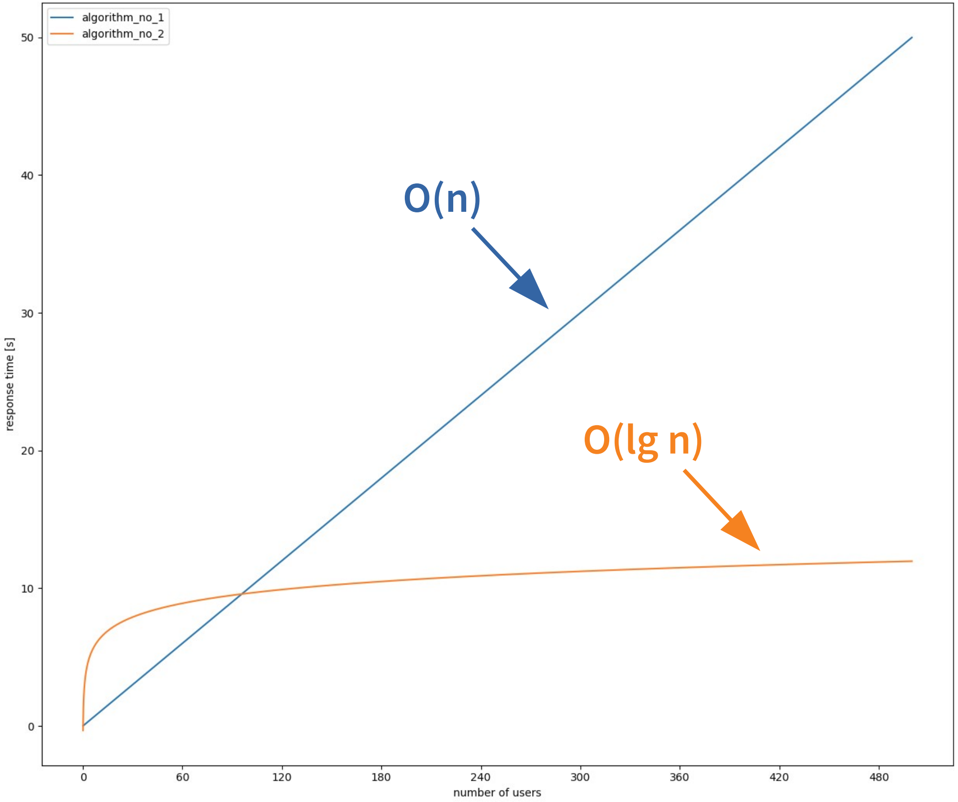
Przy pięciu użytkownikach, pierwsze rozwiązanie jest dwudziestopięciokrotnie szybsze.

Przykład I



Przy dwudziestupięciu użytkownikach, pierwsze rozwiązanie jest czterokrotnie szybsze.

Przykład I



Przy pięciuset użytkownikach, pierwsze rozwiązanie jest pięciokrotnie wolniejsze.

Przykład II

	Commodore 64	Intel i7 PC
taktowanie procesora	1 MHz	5 GHz
złożoność algorytmu	n lg n	n^2
posortowanie dziesięciu milionów liczb	4 minuty	5.5 godziny

* Dla uproszczenia przyjmujemy, że jedna operacja trwa jeden cykl zegara.

Metodyka analizy złożoności czasowej algorytmów.

Schemat badania złożoności czasowej algorytmu.

zadanie – ...
n – ...

① Analiza liczby wywołań operacji elementarncyh.

	ilość wywołań	czas pojedyńczego wywołania
def some_function(**kwargs):		
...	n	t_1
...	n^2	t_2
return ...	1	t_3

② Wyprowadzenie funkcji w notacji asymptotycznej.

t_1, t_2, t_3 - stałe

$f(n) = t_1 \cdot n + t_2 \cdot n^2 + t_3$

$O(f(n)) = O(t_1 \cdot n + t_2 \cdot n^2 + t_3) = O(n^2)$

3 Stworzenie generatora danych wejściowych zależnego od zmiennej n.

```
def data_gen(n):
    ...
    return {"argument": argument}
```

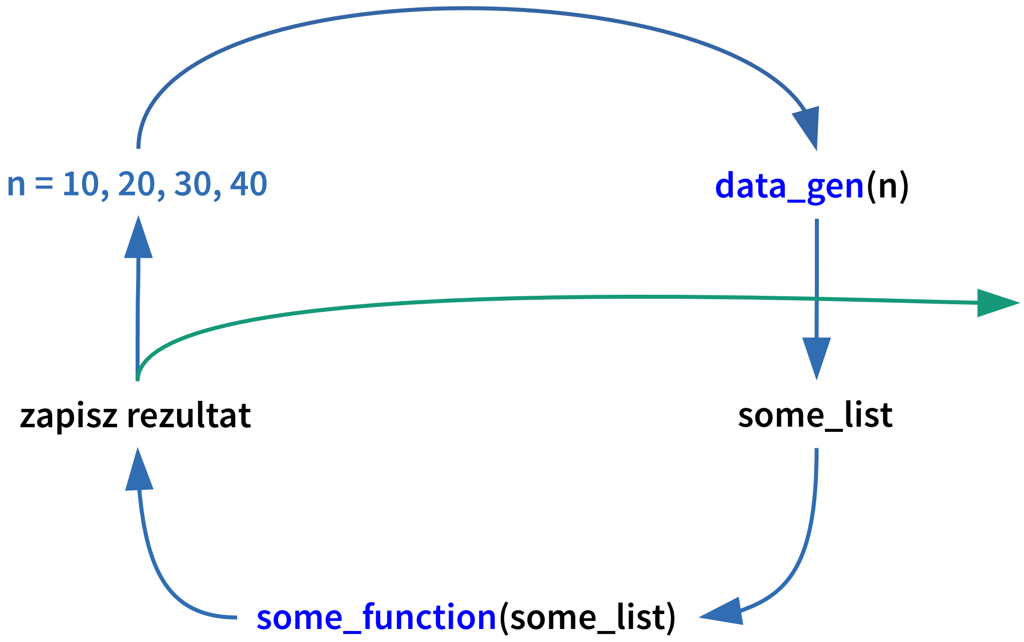
4 Wykonanie pomiarów dla zadanych wartości n oraz analiza otrzymanychrezultatów.

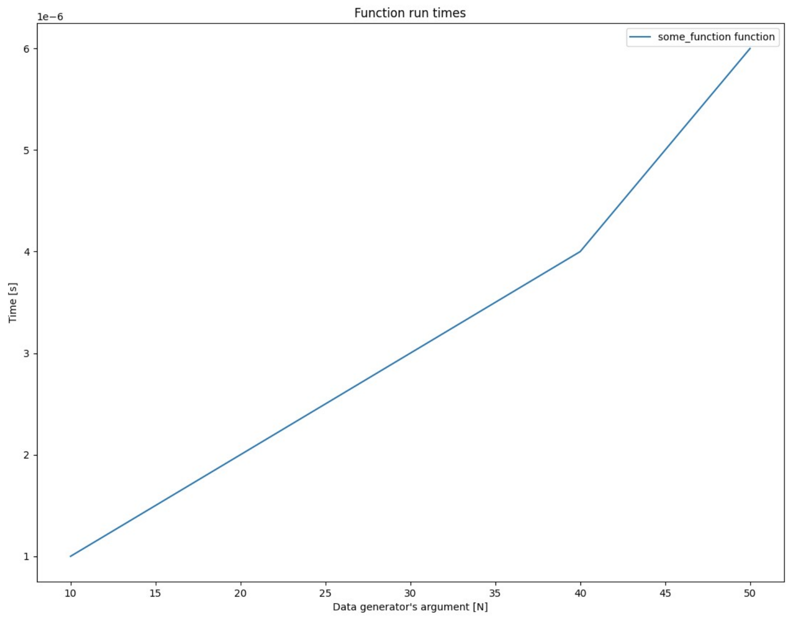
n 10 20 30 40 50

```
profiler = Profiler()

profiler.run_time_analysis(
    func=some_function,
    data_gen=data_gen,
    gen_min_arg=10,
    gen_max_arg=50,
    gen_steps=5,
    iterations=20,
    find_big_o=True
)
```

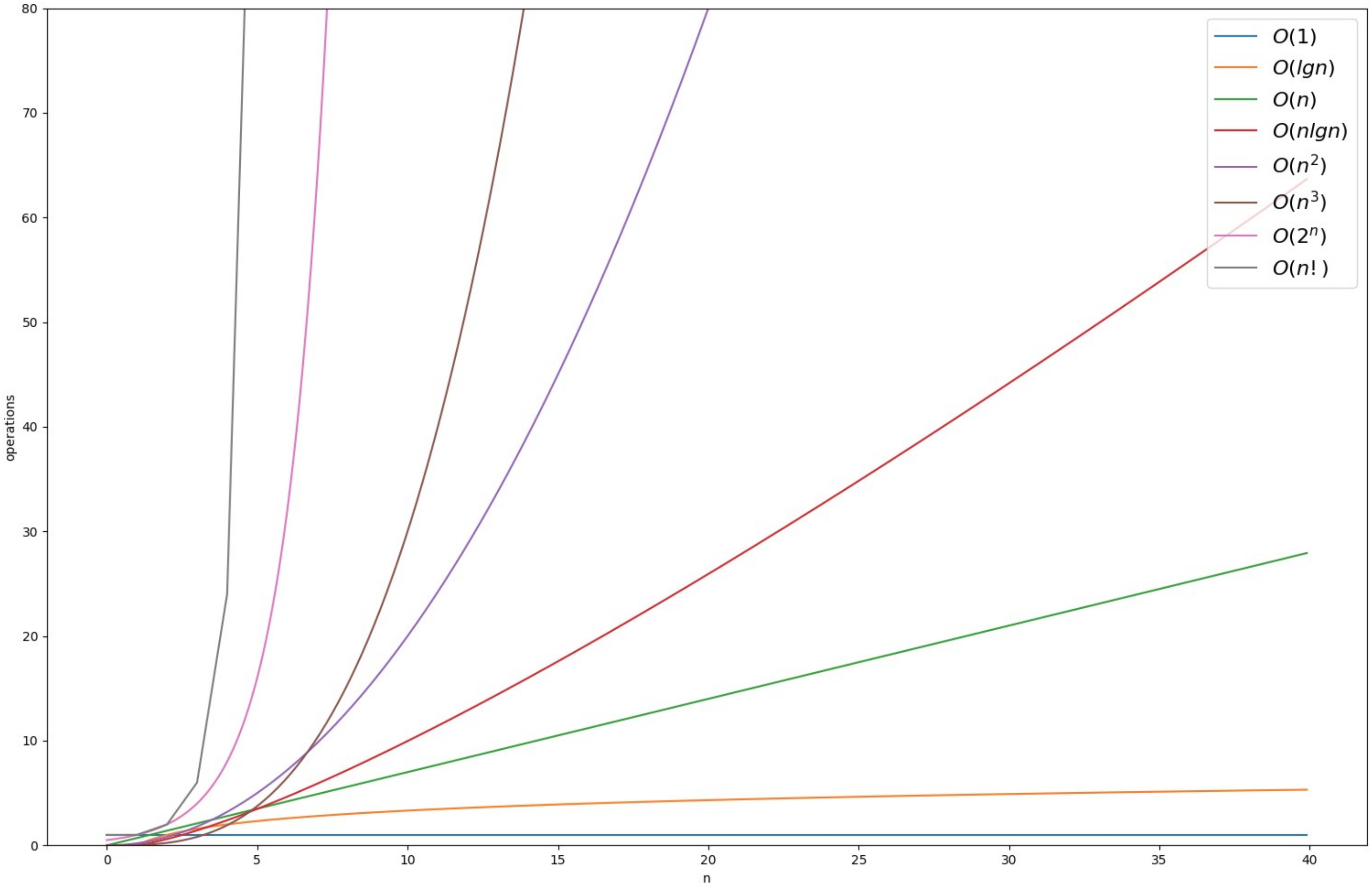
Liczba iteracji wykonanych dla otrzymania dokładniejszego wyniku.

Znajdź funkcję, która dokonuje najlepszej predykcji wyników.

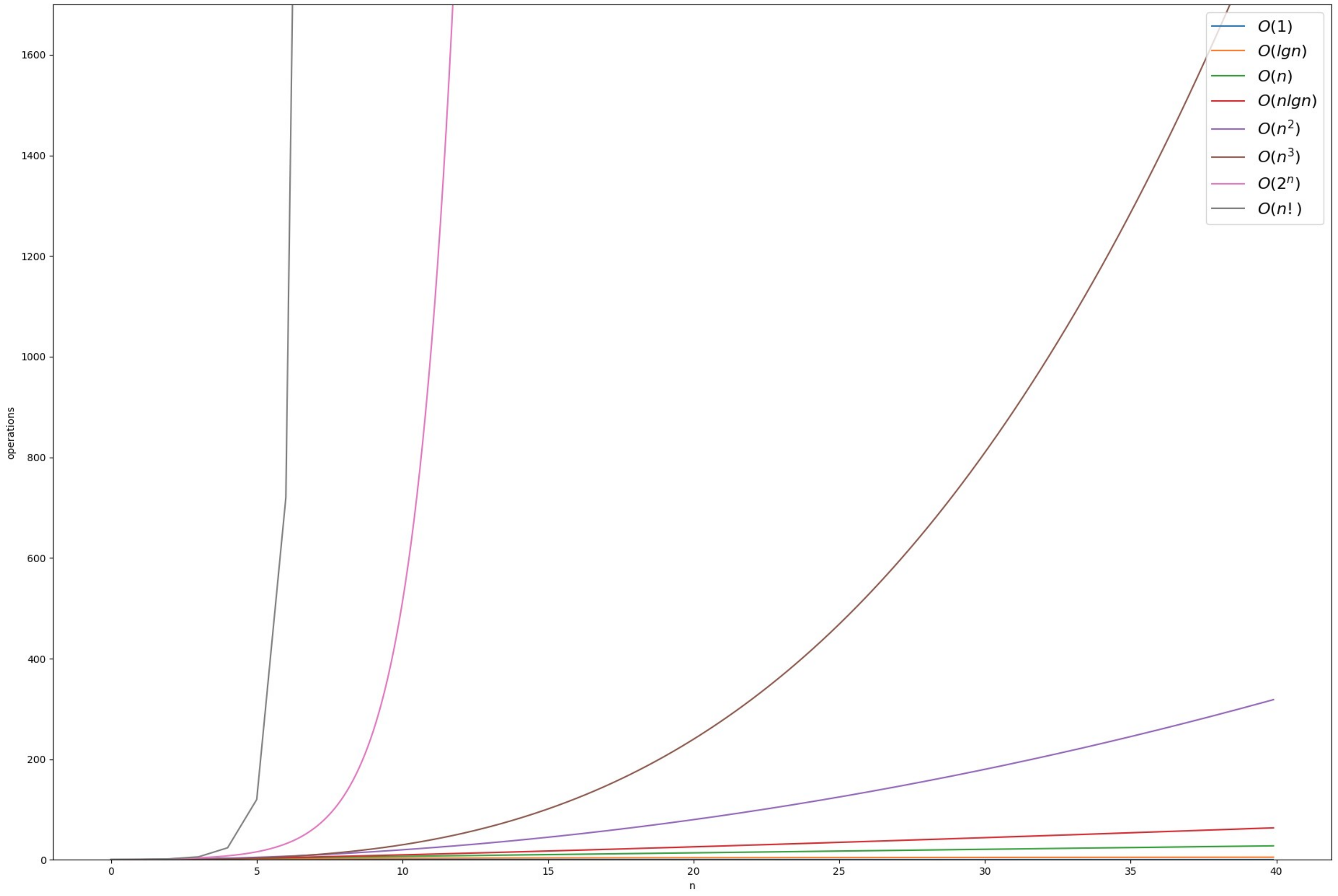

n = 10, 20, 30, 40
data_gen(n)
zapisz rezultat
some_list
some_function(some_list)


Function run times
1e−6
some_function function
Time [s]
Data generator's argument [N]
1
2
3
4
5
6
10
15
20
25
30
35
40
45
50

Rodzaje funckji.

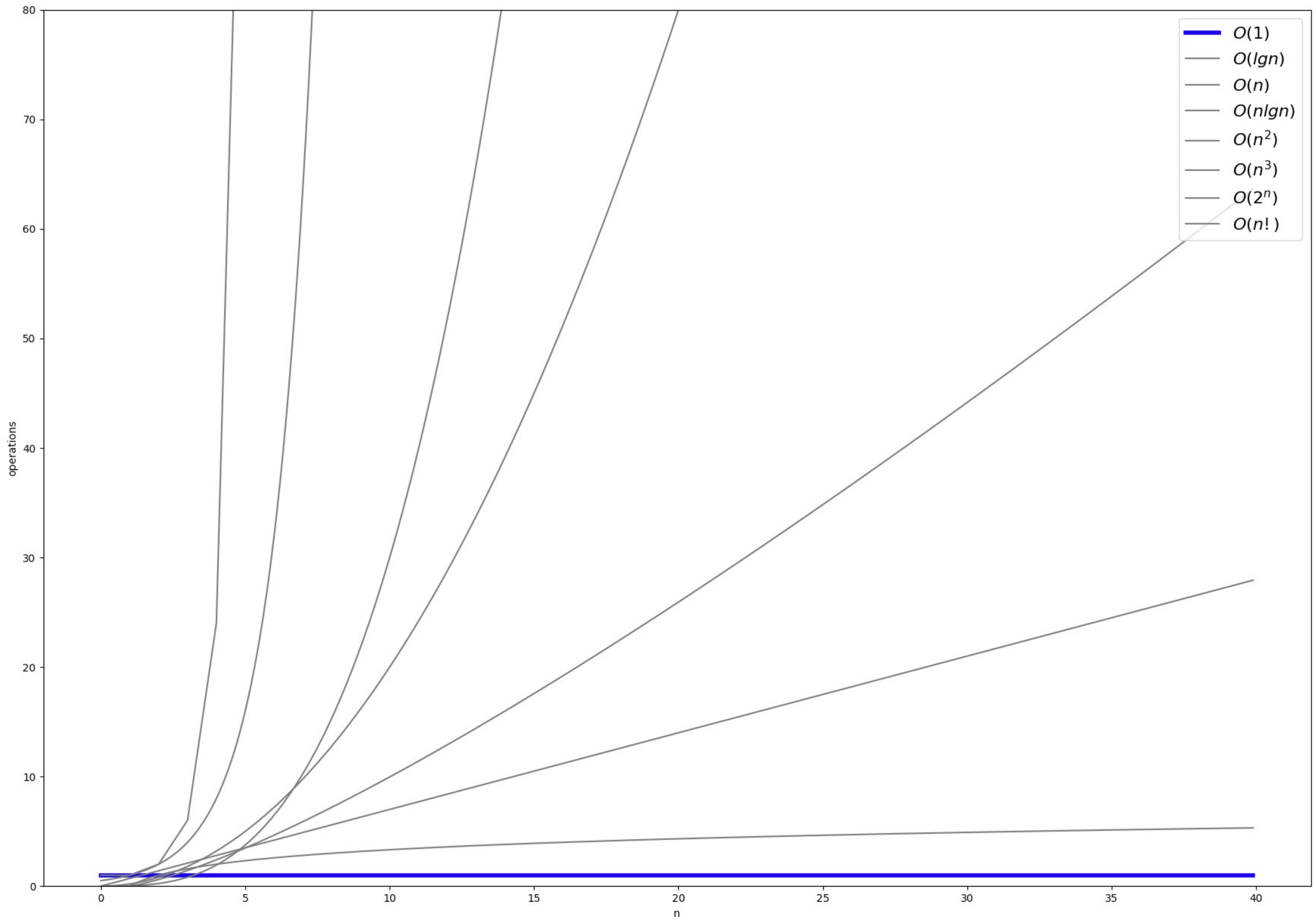


80
70
60
50
40
30
20
10
0
operations
0
5
10
15
20
25
30
35
40
n
$O(1)$
$O(lgn)$
$O(n)$
$O(nlgn)$
$O(n^2)$
$O(n^3)$
$O(2^n)$
$O(n!)$



operations
n
0
200
400
600
800
1000
1200
1400
1600
0
5
10
15
20
25
30
35
40
$O(1)$
$O(lgn)$
$O(n)$
$O(nlgn)$
$O(n^2)$
$O(n^3)$
$O(2^n)$
$O(n!)$

Stała złożoność - O(1)



O(1)
O(lgn)
O(n)
O(nlgn)
O(n^2)
O(n^3)
O(2^n)
O(n!)
operations
n
0
10
20
30
40
50
60
70
80
0
5
10
15
20
25
30
35
40

Stała złożoność - O(1)

Ilość oraz wielkość danych wejściowych może się zmieniać, lecze nie mają one wpływu na liczbę operacji wykonanych wewnątrz ciała funkcji. Nie występuje parametr n.

Stała liczba operacji trwających stałą ilość czasu.

```
def some_function(number_1=1, number_2=2):
    for _ in range(100):
        number_1 += 1

    number_1 -= 10
    ...
    ...
    ...
    result = number_1 + number_2
    return ...
```

stała złożoność - O(1)

zadanie – Zwróć ostatni element listy.
n – liczba elementów listy

big_o/time/constant_complexity.py	ilość wywołań	czas pojedynczego wywołania
`from time import sleep`		
`def get_latst_number(numbers_list):`		
`    last_number = numbers_list[-1]`	1	t_1
`    sleep(0.03)`	1	t_2
`    return last_number`	1	t_3

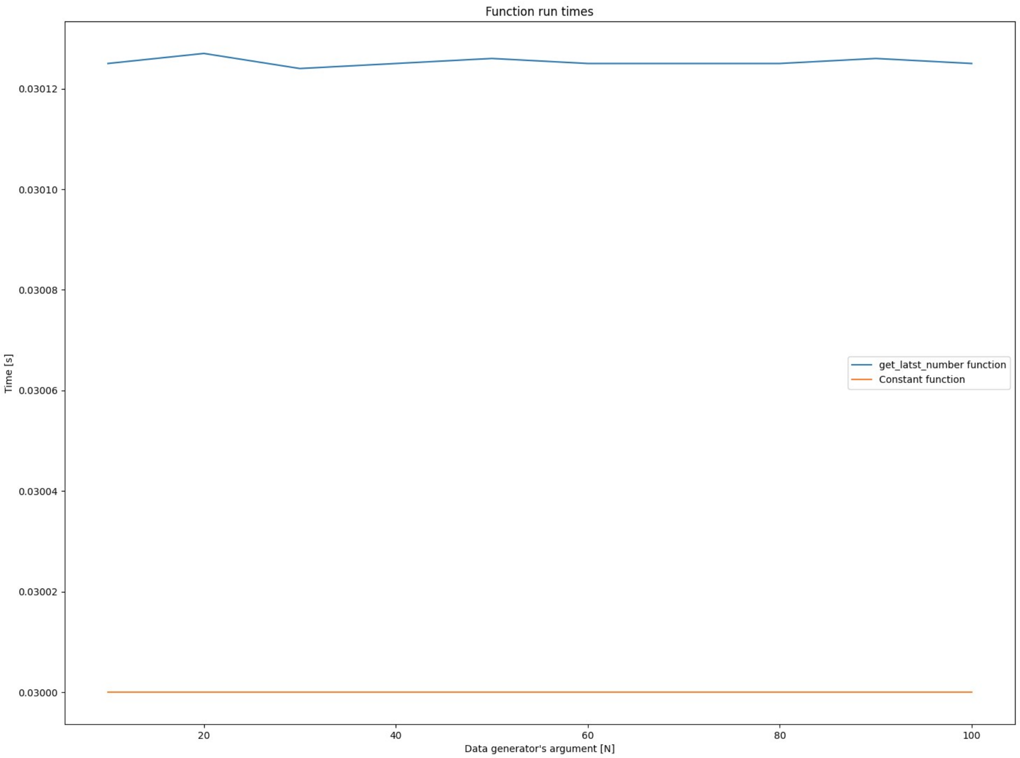
**Wielkość danych wejściowych jest zmienna lecz nie ma wpływu na liczbzę operacji.
Zakładamy, że czas dostępu do każdego elementu listy jest stały.**

t_1, t_2, t_3 - stałe

$f(n) = t_1 + t_2 + t_3$

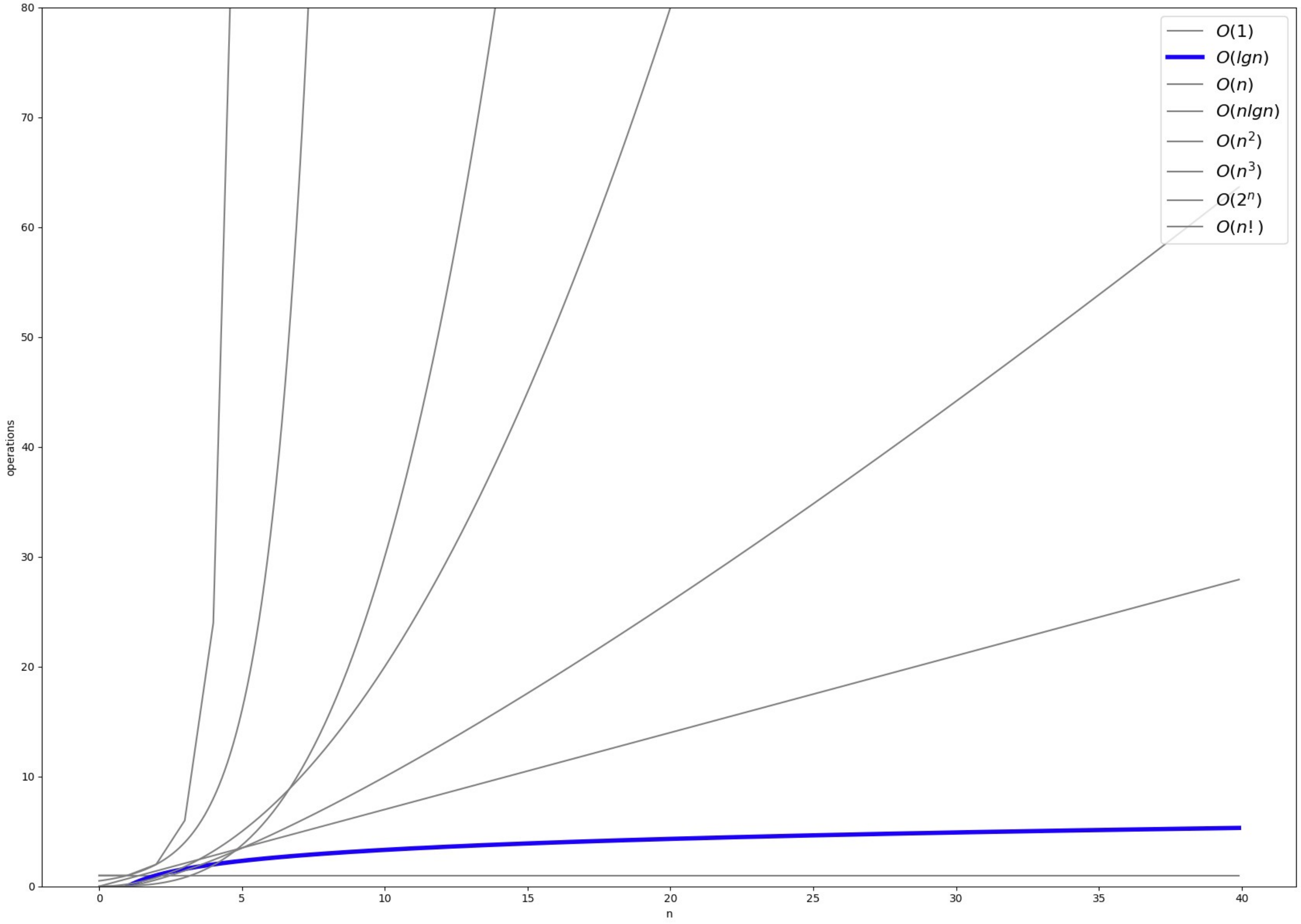
$t_1' = t_1 + t_2 + t_3$

$O(f()) = O(t_1') = O(1)$



P big_o/time/constant_complexity_profile.py

Złożoność logarytmiczna – O(lg n)



80
70
60
50
40
30
20
10
0
operations
0
5
10
15
20
25
30
35
40
n
O(1)
O(lgn)
O(n)
O(nlgn)
O(n^2)
O(n^3)
O(2^n)
O(n!)

Złożoność logarytmiczna – O(lg n)

Zmienna number nie ma wpływu na liczbę wykonanych operacji.
Tylko argument n determinuje złożoność obliczeniową funkcji.

```
def some_function(number, n):
    number += 1
    class_instance = SomeClass()

    while n >= 1:
        ...
        ...
        n /= 2
    return ...
```

Stała liczba operacji trwających stałą ilość czasu.

Podczas każdej iteracji zmniejszamy wielkość zbioru do przeprocesowania o połowę.

złożoność logarytmiczna – O(lg n)

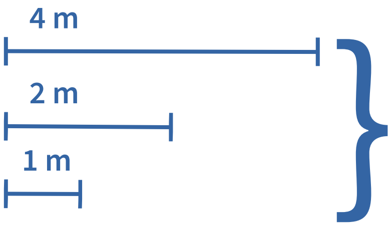
Podczas każdej iteracji zmniejszamy wielkość zbioru do przeprocesowania o połowę.

```
def some_function(n):
  if n > 1:
    some_result = …
    return  some_result + some_function(n / 2)
  else:
    return  ...
```

złożoność logarytmiczna – O(lg n)

zadanie – Oblicz ile razy dany odcinek można dzielić na połowe.
Minimalna gługość odcinka wynosi 1.
n – długość odcinka

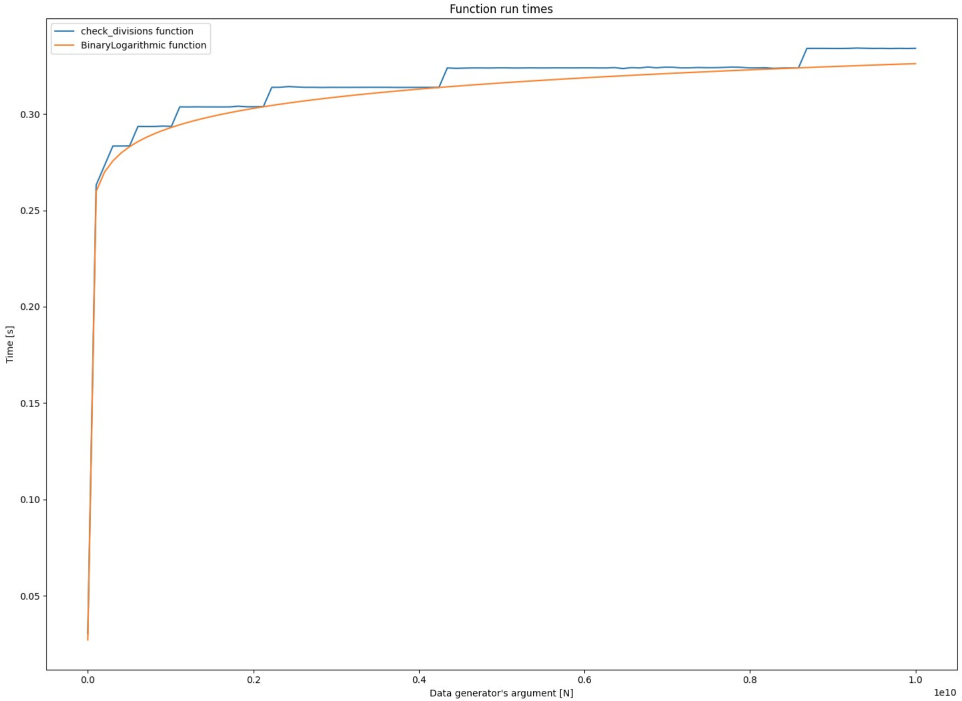
! $\log_2 n = \lg n$



$\log_2 4 = 2$

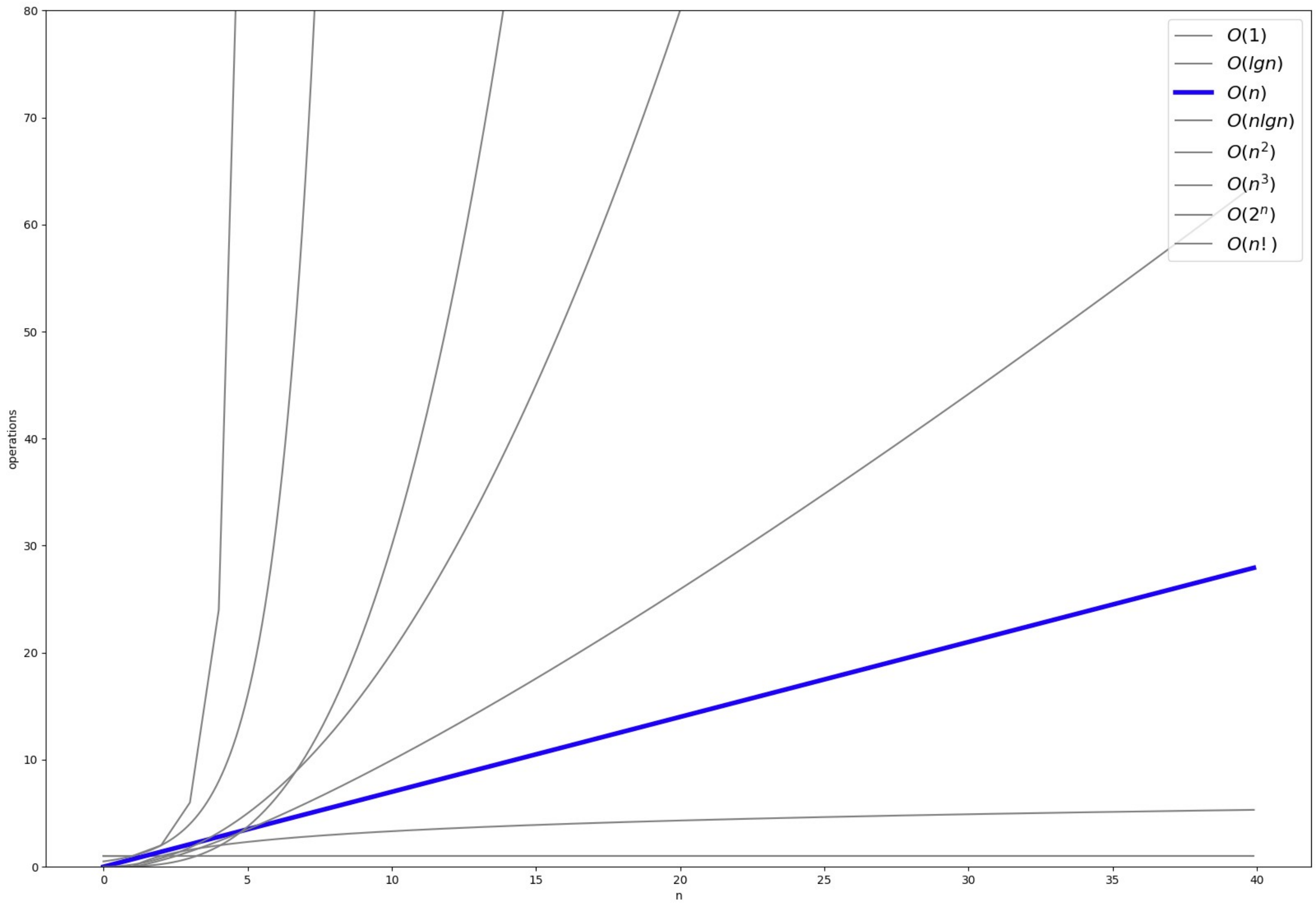
<> big_o/time/logarithmic_complexity.py	ilość wywołań	czas pojedyńczego wywołania
`from time import sleep`		
`def check_divisions(segment):`		
`    counter = 0`	1	t_1
`    while (segment / 2) >= 1:`	lg n + 1	t_2
`        segment /= 2`	lg n	t_3
`        counter += 1`	lg n	t_4
`        sleep(0.01)`	lg n	t_5
`    return counter`	1	t_6

$$f(n) = t_1 + t_2 \cdot (\lg n + 1) + t_3 \cdot \lg n + t_4 \cdot \lg n + t_5 \cdot \lg n + t_6 =$$
$$= (t_2 + t_3 + t_4 + t_5) \cdot \lg n + (t_1 + t_2 + t_6)$$
$$t_1' = t_2 + t_3 + t_4 + t_5$$
$$t_2' = t_2 + t_3 + t_4 + t_5$$
$$O(f(n)) = O(t_1' \cdot \lg n + t_2') = O(\lg n)$$



P big_o/time/logarithmic_complexity_profile.py

Złożoność liniowa - O(n)



80
70
60
50
40
30
20
10
0
operations
0
5
10
15
20
25
30
35
40
n
$O(1)$
$O(lgn)$
$O(n)$
$O(nlgn)$
$O(n^2)$
$O(n^3)$
$O(2^n)$
$O(n!)$

Złożoność liniowa - O(n)

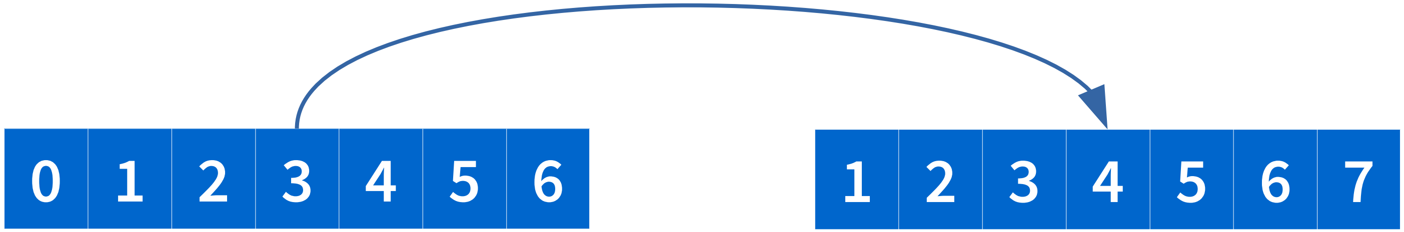
Liczba wykonanych operacji jest liniowo zależna od wielkości zbioru danych wejściowych.

```
def some_function(n):
    for ... in range(n):
        ...

    return ...
```

```
def some_function(n):
    if n > 0:
        ...
        return  ... + some_function(n - 1)
    else:
        return  ...
```

Złożoność liniowa - O(n)

zadanie – Zwiększ o jeden każdy element listy.
n – liczba elementów listy



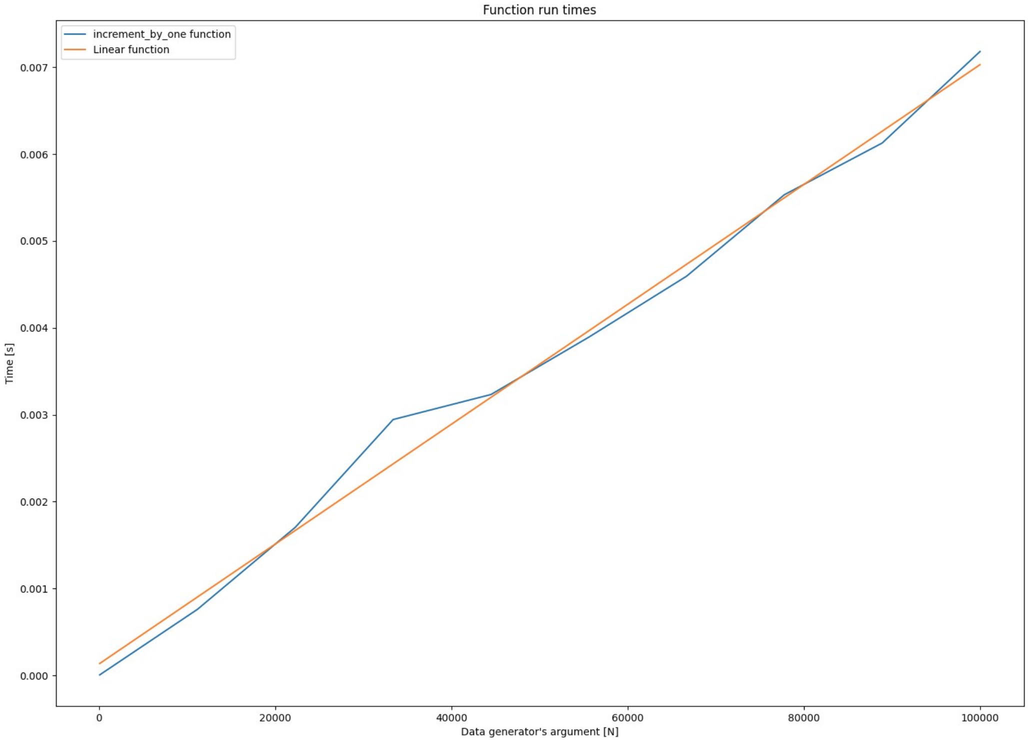
big_o/time/linear_complexity.py	ilość wywołań	czas pojedyńczego wywołania
`def increment_by_one(numbers_list):`		
`    incremented_list = []`	1	t_1
`    for number in numbers_list:`	n	t_2
`        incremented_number = number + 1`	n	t_3
`        incremented_list.append(incremented_number)`	n	t_4
`    return incremented_list`	1	t_5

$$f(n) = t_1 + t_2 \cdot n + t_3 \cdot n + t_4 \cdot n + t_5 = (t_2 + t_3 + t_4) \cdot n + t_1 + t_5$$

$$t_1' = t_2 + t_3 + t_4$$

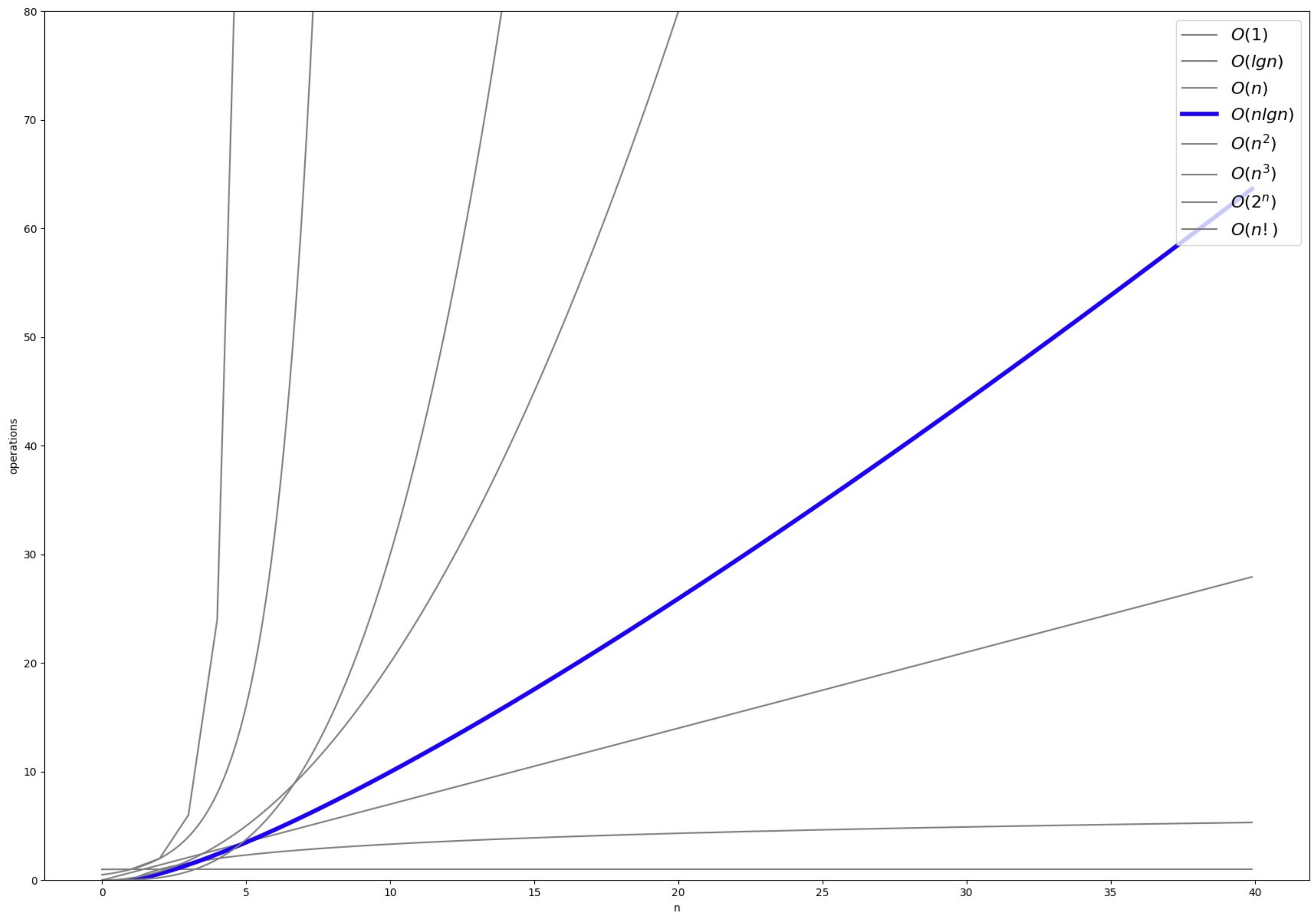
$$t_2' = t_1 + t_5$$

$$O(f(n)) = O(t_1' \cdot n + t_2') = O(n)$$



P big_o/time/linear_complexity_profile.py

Złożoność liniowo logarytmiczna – O(nlg n)



80
70
60
50
40
30
20
10
0
operations
0
5
10
15
20
25
30
35
40
n
$O(1)$
$O(lgn)$
$O(n)$
$O(nlgn)$
$O(n^2)$
$O(n^3)$
$O(2^n)$
$O(n!)$

złożoność liniowo logarytmiczna – O(nlg n)

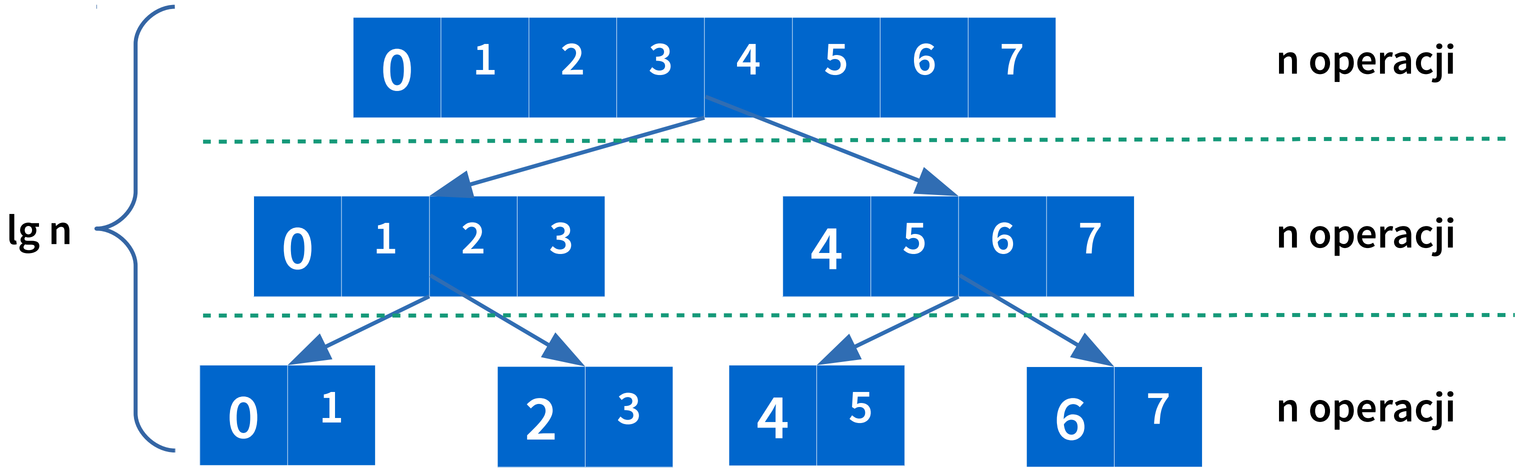
Wykonujemy lg n iteracji na całym zbiorze danych. Dzielimy problem na dwa podproblemy i rozwiązujemy każdy z nich.

```
def some_function(n):
   counter = n

   while counter >= 1:
      for … in range(n):
         …

      counter /= 2

   return ...
```

złożoność liniowo logarytmiczna – O(nlg n)

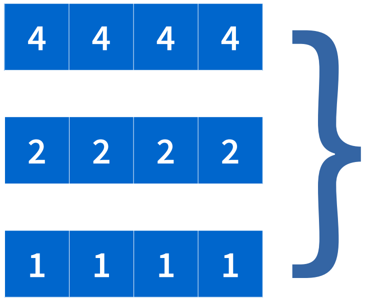
Zbiór danych dzielimy na połowę.
Dla każdej połowy zbioru, wykonujemy operacje na wszystkich jego elementach.

```
def some_function(n=[0,1,2,3,4,5,6,7]):
  if len(n) > 1:
    for ... in n:
      ...
    middle_index = len(n) // 2
    some_function(n[:middle_index])
    some_function(n[middle_index:])
```



Złożoność liniowo logarytmiczna – O(nlg n)

zadanie – Dopóki odcinek jest większy bądź równy jeden, dziel wszystkie element listy na połowę.
Długość początkowa każdego odcinka jest równa długości listy.
n – liczba elementów listy



$\log_2 4 = 2$

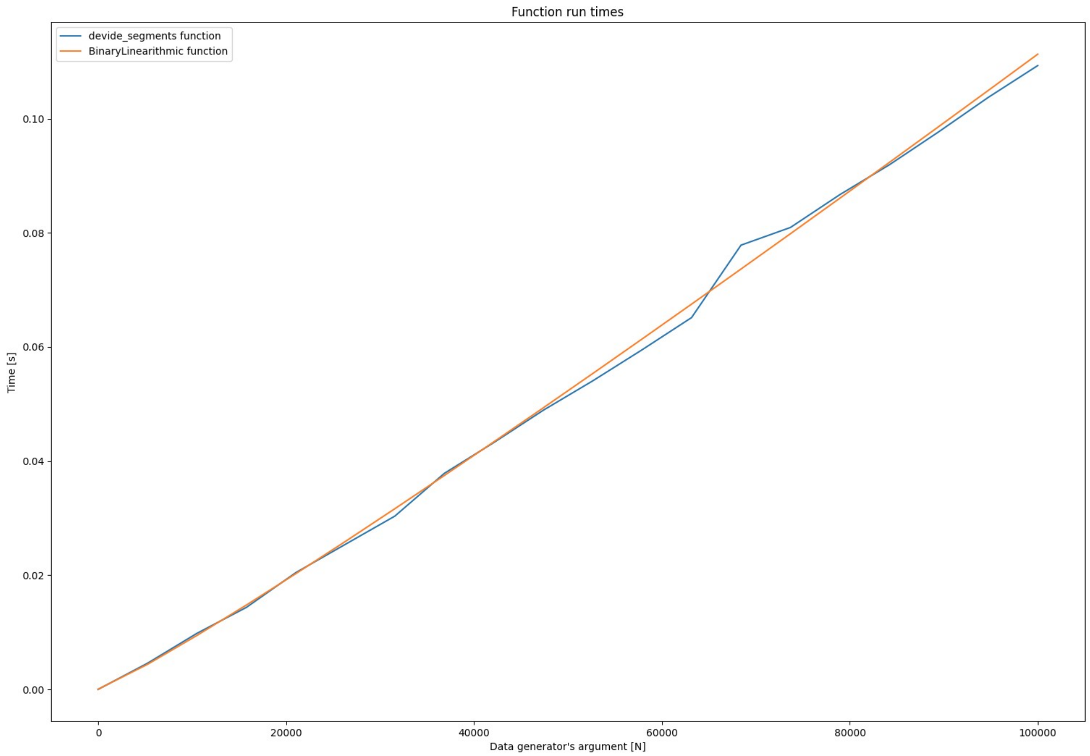
big_o/time/linearithmic_complexity.py	ilość wywołań	czas pojedynczego wywołania
`def devide_segments(segments):`		
`    n = len(segments)`	1	t_1
`    counter = n`	1	t_2
`    while (counter / 2) >= 1:`	lg n + 1	t_3
`        for index in range(n):`	n ▪ lg n	t_4
`            segments[index] /= 2`	n ▪ lg n	t_5
`        counter /= 2`	n ▪ lg n	t_6
`    return segments`	1	t_7

$f(n) = t_1 + t_2 + t_3 \cdot (\lg n + 1) + t_4 \cdot n \cdot \lg n + t_5 \cdot n \cdot \lg n + t_6 \cdot n \cdot \lg n + t_7 = (t_4 + t_5 + t_6) \cdot n \cdot \lg n + t_3 \cdot \lg n + (t_1 + t_2 + t_3 + t_7)$

$t_1' = t_4 + t_5 + t_6$

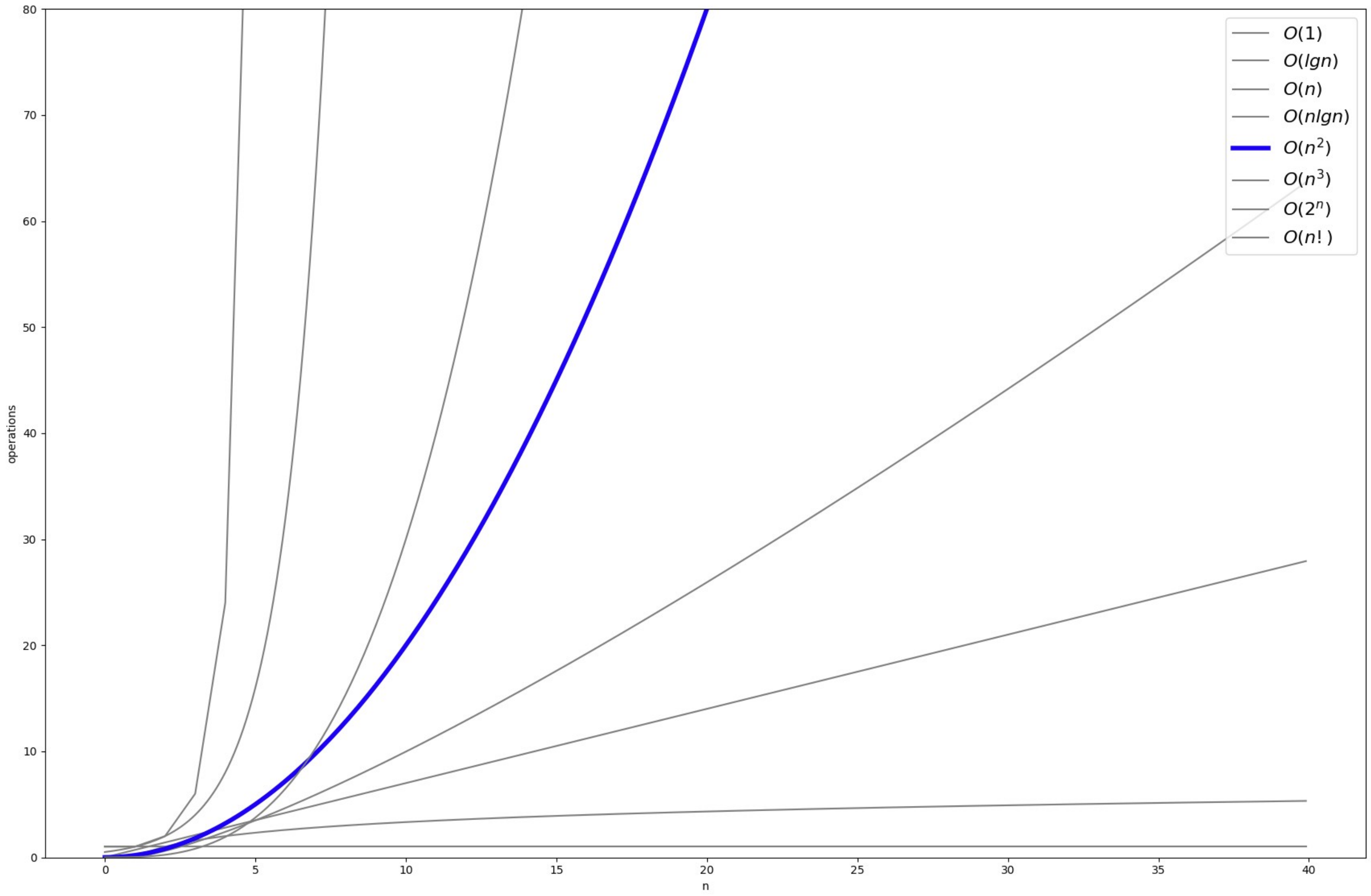
$t_2' = t_1 + t_2 + t_3 + t_7$

$O(f(n)) = O(t_1' \cdot n \cdot \lg n + t_3 \cdot \lg n + t_2') = O(n \lg n)$



P big_o/time/linearithmic_complexity_profile.py

Złożoność kwadratowa - O(n^2)



80
70
60
50
40
30
20
10
0
operations
0
5
10
15
20
25
30
35
40
n
$O(1)$
$O(lgn)$
$O(n)$
$O(nlgn)$
$O(n^2)$
$O(n^3)$
$O(2^n)$
$O(n!)$

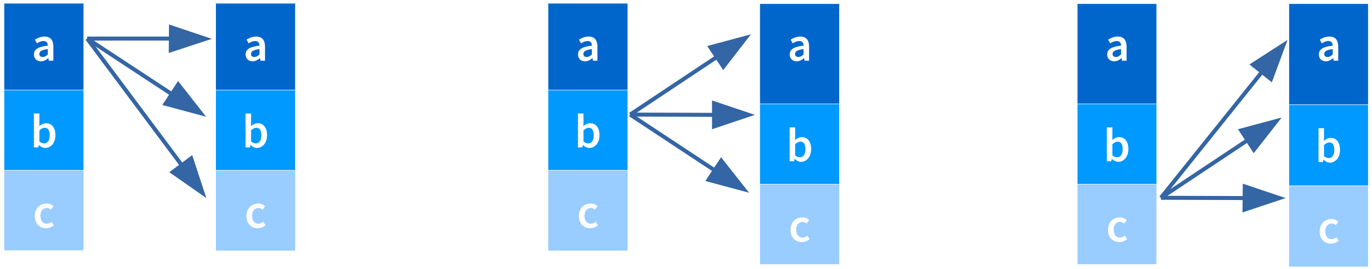
złożoność kwadratowa - O(n^2)

Wykonujemy operacje na wszystich parach elementów wejściowego zbioru danych.

```
def some_function(n):
    for x in range(n):
        for y in range(n):
            process(x, y)

    return ...
```

złożoność kwadratowa - O(n^2)

zadanie – Znajdź wszystkie permutacje z powtórzeniami elementów listy.
n – liczba elementów listy



aa	ab	ac	ba	bb	bc	ca	cb	cc

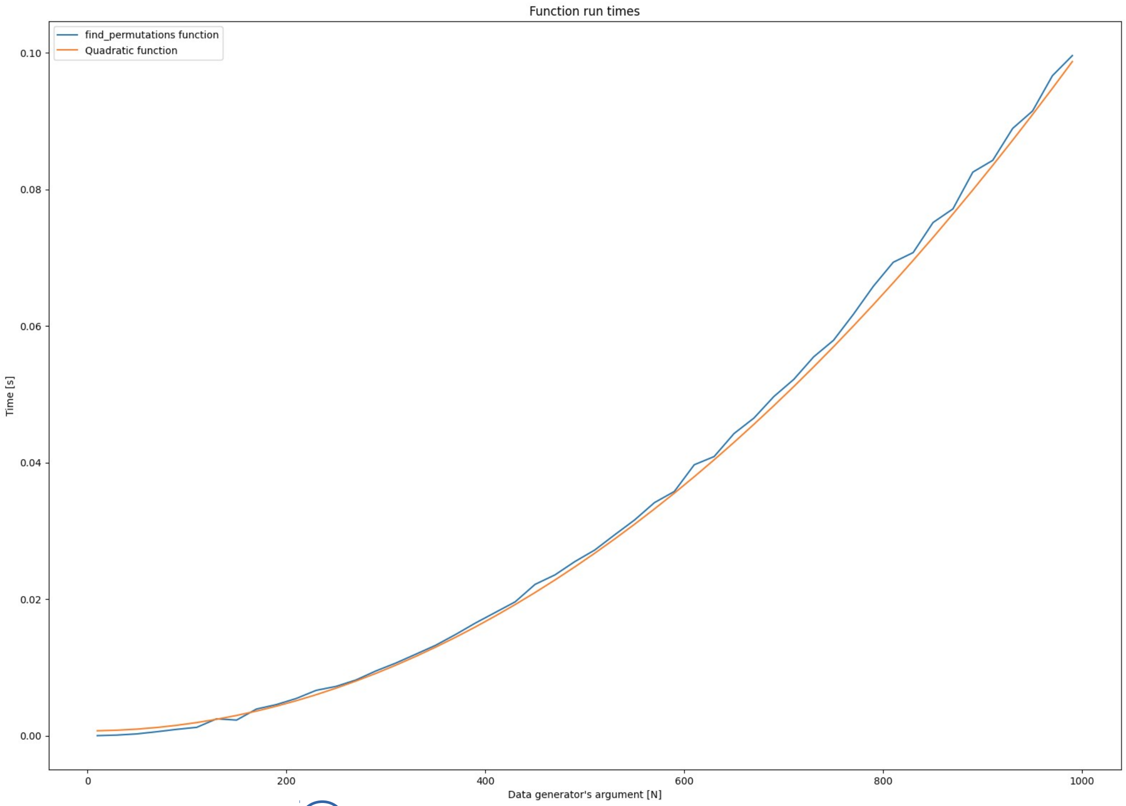
big_o/time/quadratic_complexity.py	ilość wywołań	czas pojedyńczego wywołania
`def find_permutations(characters_list):`		
`    permutations = []`	1	t_1
`    for first_character in characters_list:`	n	t_2
`        for second_character in characters_list:`	n^2	t_3
`            permutations.append(first_character + second_character)`	n^2	t_4
`    return permutations`	1	t_5

$f(n) = t_1 + t_2 \cdot n + t_3 \cdot n^2 + t_4 \cdot n^2 + t_5 = (t_3 + t_4) \cdot n^2 + t_2 \cdot n + t_1 + t_5$

$t_1' = t_3 + t_4$

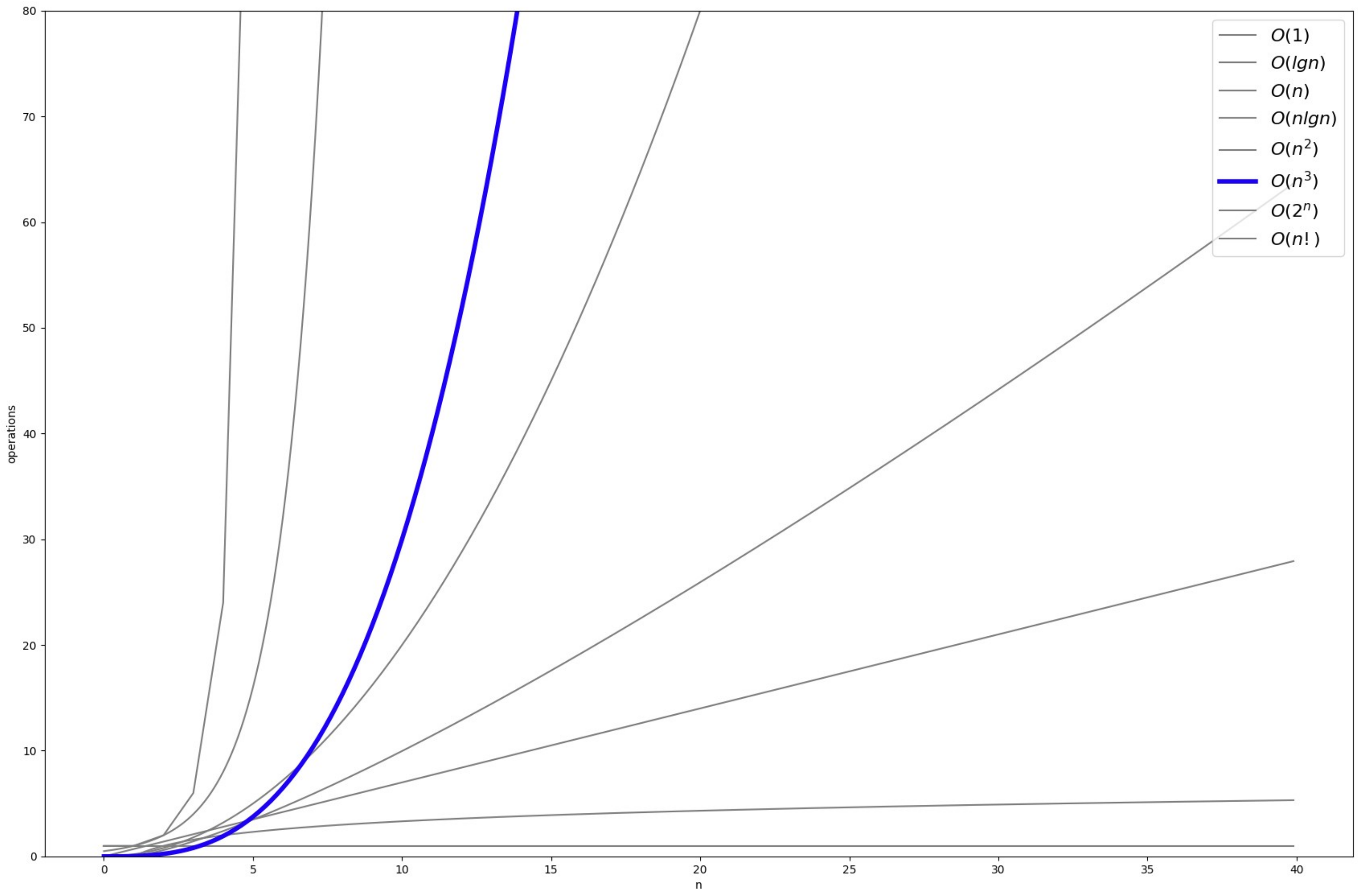
$t_2' = t_1 + t_5$

$O(f(n)) = O(t_1' \cdot n^2 + t_2 \cdot n + t_2') = O(n^2)$



P big_o/time/quadratic_complexity_profile.py

Złożoność sześcienna - O(n^3)



operations
n
0
5
10
15
20
25
30
35
40
10
20
30
40
50
60
70
80
$O(1)$
$O(lgn)$
$O(n)$
$O(nlgn)$
$O(n^2)$
$O(n^3)$
$O(2^n)$
$O(n!)$

złożoność sześcienna - O(n^3)

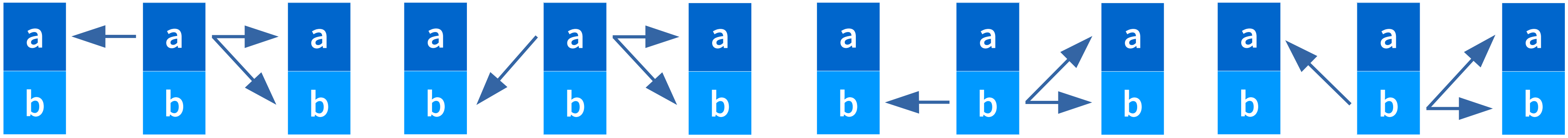
Podczas obliczania każdego możliwego rozwiązania, trzy razu musimy wybrać jeden element z n-elementowego zbioru danych.

Ów zbiory mogą być zbiorami pośrednich wyników. Liczy się liczba ich elementów.

```
def some_function(n):
    for x in range(n):
        for y in range(n):
            for z in range(n):
                process(x, y, z)

    return ...
```

złożoność sześcienna - O(n^3)

zadanie – Znajdź wszystkie trzy elementowe permutacje z powtórzeniami elementów listy.
n – liczba elementów listy



aaa	aab	baa	bab	bba	bbb	aba	abb

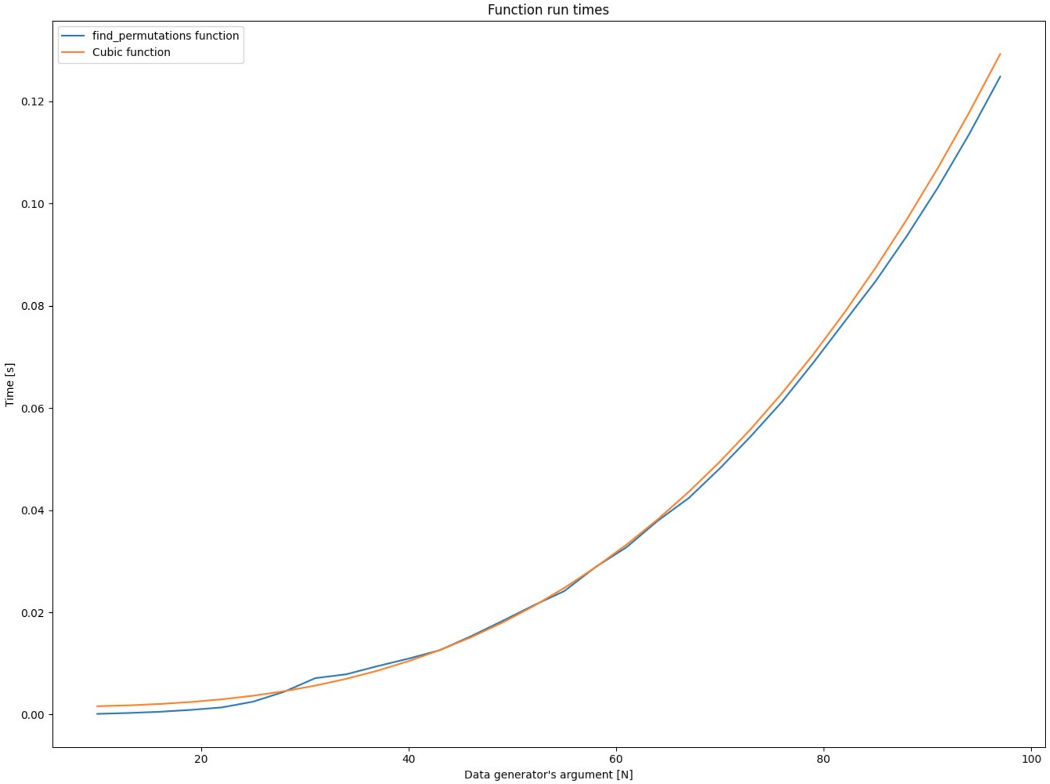
big_o/time/qubic_complexity.py	ilość wywołań	czas pojedyńczego wywołania
`def find_permutations(characters_list):`		
`    permutations = []`	1	t_1
`    for first_character in characters_list:`	n	t_2
`        for second_character in characters_list:`	n^2	t_3
`            for third_character in characters_list:`	n^3	t_4
`                permutation = first_character + second_character + third_character`	n^3	t_5
`                permutations.append(permutation)`	n^3	t_6
`    return permutations`	1	t_7

$$f(n) = t_1 + t_2 \cdot n + t_3 \cdot n^2 + t_4 \cdot n^3 + t_5 \cdot n^3 + t_6 \cdot n^3 + t_7 = (t_4 + t_5 + t_6) \cdot n^3 + t_3 \cdot n^2 + t_2 \cdot n + t_1 + t_7$$

$$t_1' = t_4 + t_5 + t_6$$

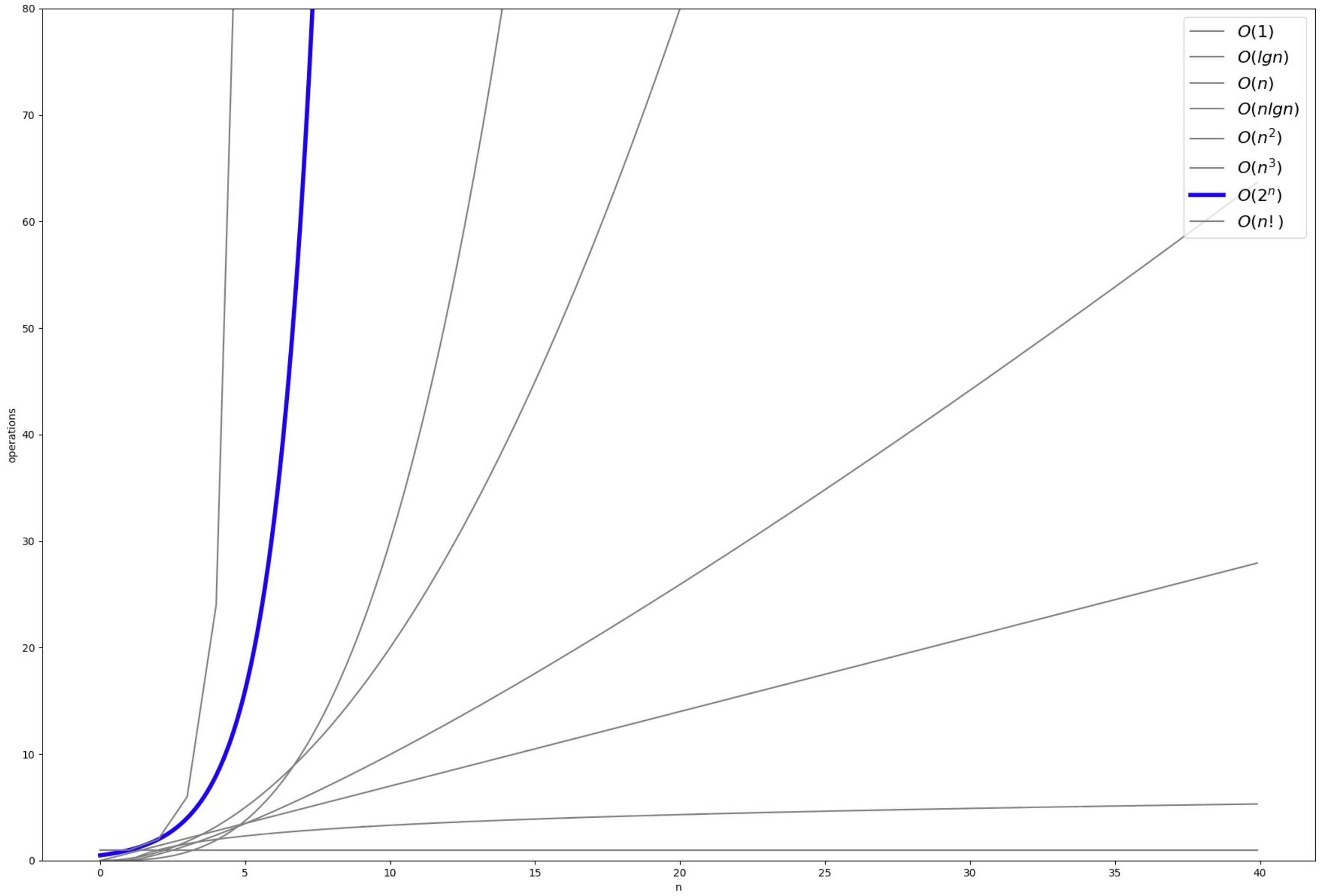
$$t_2' = t_1 + t_7$$

$$O(f(n)) = O(t_1' \cdot n^3 + t_3 \cdot n^2 + t_2 \cdot n + t_2') = O(n^3)$$



P big_o/time/qubic_complexity_profile.py

Złożoność wykładnicza – $O(2^n)$



80
70
60
50
40
30
20
10
0
operations
0
5
10
15
20
25
30
35
40
n
$O(1)$
$O(lgn)$
$O(n)$
$O(nlgn)$
$O(n^2)$
$O(n^3)$
$O(2^n)$
$O(n!)$

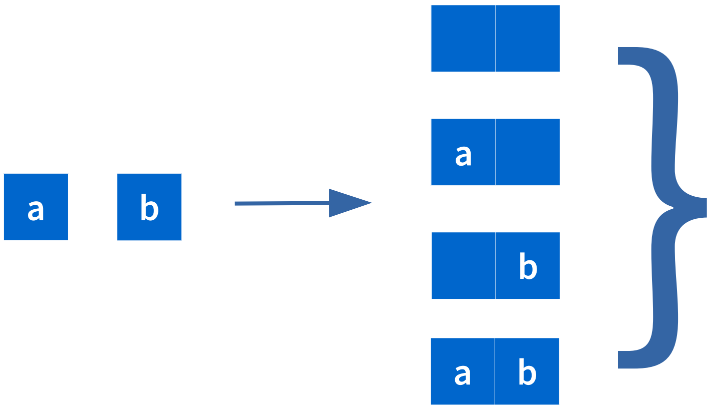
Złożoność wykładnicza – $O(2^n)$

Podczas każdej iteracji, wewnętrzna pętla wykonuje dwukrotnie więcej operacji.

```
def some_function(n):
    counter = 2

    for _ in range(n):
        for _ in range(counter):
            ...
        counter *= 2

    return ...
```

Złożoność wykładnicza – $O(2^n)$

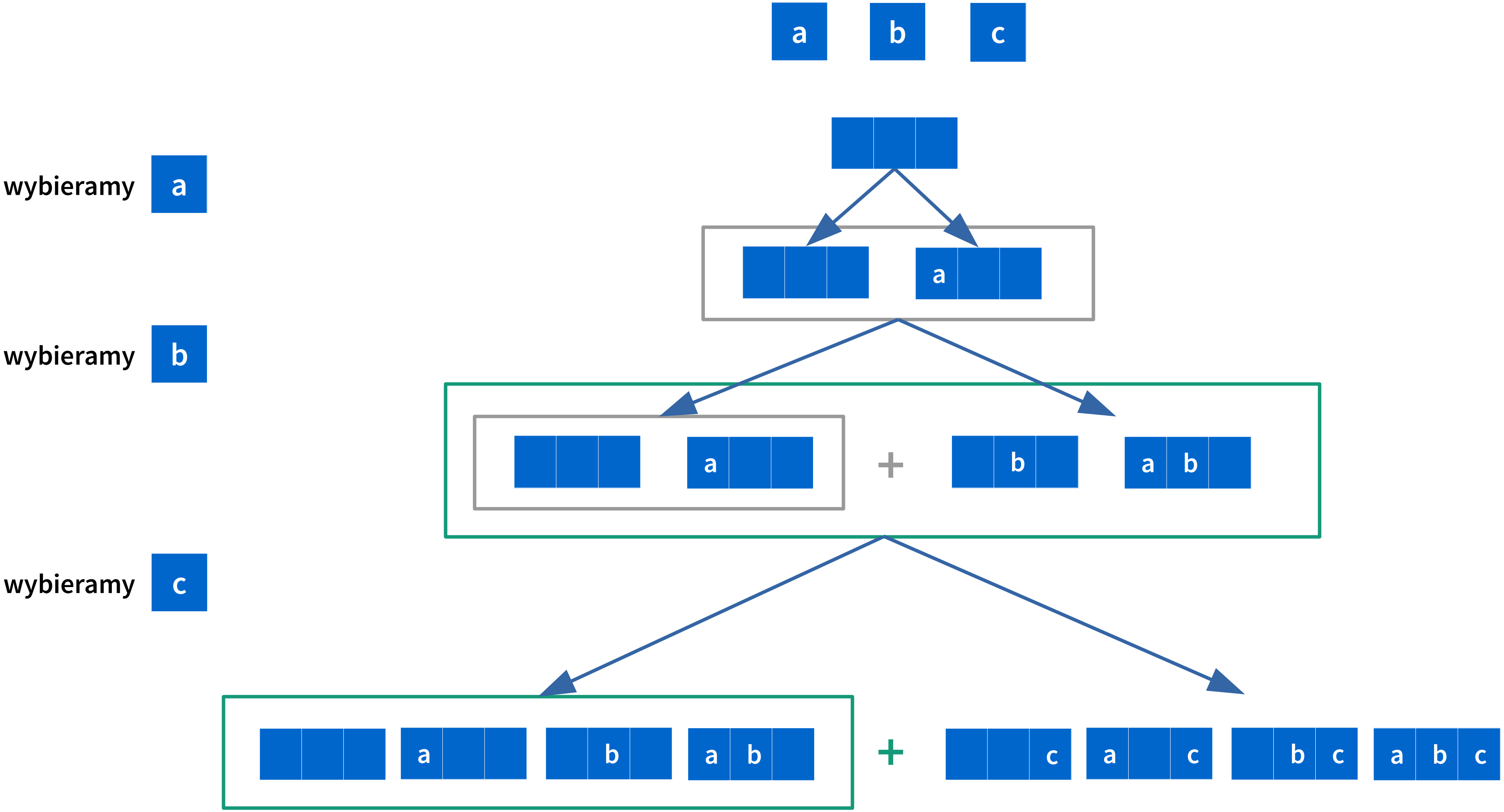
zadanie – Dla dostępnych dodatków, znajdź wszystkie możliwe rodzaje pizzy w menu.
n – liczba dodatków



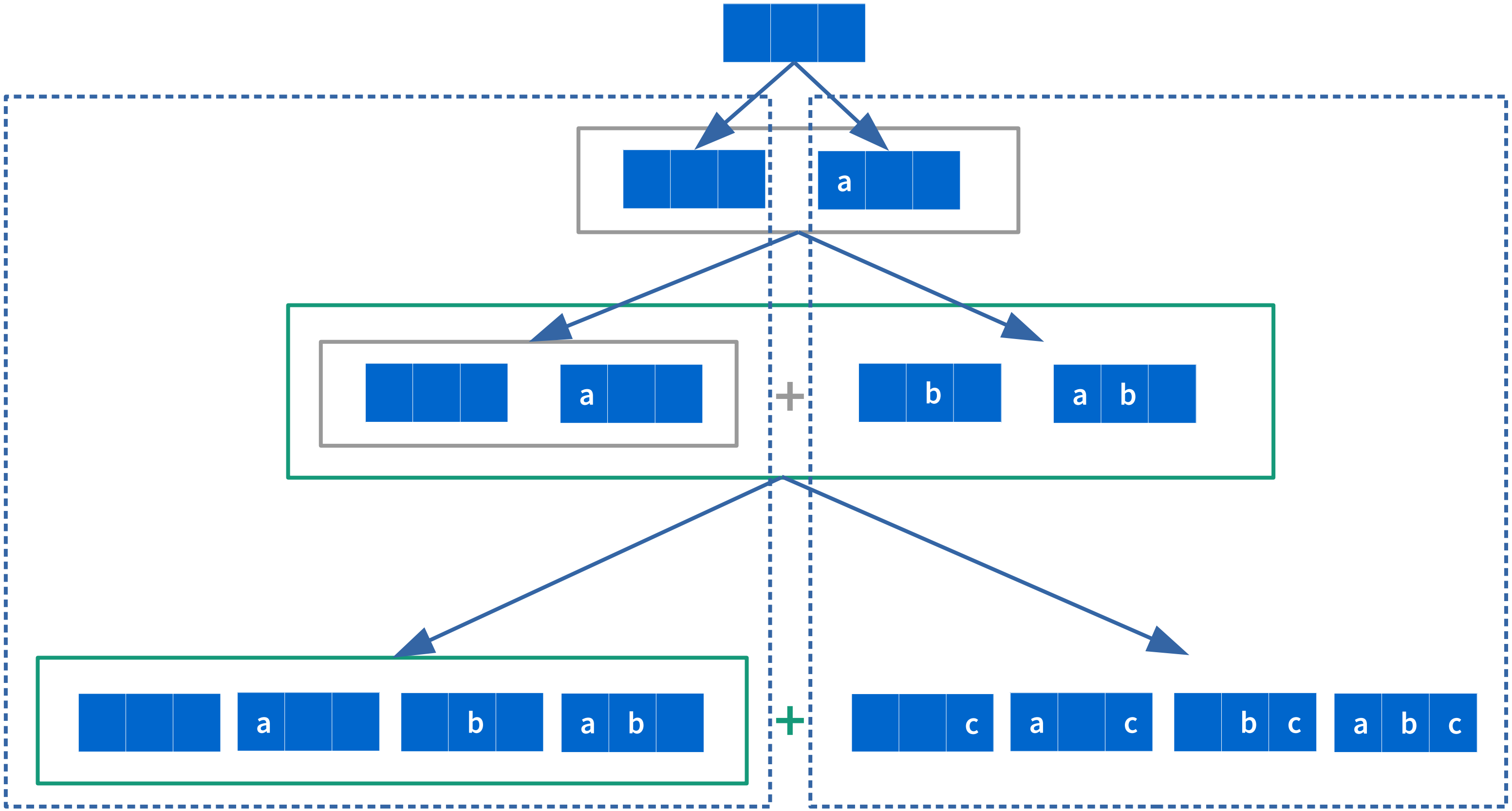
$$2 \cdot 2 = 2^2 = 4$$

Dla każdego dodatku mamy dwie możliwość. Dodajemy go do pizzy bądź nie.

$$\underbrace{2 \cdot 2 \cdot 2 \cdot \ldots \ldots \cdot 2}_{n} = 2^n$$



a
b
c
wybieramy a
wybieramy b
wybieramy c
a
+
b
a b
a
b
a b
+
c
a c
b c
a b c



Kopiujemy elementy listy $2^n - 1$ razy.
Dodajemy nowy składnik $2^n - 1$ razy.

big_o/time/exponential_complexity.py	ilość wywołań	czas pojedyńczego wywołania
`def find_menu(toppings):`		
`    menu = ['']`	1	t_1
`    for topping in toppings:`	n	t_2
`        new_variants = []`	n	t_3
`        for variant in menu:`	$2^n - 1$	t_4
`            new_variants.append(variant)`	$2^n - 1$	t_5
`        for index in range(len(new_variants)):`	$2^n - 1$	t_6
`            new_variants[index] += topping`	$2^n - 1$	t_7
`        menu += new_variants`	n	t_8
`    return menu`	1	t_9

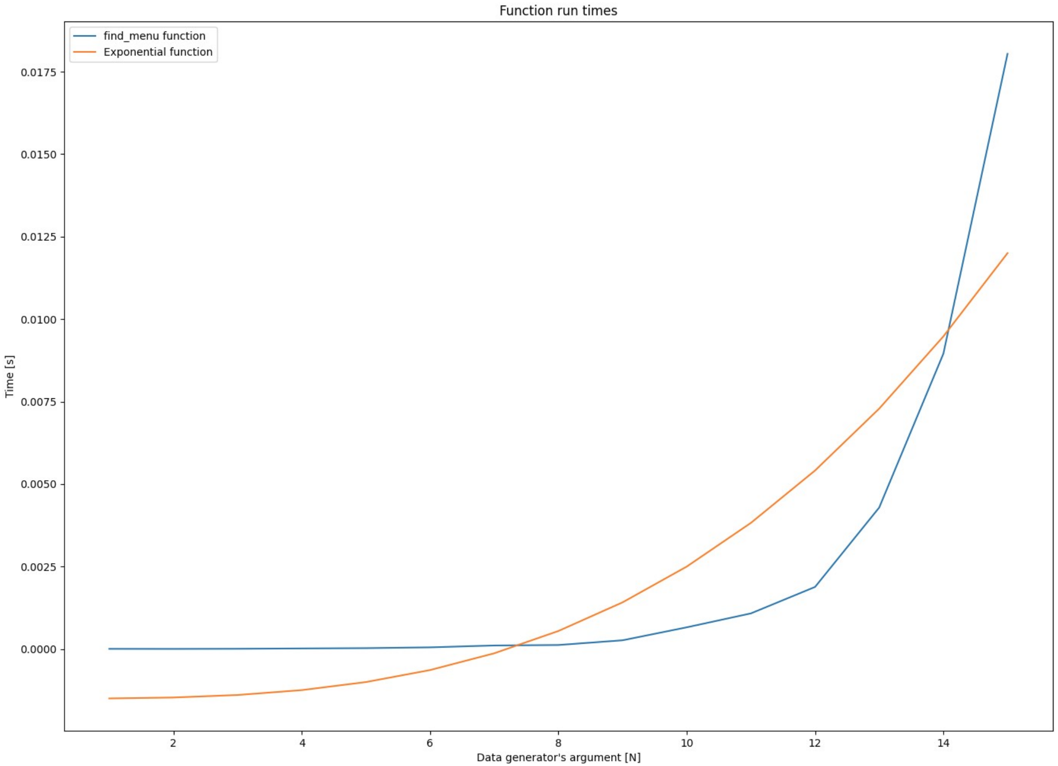
$f(n) = t_1 + t_2 \cdot n + t_3 \cdot n + t_4 \cdot (2^n - 1) + t_5 \cdot (2^n - 1) + t_6 \cdot (2^n - 1) + t_7 \cdot (2^n - 1) + t_8 \cdot n + t_9 = (t_4 + t_5 + t_6 + t_7) \cdot 2^n + (t_2 + t_3 + t_8) \cdot n + t_1 - t_4 - t_5 - t_6 + t_7 + t_9$

$t_1' = t_4 + t_5 + t_6 + t_7$

$t_2' = t_2 + t_3 + t_8$

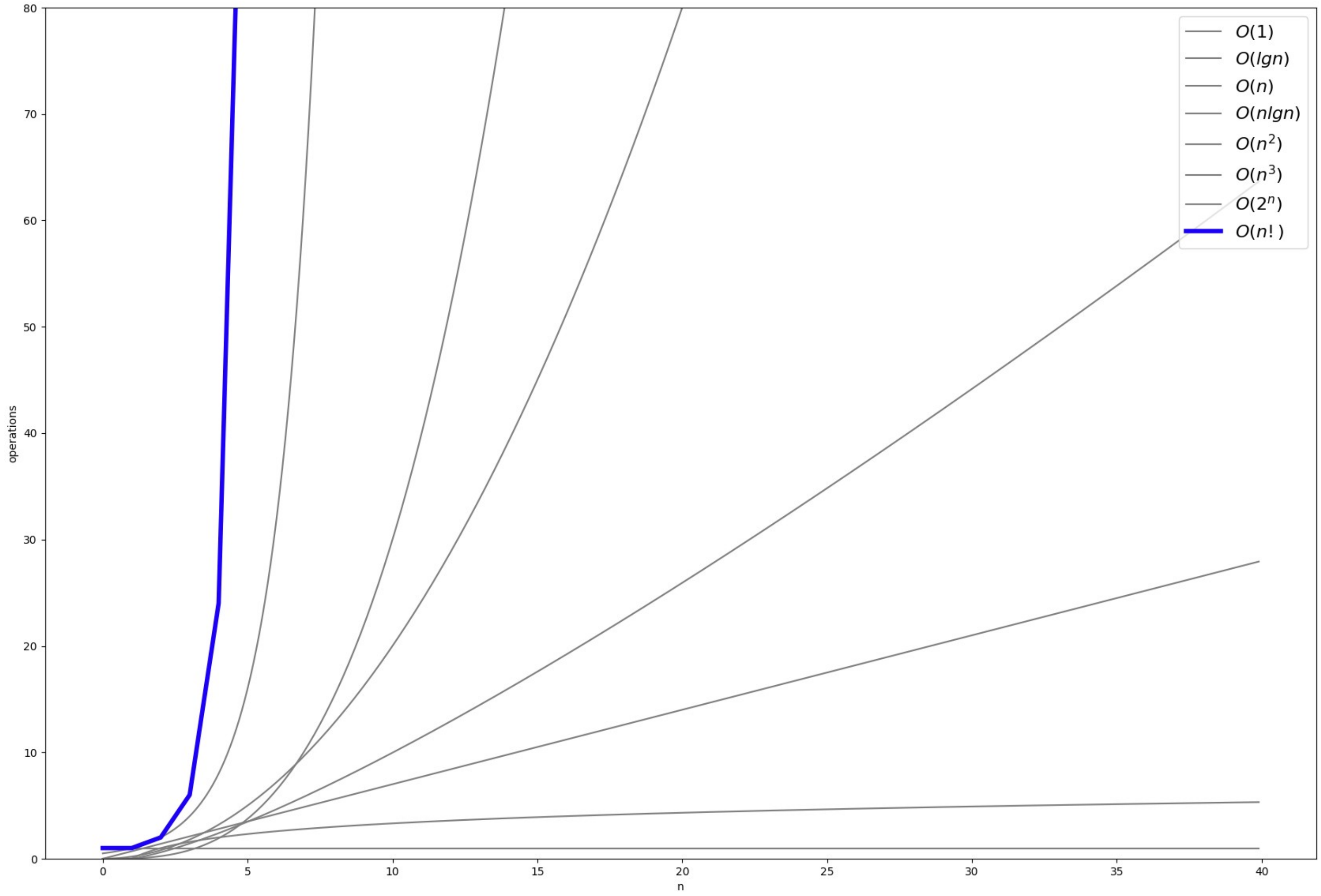
$t_3' = t_1 - t_4 - t_5 - t_6 + t_7 + t_9$

$O(f(n)) = O(t_1' \cdot 2^n + t_2' \cdot n + t_3') = O(2^n)$



P big_o/time/exponential_complexity_profile.py

Złożoność rzędu silnia – O(n!)


O(1)
O(lgn)
O(n)
O(nlgn)
O(n^2)
O(n^3)
O(2^n)
O(n!)
operations
n
0
10
20
30
40
50
60
70
80
0
5
10
15
20
25
30
35
40

złożoność rzędu silnia – O(n!)

Dla zbioru n-elementowego, tworzymy n możliwych podproblemów, z których każdy posiada n -1 elementowy zbiór danych wejściowych. Całkowita liczba możliwych rozwiązań wynosi n!

```
def some_function(n):
    counter = 1

    for m in range(n, 1, -1):
        counter *= m

        for _ in range(counter):
            ...

    return ...
```

złożoność rzędu silnia – O(n!)

zadanie – Znajdź wszystkie sposoby zjedzenia bombonierki. Każda pralina jest unikatowy, zatem celem zadania jest odnalezienie wszystkich permutacji bez powtórzeń zbioru n–elementowego.

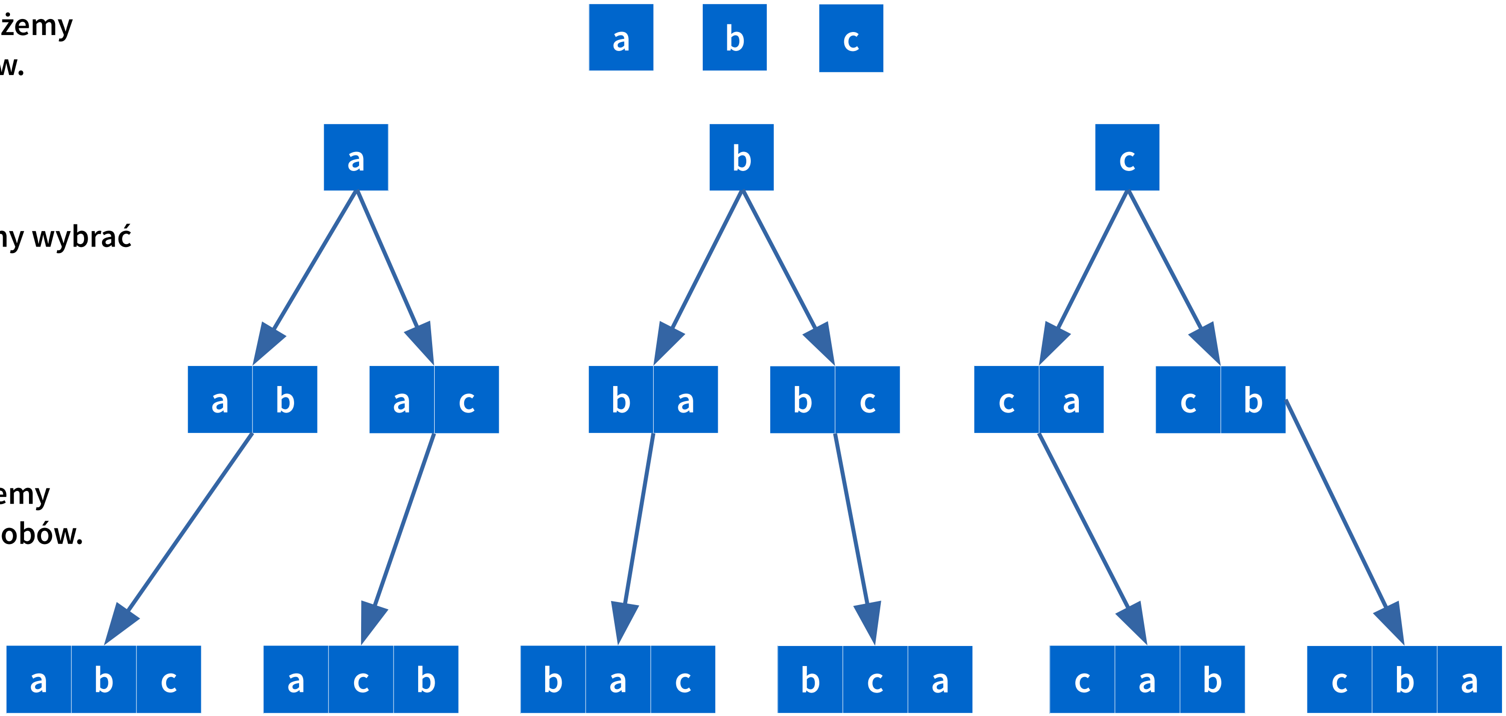
n – liczba pralin

Pierwszy element możemy wybrać na n sposobów.

Drugi element możemy wybrać na n - 1 sposobów.

.
.
.

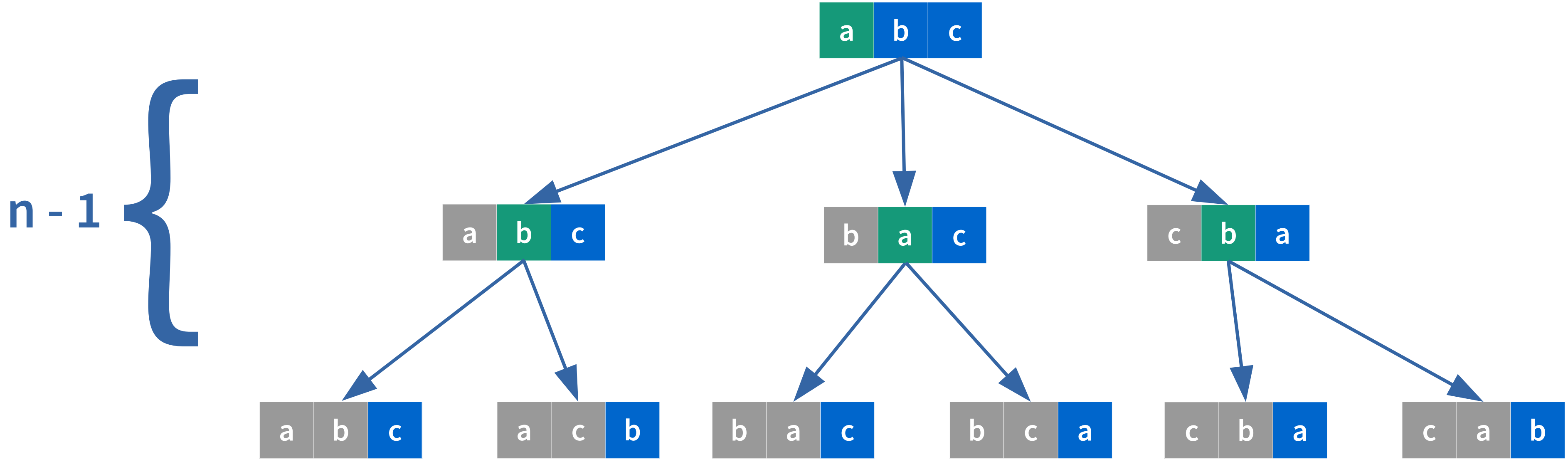
Ostatni element możemy wybrać na jeden sposobów.

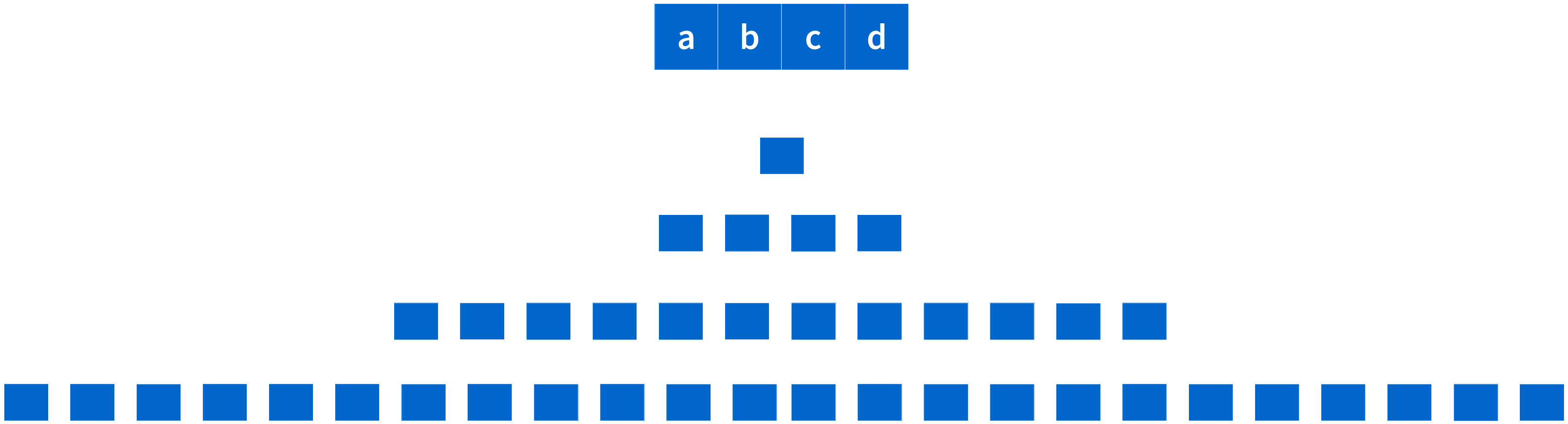


Liczba różnych ścieżek wynosi: $n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1 = n!$

- Zamieniamy miejscami z każdym elementem o indeksie równym bądź większym.

- Wyczerpane są wszystkie możliwości dla zaznaczonej długości podciągu.





iteracje:

I - 4

II – 4 + 4▪3

III – 4 + 4▪3 + 4▪3▪2

Liczba operacji kopiowania tablicy / zamiany pól:

$$4 + 4 * 3 + 4 * 3 * 2 = 4(1 + 3 + 3 * 2) = 4 * 3(\frac{1}{3} + 1 + 2) = 4 * 3 * 2(\frac{1}{6} + \frac{1}{2} + 1) = 4 * 3 * 2(\frac{1}{1} + \frac{1}{2} + \frac{1}{6})$$

$$n! * \sum_{k=1}^{n} \frac{1}{k!} = n! * (\sum_{k=1}^{n} \frac{1}{k!} + 1 - 1) = n! * (\sum_{k=0}^{n} \frac{1}{k!} - 1)$$

$$\sum_{k=0}^{\infty} \frac{1}{k!} = e$$

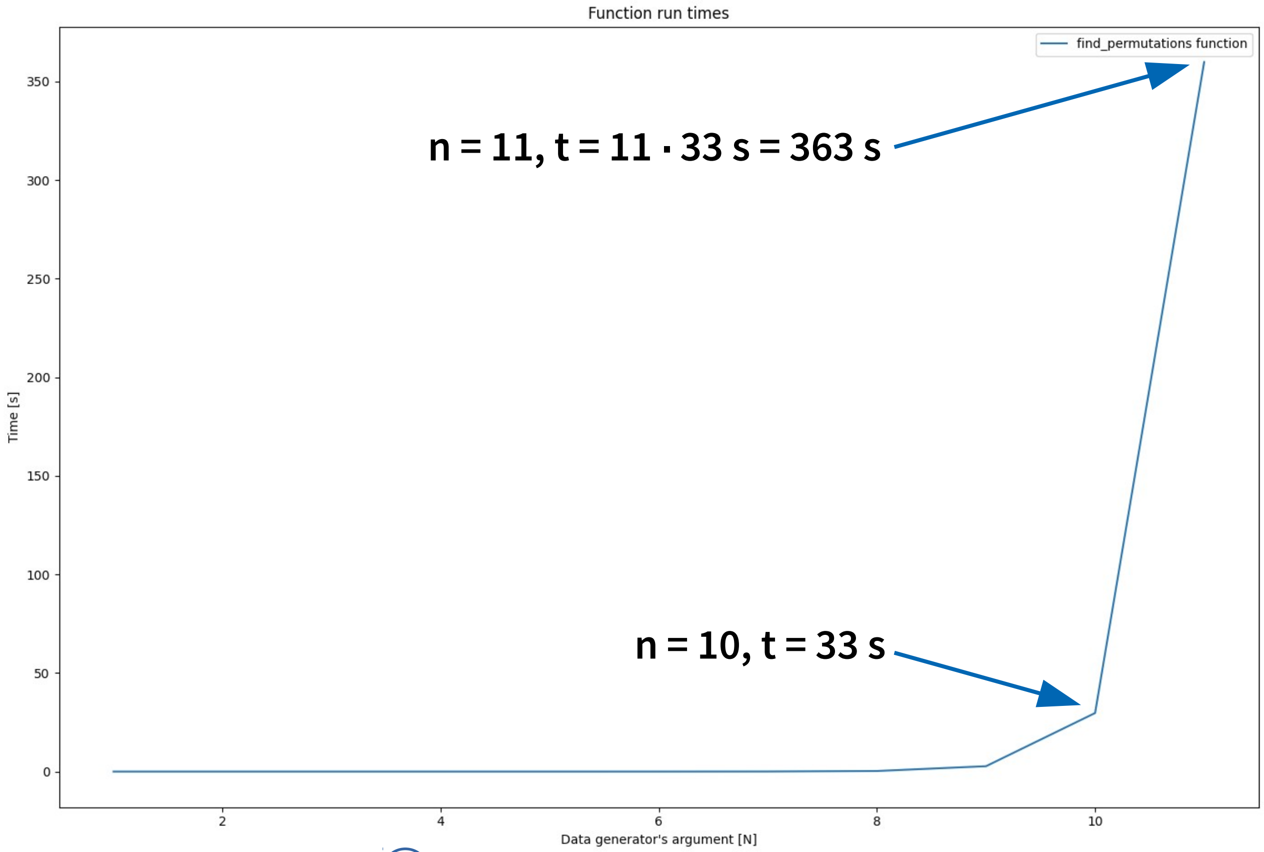
big_o/time/factorial_complexity.py	ilość wywołań	czas pojedyńczego wywołania
from copy import deepcopy		
def find_permutations(items):		
n = len(items[0])	1	t_1
for i in range(n - 1):	n - 1	t_2
new_items = []	n - 1	t_3
for item in items:	n!·(n - 1)	t_4
for j in range(i, n):	n!·(n - 1)	t_5
new_item = deepcopy(item)	n!·(n - 1)	t_6
temp = new_item[i]	n!·(n - 1)	t_7
new_item[i] = new_item[j]	n!·(n - 1)	t_8
new_item[j] = temp	n!·(n - 1)	t_9
new_items.append(new_item)	n!·(n - 1)	t_{10}
items = new_items	(n – 1)	t_{11}
return items	1	t_{12}

```python
from copy import deepcopy

def find_permutations(items):
    n = len(items[0])

    for i in range(n - 1):
        new_items = []

        for item in items:
            for j in range(i, n):
                new_item = deepcopy(item)
                temp = new_item[i]
                new_item[i] = new_item[j]
                new_item[j] = temp
                new_items.append(new_item)

        items = new_items

    return items
```

$f(n) = (t_4 + t_5 + t_6 + t_7 + t_8 + t_9 + t_{10}) \cdot n! \cdot (n - 1) + (t_2 + t_3 + t_{11}) \cdot (n - 1) + t_1 + t_{12}$

$t_1' = t_4 + t_5 + t_6 + t_7 + t_8 + t_9 + t_{10}$ $\quad t_2' = t_2 + t_3 + t_{11}$ $\quad t_3' = t_1 + t_{12}$

$O(f(n)) = O(t_1' \cdot n! \cdot (n - 1) + t_2' \cdot (n - 1) + t_3') = O(n!)$



P big_o/time/factorial_complexity_profile.py

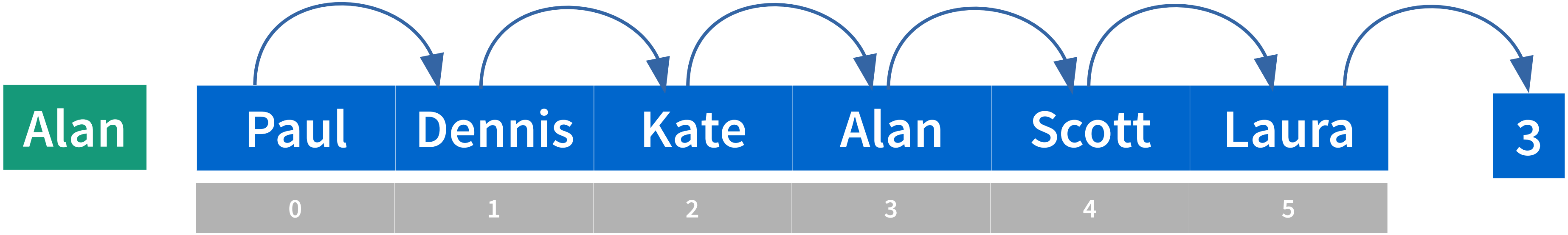
Pozostałe funkcje notacji asymptotycznej.

O – ograniczenie górne
Ω – ograniczenie dolne
Θ – ograniczenie dla konkretnego przypadku

Θ(f(n)) jest spełnione dla wszystkich n tylko gdy Θ(f(n)) = Ω(f(n)) = O(f(n))
Θ jest wtedy zarówno ograniczeniem dolnym jak i górnym.

Przykład I

zadanie – Znajdź w liście indeks podanego imienia używając wuszukiwania liniowego.
n – liczba elementów listy

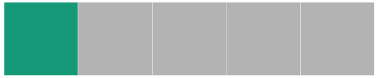


big_o/asymptotic_notation/linear_search.py

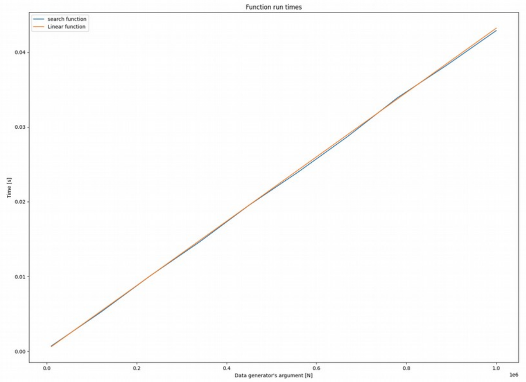
```
def search(names, wanted_name):
    wanted_index = False

    for index, name in enumerate(names):
        if name == wanted_name:
            wanted_index = index

    return wanted_index
```

Przypadek najlepszy

Pierwszy element listy jest elementem poszukiwanym.



$\Theta(n) = \Omega(n) = n$

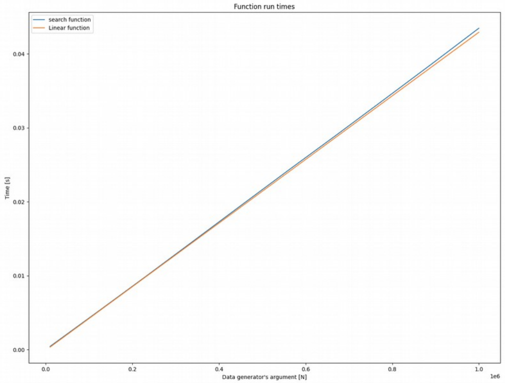


P big_o/asymptotic_notation/linear_search_best_case_profile

Przypadek średni

Środkowy element listy jest elementem poszukiwanym.

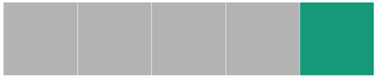
$\Theta(n) = n$



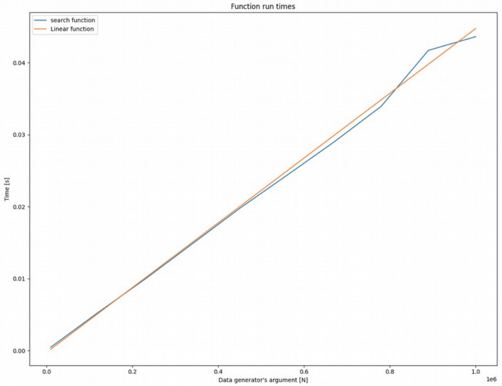
P big_o/asymptotic_notation/linear_search_average_case_profile

Przypadek najgorszy

Ostatni element listy jest elementem poszukiwanym.



$\Theta(n) = O(n) = n$

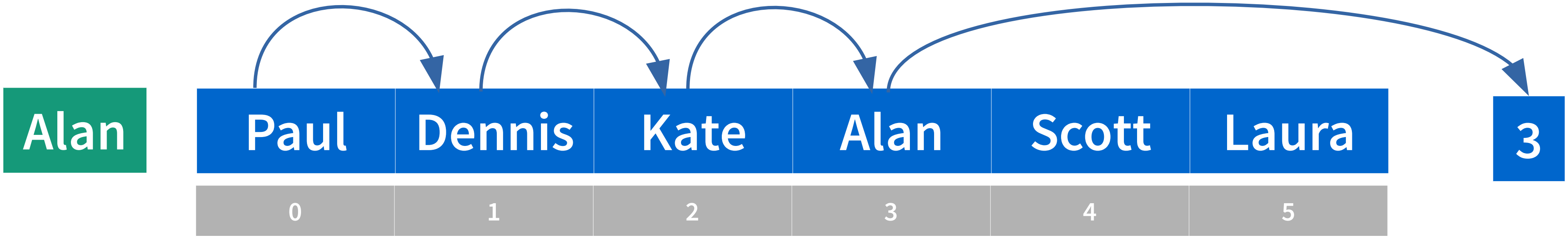


P big_o/asymptotic_notation/linear_search_worst_case_profile

$\Omega(n) = O(n)$ => dla każdego n - $\Theta(n)$

Przykład II

zadanie – Znajdź w liście indeks podanego imienia używając usprawnionego wuszukiwania liniowego.
n – liczba elementów listy

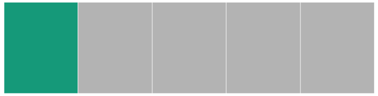


<> big_o/asymptotic_notation/better_linear_search.py

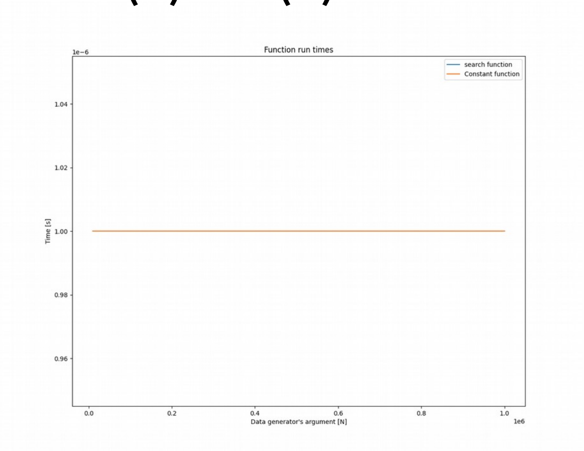
```
def search(names, wanted_name):
    for index, name in enumerate(names):
        if name == wanted_name:
            return index

    return False
```

Przypadek najlepszy

Pierwszy element listy jest elementem poszukiwanym.



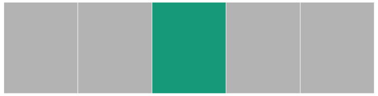
$\Theta(1) = \Omega(1) =$ stała



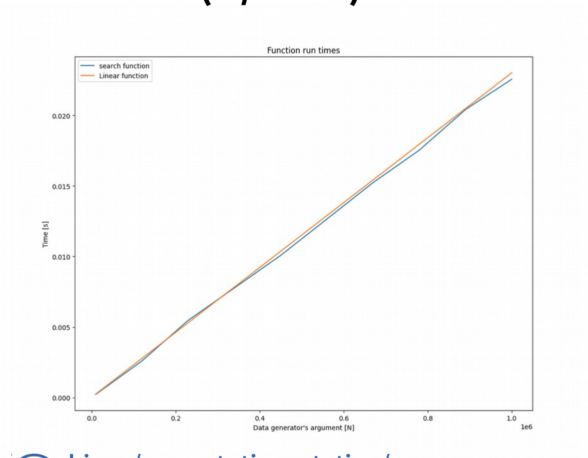
P big_o/asymptotic_notation/better_linear_search_best_case_profile.py

Przypadek średni

Środkowy element listy jest elementem poszukiwanym.



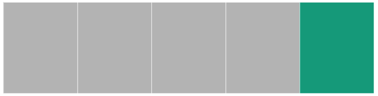
$\Theta(1/2 \cdot n) = n$



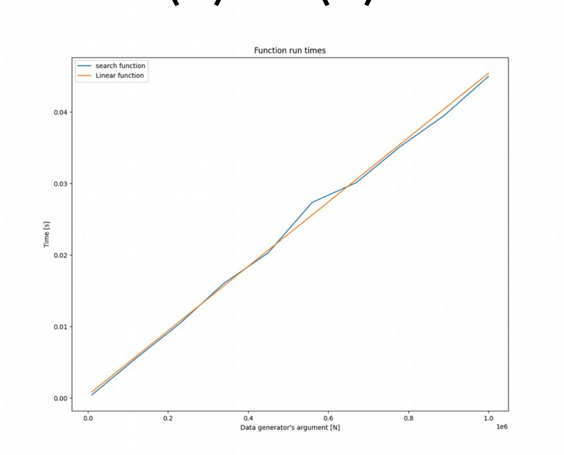
P big_o/asymptotic_notation/better_linear_search_average_case_profile.py

Przypadek najgorszy

Ostatni element listy jest elementem poszukiwanym.



$\Theta(n) = O(n) = n$



P big_o/asymptotic_notation/better_linear_search_worst_case_profile.py

Użycie pamięci jako funckji czasu.

Wielkość wykorzystywanej pamięci, w przeciwieństwie do czasu, może nie tylko rosnąć ale również maleć w trakcie wykonywania algorytmu.

Badanie zmian wielkości użytej pamięci w czasie dla zadanej wartości n.

```
profiler = Profiler()

profiler.run_time_based_memory_usage(
    func=some_function,
    kwargs=data_gen(100),
    interval=0.001
)
```

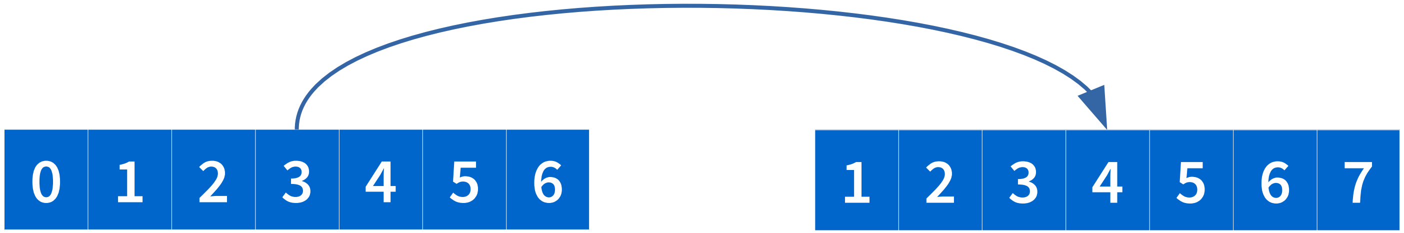
Uruchom podaną funckję i przedstaw pomiary na wykresie.

Wygeneruj dane wejściowe dla n = 100.

Badaj zużycie pamięci co 0.001 sekundy.

Przykład

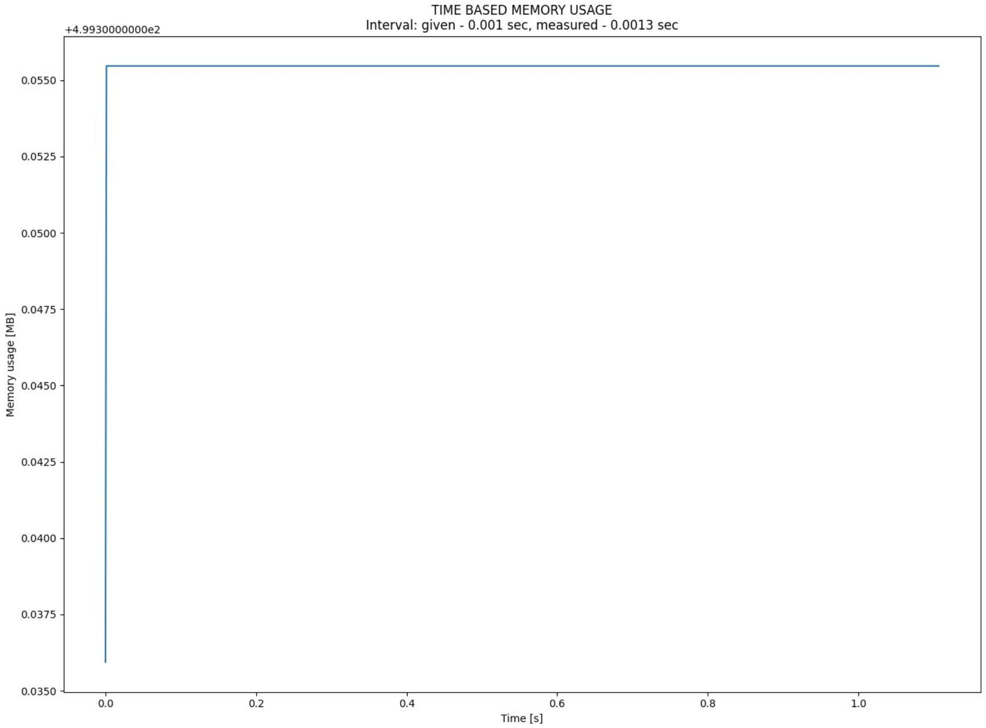
zadanie – Zwiększ o jeden każdy element listy.
n = 10000000 – liczba elementów listy



Implemecja I – stałe żuzycie pamięci w czasie

<> big_o/memory/constant_memory_consumption.py

```python
def increment_by_one(numbers_list):
    for index, number in enumerate(numbers_list):
        numbers_list[index] += 1

    return numbers_list
```



P big_o/memory/constant_memory_consumption_profile.py

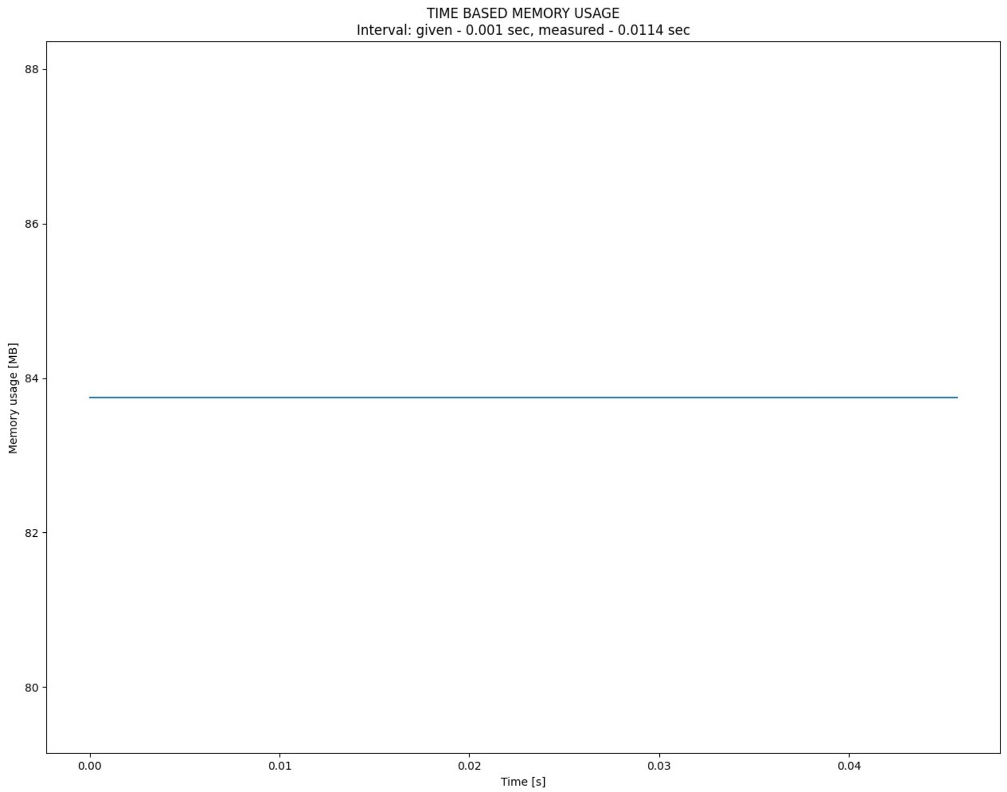
```python
def increment_by_one(numbers_list):
    for index, number in enumerate(numbers_list):
        numbers_list[index] += 1

    return numbers_list
```

! Operacje wykonujemy “na miejscu”. Nie kopiujemy danych wejściowych lecz mutujemy listę przekazaną jako argument funckji.

Implemecja II – stałe żuzycie pamięci w czasie

big_o/memory/constant_memory_consumption_with_recursion.py

```python
def increment_by_one(numbers_list, index=0):
    if index < len(numbers_list):
        numbers_list[index] += 1
        return increment_by_one(numbers_list, index + 1)
    else:
        return numbers_list
```



big_o/memory/constant_memory_consumption_profile_with_recursion.py

```python
def increment_by_one(numbers_list, index=0):
  if index < len(numbers_list):
    numbers_list[index] += 1
    return increment_by_one(numbers_list, index + 1)
  else:
    return numbers_list
```

numbers_list zawiera referencję do tego samego obiektu w pamięci

0	1	2

```python
def increment_by_one(numbers_list, index=1):
  if index < len(numbers_list):
    numbers_list[index] += 1
    return increment_by_one(numbers_list, index + 1)
  else:
    return numbers_list
```

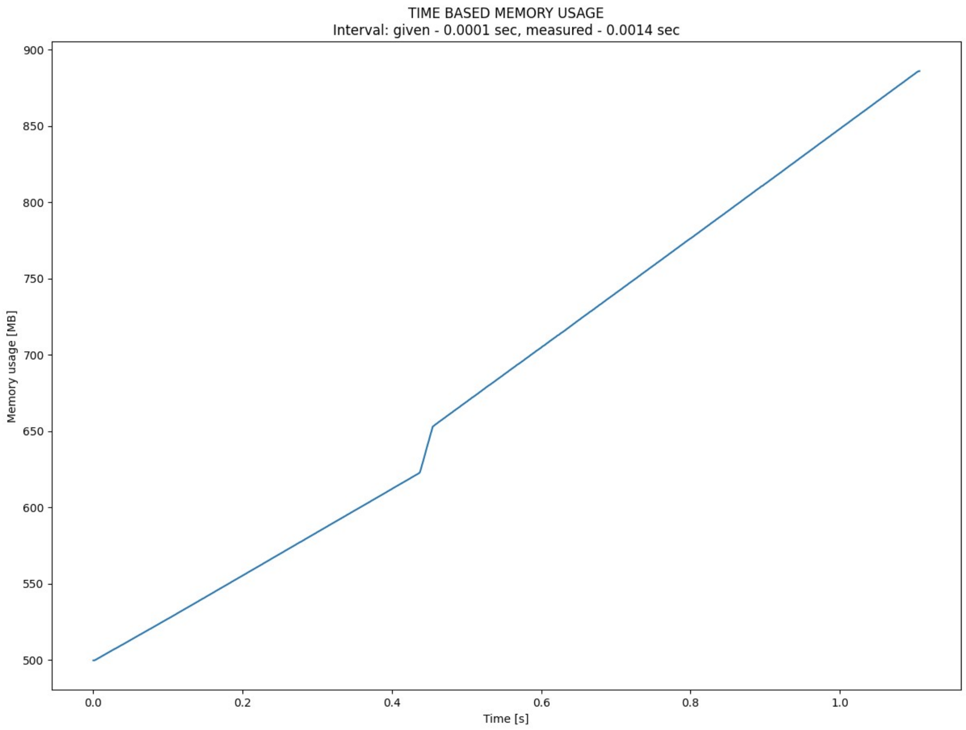
```python
def increment_by_one(numbers_list, index=2):
  if index < len(numbers_list):
    numbers_list[index] += 1
    return increment_by_one(numbers_list, index + 1)
  else:
    return numbers_list
```

```python
def increment_by_one(numbers_list, index=3):
  if index < len(numbers_list):
    numbers_list[index] += 1
    return increment_by_one(numbers_list, index + 1)
  else:
    return numbers_list
```

Implemecja III – rosnące żuzycie pamięci w czasie

<> big_o/memory/growing_memory_consumption.py

```
def increment_by_one(numbers_list):
    incremented_list = []

    for number in numbers_list:
        incremented_number = number + 1
        incremented_list.append(incremented_number)

    return incremented_list
```

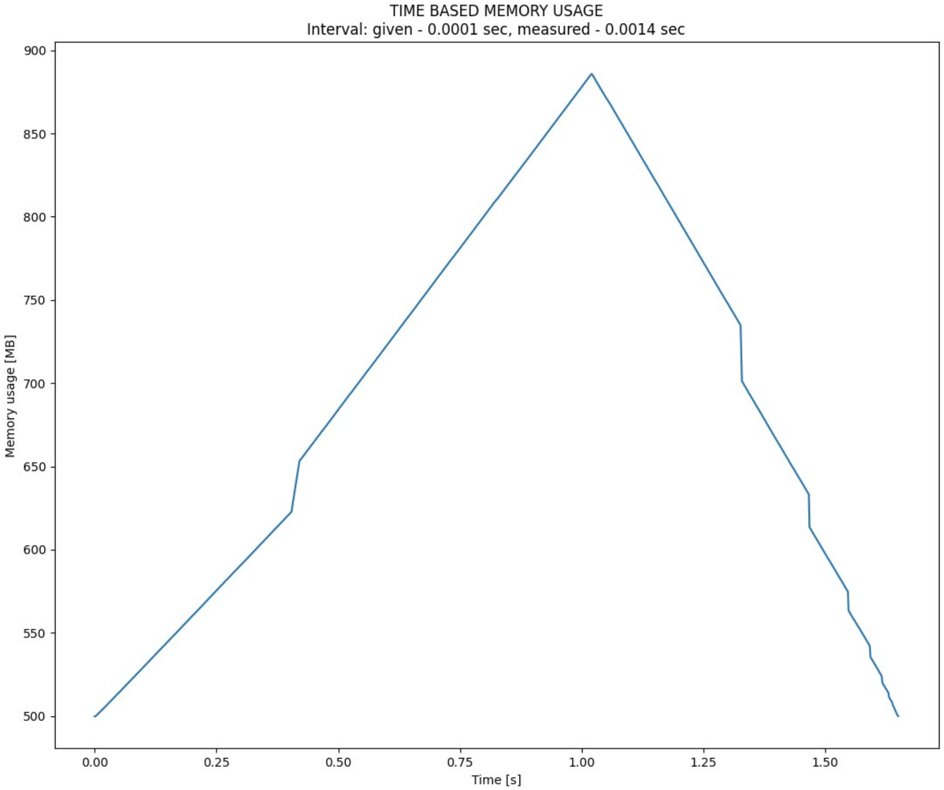


P big_o/memory/growing_memory_consumption_profile.py

Implemecja IV – flutkuujące żuzycie pamięci w czasie

big_o/memory/fluctuating_memory_consumption.py

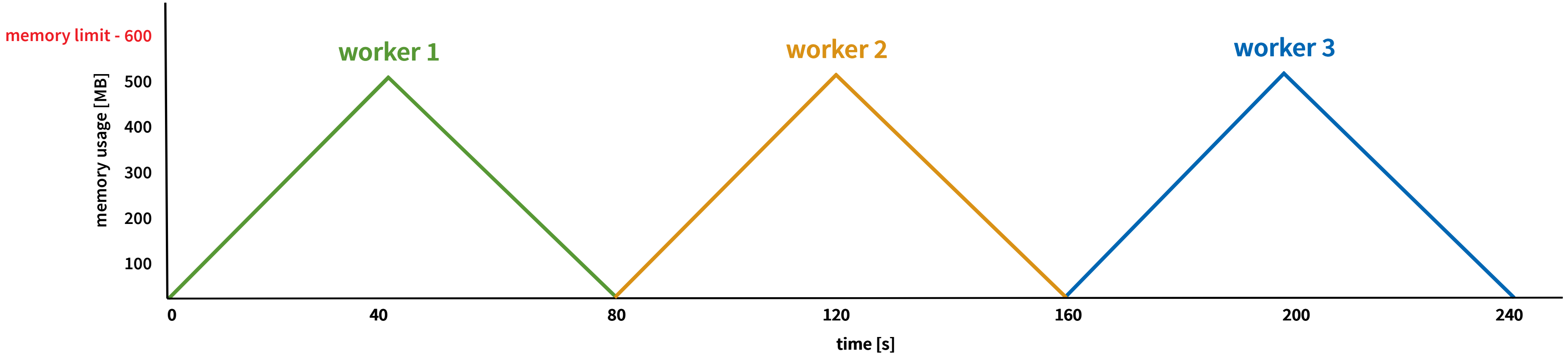
```python
def increment_by_one(numbers_list):
    incremented_list = []

    for number in numbers_list:
        incremented_number = number + 1
        incremented_list.append(incremented_number)

    for _ in range(len(numbers_list)):
        numbers_list.pop()

    return incremented_list
```



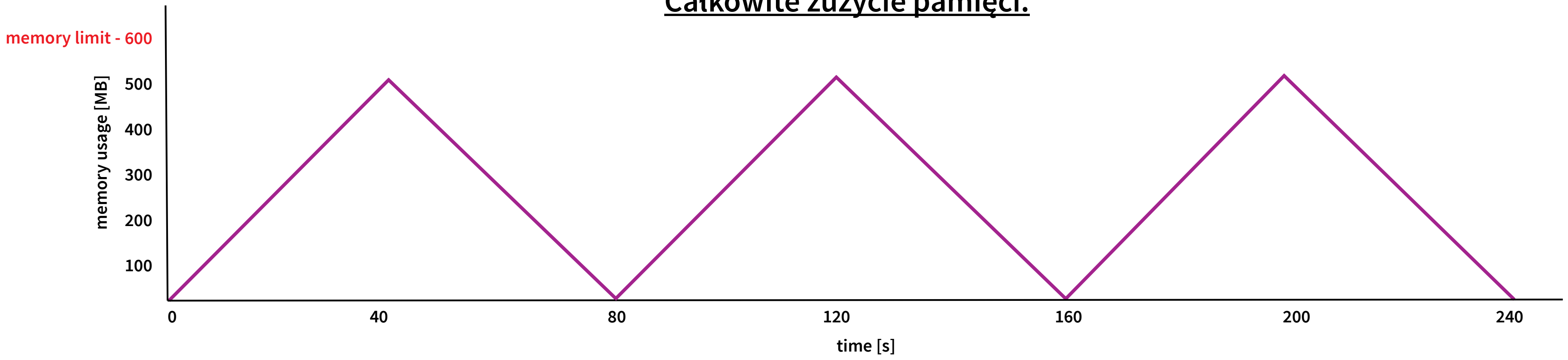
big_o/memory/fluctuating_memory_consumption_profile.py

Złożoność pamięciową obliczamy na podstawie maksymalnego chwilowego wykorzystania pamięci.

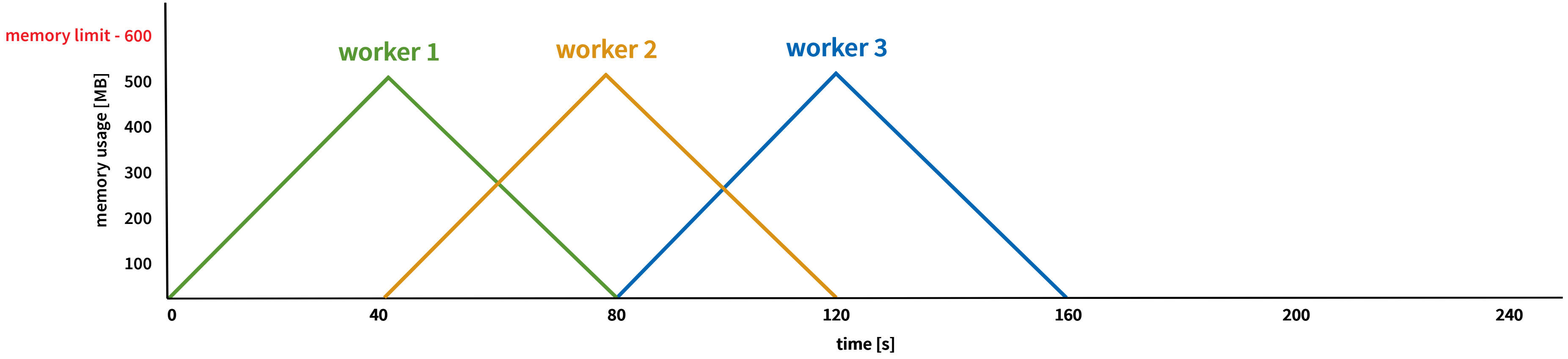
Szczegółowe zużycie pamięci.



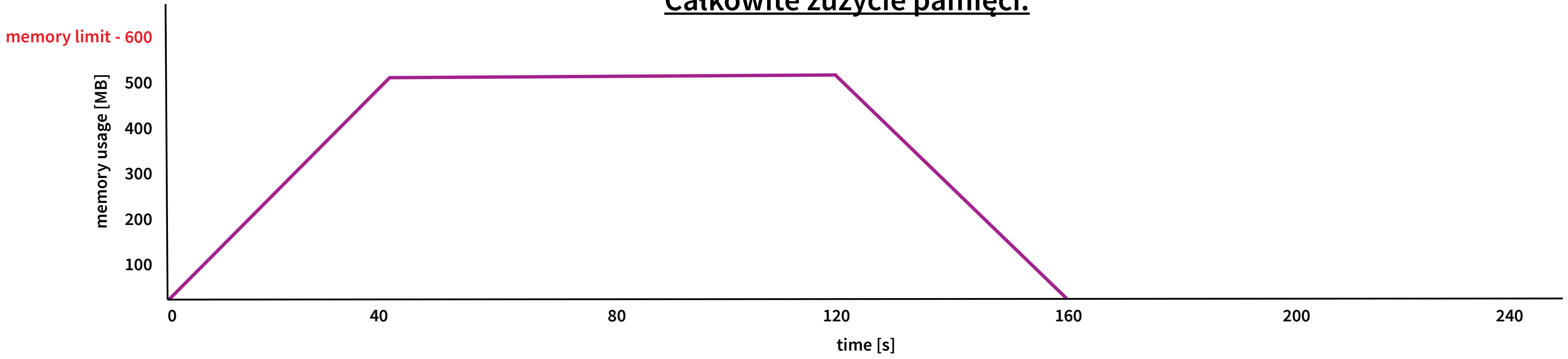
Całkowite zużycie pamięci.



Szczegółowe zużycie pamięci.



Całkowite zużycie pamięci.



Metodyka analizy złożoności pamięciowej algorytmów.

Schemat badania złożoności pamięciowej algorytmu.

zadanie – ...
n – ...

1. Analiza liczby wywołań operacji alokujących nowe obiekty w pamięci.

	ilość alokacji	wielkość alokowanej pamięci
`def some_function(**kwargs):`		
`    ...`	n	t_1
`    ...`	n^2	t_2
`    return ...`	1	t_3

2. Wyprowadzenie funkcji w notacji asymptotycznej.

t_1, t_2, t_3 - stałe

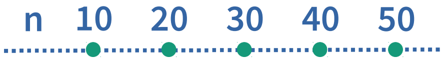
$$f(n) = t_1 \cdot n + t_2 \cdot n^2 + t_3$$

$$O(f(n)) = O(t_1 \cdot n + t_2 \cdot n^2 + t_3) = O(n^2)$$

3 Stworzenie generatora danych wejściowych zależnego od zmiennej n.

```python
def data_gen(n):
    ...
    return {"argument": argument}
```

4 Wykonanie pomiarów dla zadanych wartości n oraz analiza otrzymanychrezultatów.




```python
profiler = Profiler()

profiler.run_memory_analysis(
  func=some_function,
  data_gen=data_gen,
  gen_min_arg=10,
  gen_max_arg=50,
  gen_steps=5,
  find_big_o=True
)
```

Znajdź funkcję, która dokonuje najlepszej predykcji wyników.

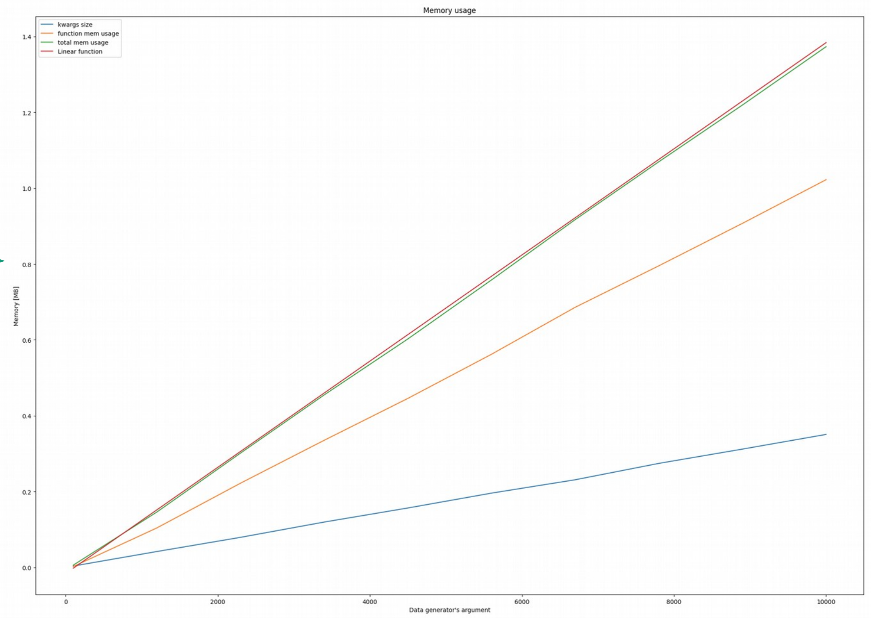
n = 10, 20, 30, 40

data_gen(n)

zapisz rezultat

some_list

some_function(some_list)




Memory usage
kwargs size
function mem usage
total mem usage
Linear function
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Memory [MB]
0
2000
4000
6000
8000
10000
Data generator's argument
Funkcja, która dokonuje najlepszej predykcji wyników.
Suma wielkość danych wejściowych oraz wygenerowanych.
Wielkość pamięci alokowanej przez badaną funkcje.
Wielkość danych wejściowych.

Rodzaje funckji.

Stała złożoność - O(1)

Stała złożoność - O(1)

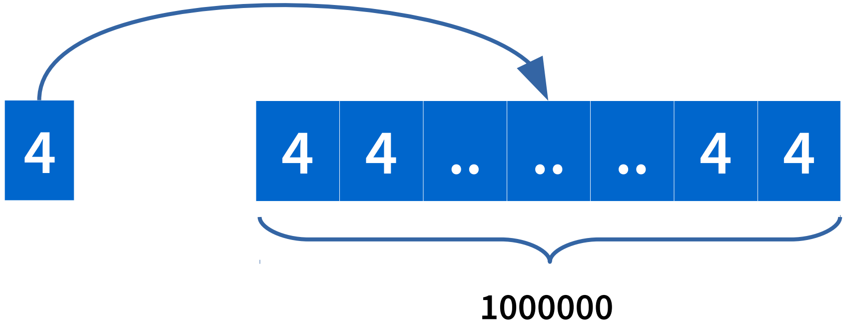
Ilość oraz wielkość danych wejściowych nie zmienia się.
Nie występuje parametr n.

Stała liczba operacji alokujących stałą wielkość pamięci.

```
def some_function(char_1='a', char_2='b'):
    uppercase_char_1 = char_1.upper()
    uppercase_char_2 = char_2.upper()
    uppercase_string = uppercase_char_1 + uppercase_char_2
    ...
    ...
    ...
    class_instance_1 = SomeClass1()
    class_instance_2 = SomeClass2()
```

Stała złożoność - O(1)

zadanie – Zainicjuj listę o długości 1000000 przez ustawienie wszystkich komórek na zadaną cyfrę.



big_o/memory/constant_complexity.py	ilość alokacji	wielkość alokowanej pamięci
`def init_list(digit):`		
`    initiated_list = []`	1	t_1
`    for _ in range(1000000):`	1	t_2
`        initiated_list.append(digit)`	1000000	t_3
`    return initiated_list`		

Stała złożoność - O(1)

t_1, t_2, t_3 - stałe

$f() = 1000000 \cdot t_1 + t_2 + t_3$

$O(f()) = O(1000000 \cdot t_1 + t_2 + t_3) = O(1)$

MEMORY CHECK

Function: init_list(digit)

digit 0.0 MB: 1

Kwargs size: 0.0002 MiB
Function usage: 8.2946 MiB

Total usage: 8.2948 MiB

MEMORY CHECK

Function: init_list(digit)

digit 0.0 MB: 9

Kwargs size: 0.0002 MiB
Function usage: 8.2946 MiB

Total usage: 8.2948 MiB

P big_o/memoryconstant_complexity_profile.py

Złożoność logarytmiczna – O(lg n)

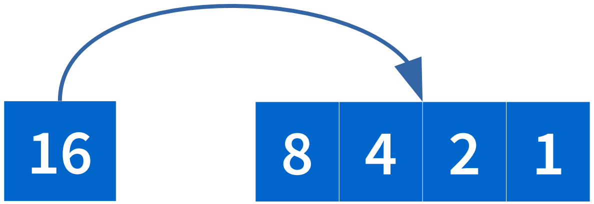
Złożoność logarytmiczna – O(lg n)

Podczas każdego podziału zbioru wejściowego na połowe, otrzymujemu część wyniku, który należy zapamiętać.

```
def some_function(n):
    result = []

    while n >= 1:
        ...
        result.append(...)
        n /= 2

    return ...
```

Złożoność logarytmiczna – O(lg n)

zadanie – Znajdź wszystkie wyniki dzielenia zadanej liczby przez kolejne potęgi liczby dwa.
Minimalna wartość operacji dzielenia wynosi jeden.
n – liczba naturalna

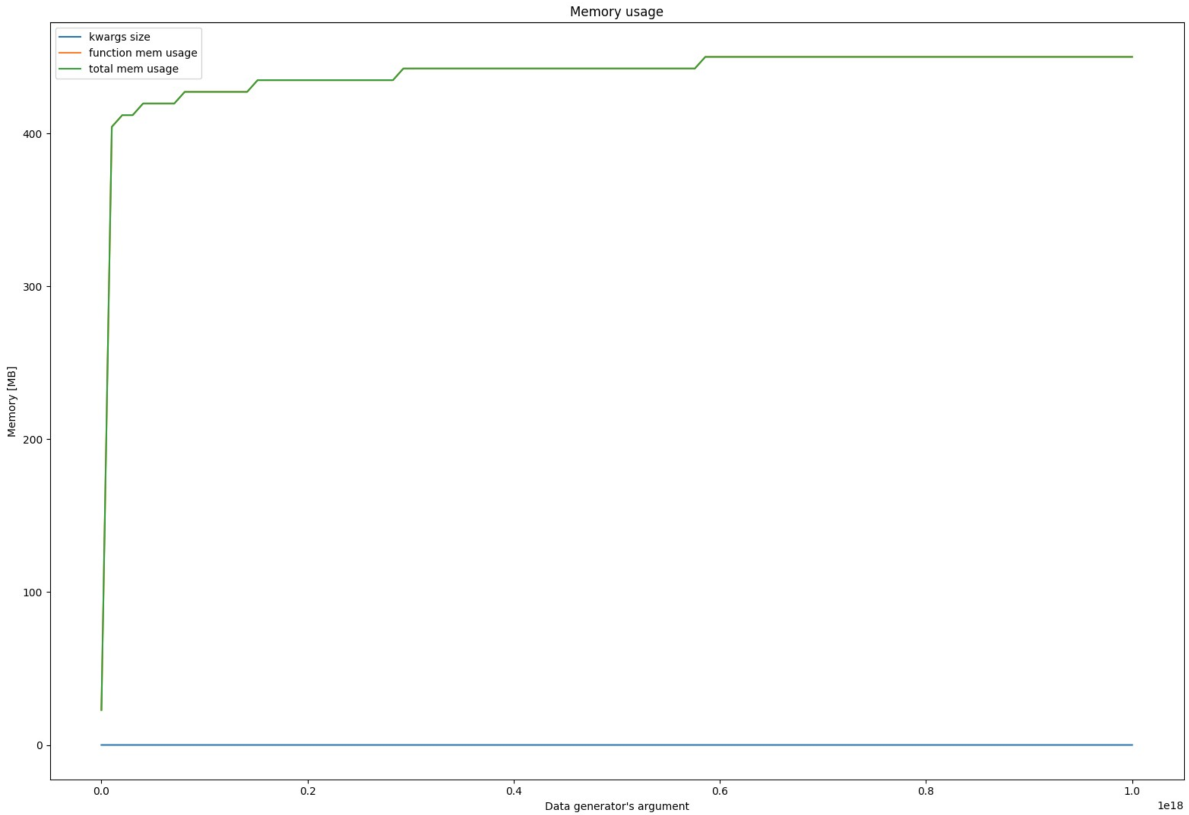


big_o/memory/logarithmic_complexity.py	ilość alokacji	wielkość alokowanej pamięci
`def check_divisions(number):`		
`    dividers = []`	1	t_1
`    number /= 2`	1	t_2
`    while number >= 1:`		
`        dividers.append(number)`	lg n	t_3
`        number /= 2`		
`    return dividers`		

$$f(n) = t_1 + t_2 + t_3 \cdot \lg n$$

$$t_1' = t_1 + t_2$$

$$O(f(n)) = O(t_3 \cdot \lg n + t_1') = O(\lg n)$$



Ⓟ big_o/memory/logarithmic_complexity_profile.py

Złożoność liniowa - O(n)

Złożoność liniowa - O(n)

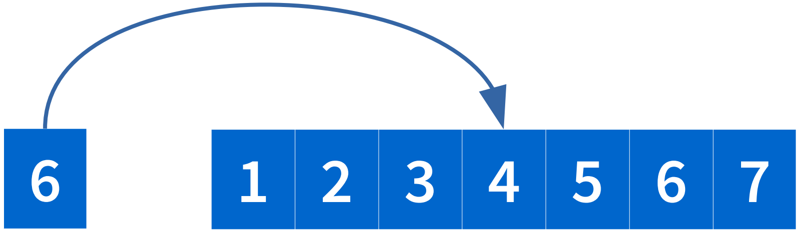
Liczba alokacji pamięci jest liniowo zależna od wielkości zbioru n.

```
def some_function(n):
    some_list = []

    for ... in range(n):
        ...
        some_list.append(...)

    return ...
```

```
def some_function(n):
    if n > 0:
        ...
        result = ...
        return result + some_function(n - 1)
    else:
        return  ...
```

złożoność liniowa - O(n)

zadanie – Zainicjuj listę kolejnymi liczbami naturalnymi.
n – liczba elementów listy

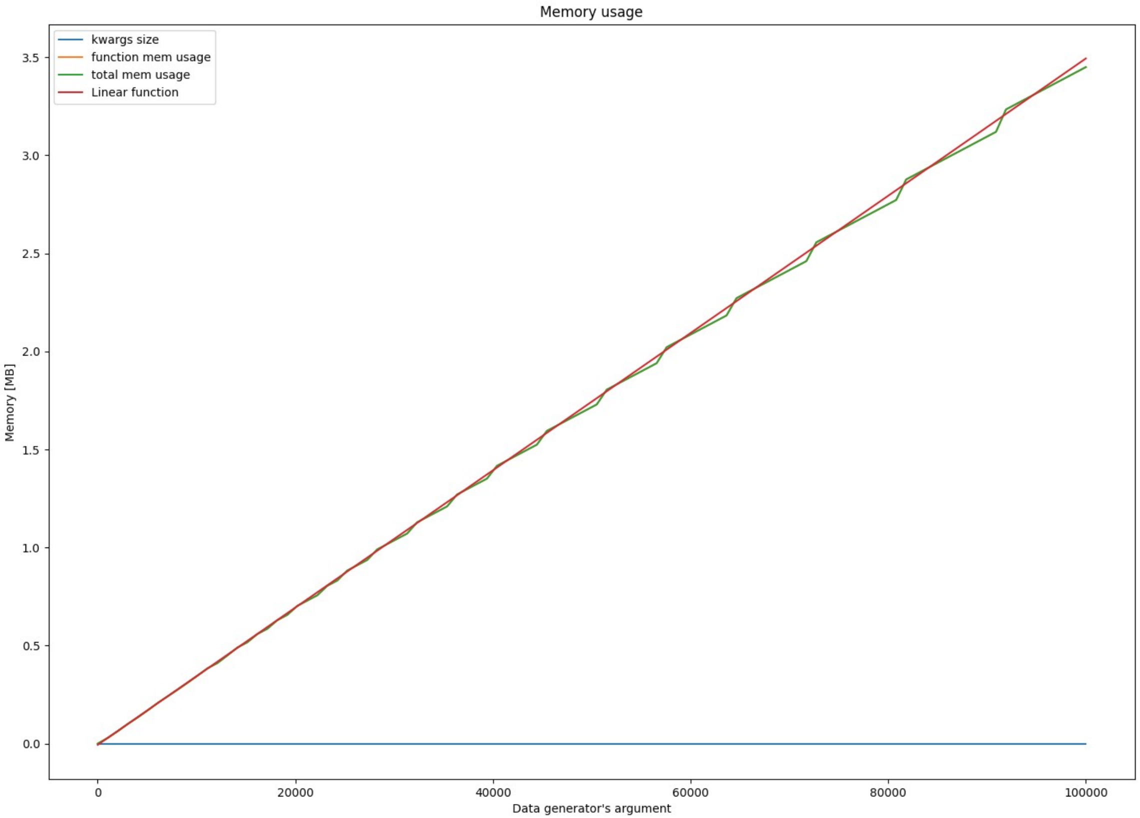


big_o/memory/linear_complexity.py	ilość wywołań	czas pojedynczego wywołania
def init_list(list_length):		
initiated_list = []	1	t_1
for number in range(list_length):	1	t_2
initiated_list.append(number)	n	t_3
return initiated_list		

$$f(n) = t_1 + t_2 + t_3 \cdot n$$

$$t_1' = t_1 + t_2$$

$$O(f(n)) = O(t_3 \cdot n + t_1') = O(n)$$



P big_o/memory/linear_complexity_profile.py