laboratoria_lab1_PAPRO_lab1_instr_wstępna

March 7, 2021

1 Paradygmaty programowania

2 Laboratorium 1 – badanie wydajności wybranych algorytmów

Pierwsze laboratorium jest poświęcone określaniu złożoności obliczeniowej i czasowej algorytmów. Będziemy szukać odpowiedzi na pytanie, jak "skaluje się" dana metoda rozwiązywania jakiegoś problemu: czy dla problemu dwukrotnie większego od dotychczasowego nakład obliczeń wzrośnie dwu- czy czterokrotnie, a może o jakiś inny, trudny do przewidzenia czynnik? Analizę teoretyczną złożoności można przeprowadzić na podstawie znajomości zasady działania algorytmu. W praktyce jednak złożoność może zależeć od implementacji algorytmu i od użytych struktur danych. Dlatego analizy teoretyczne należy zawsze poprzeć eksperymentem. Na początek nauczymy się więc korzystać z dostępnego w języku Python narzędzia do pomiaru czasu.

2.1 Pomiar czasu w języku Python

Czas wykonania fragmentu kodu lub całego programu w języku Python łatwo zmierzyć za pomocą funkcji `timeit` oraz `repeat`, zdefiniowanych w module `timeit`. Mierzony jest czas potrzebny na `number`-krotne wykonanie kodu przekazanego w cudzysłowach jako `stmt`. Domyślna wartość parametru `number` to milion. Oto przykład wywołania:

[1]:
```
import timeit
timeit.timeit(stmt='["a"]*1000000', number=100)
```

[1]:
```
0.3152790990000085
```

W naszym przykładzie zmierzyliśmy czas potrzebny na stukrotną alokację listy o milionie elementów i każdorazowe wypełnienie jej literami `a`. (W rzeczywistości są to listy jednoliterowych łańcuchów, ponieważ w języku Python nie ma osobnego typu danych odpowiadającego pojedynczemu znakowi.)

Jeśli w wykonywanym wielokrotnie kodzie zamierzamy odwoływać się do zdefiniowanych uprzednio zmiennych lub funkcji, możemy postępić dwojako. Krótki ciąg poleceń, oddzielonych od siebie średnikami, możemy przekazać jako parametr wywołania `setup`. Przekazane polecenia zostaną wykonane jednokrotnie przed rozpoczęciem pomiaru czasu. W poniższym przykładzie `setup` służy do wyświetlenia komunikatu o rozpoczęciu pomiaru i do inicjalizacji licznika powtórzeń `c`.

[2]:
```
timeit.timeit(setup='print("Zaczynamy pomiar czasu:"); c=1', stmt='print(c);␣
↪c+=1', number=3)
```

```
Zaczynamy pomiar czasu:
1
2
3
```

[2]:
```
6.737999996175859e-05
```

W kodzie, którego czas wykonania mierzymy, możemy też odwoływać się do wszystkich zmiennych i funkcji dostępnych we wcześniejszych partiach programu. Wystarczy użyć dodatkowego argumentu `globals`, jak w poniższym przykładzie:

[3]:
```
x = 42

def divide_by_two(z):
    return z//2

print(timeit.timeit(stmt='print(divide_by_two(x))', number=4,␣
 ↪globals=globals()))
```

```
21
21
21
21
0.0004259380000348756
```

Na zmierzony czas mogą wpływać inne procesy wykonywane równocześnie w systemie. Dlatego pomiar czasu warto wykonać kilkukrotnie, a następnie wybrać najmniejszy z zaobserwowanych czasów. Za kilkukrotne uruchomienie pomiaru odpowiada funkcja pomocnicza `repeat`. Różni się ona od funkcji `timeit` jedynie dodatkowym argumentem (nazwanym również `repeat`), decydującym o liczbie takich uruchomień. Domyślna wartość tego argumentu to 3 lub 5, w zależności od wersji interpretera języka Python. Przed każdym pomiarem jest wykonywana operacja `setup`. Zwracana jest lista zmierzonych czasów.

[4]:
```
times = timeit.repeat(setup='print("Kolejne wywołanie timeit"); c=1',␣
 ↪stmt='print(c); c+=1', number=2, repeat=3, globals=globals())
print("Lista czasów wykonania:", times)
print("Najkrótszy czas wykonania:", min(times))
```

```
Kolejne wywołanie timeit
1
2
Kolejne wywołanie timeit
1
2
Kolejne wywołanie timeit
1
2
Lista czasów wykonania: [6.061699997417236e-05, 4.750400000830268e-05,
6.652799999073977e-05]
```

```
Najkrótszy czas wykonania: 4.750400000830268e-05
```

Należy pamiętać, że sama procedura mierzenia czasu za pomocą funkcji `timeit` również wymaga pewnej liczby cykli zegarowych. Można się o tym przekonać, wykonując następujący kod, w którym w ogóle nie występuje `stmt`:

[5]:
```
timeit.timeit(number=1)
```

[5]:
```
1.2720000199806236e-06
```

Zapewne zaobserwowaliśmy czas na poziomie pojedynczych mikrosekund. Załóżmy, że pojedyncze wykonanie operacji, której czas trwania chcemy zmierzyć, zajmuje również mikrosekundę. Wtedy pomiar czasu jej trwania będzie obarczony błędem bliskim stu procent. Na szczęście czas wykonania "pustej" operacji `timeit` nie skaluje się liniowo z liczbą powtórzeń, o czym przekonamy się, zwiększając tę liczbę z jednego do miliona:

[6]:
```
timeit.timeit(number=1000000)
```

[6]:
```
0.008601741000006768
```

Tym razem zaobserwowaliśmy prawdopodobnie czas rzędu setnych części sekundy. Wykonanie naszej "mikrosekundowej" operacji milion razy potrwa oczywiście około sekundy, zatem błąd pomiaru czasu zostanie zredukowany do około procenta.

> **UWAGA** W pewnych sytuacjach nie możemy sobie pozwolić na wielokrotne powtórzenie operacji, której czas wykonania mierzymy. Bywa tak przede wszystkim wtedy, gdy operacja ta modyfikuje stan struktury danych, na której operuje, np. wstawia do niej element lub go usuwa. Wtedy przy każdym powtórzeniu `timeit` mamy do czynienia ze strukturą danych o innym rozmiarze, zatem czas wykonania każdego z powtórzeń może być inny. W takiej sytuacji należy ograniczyć się do pojedynczego wywołania `timeit`, czyli zdefiniować `number=1`. Bezpośrednio przed takim pomiarem należy zmierzyć czas wykonania "pustej" operacji `timeit`, a następnie odjąć go od wyniku.

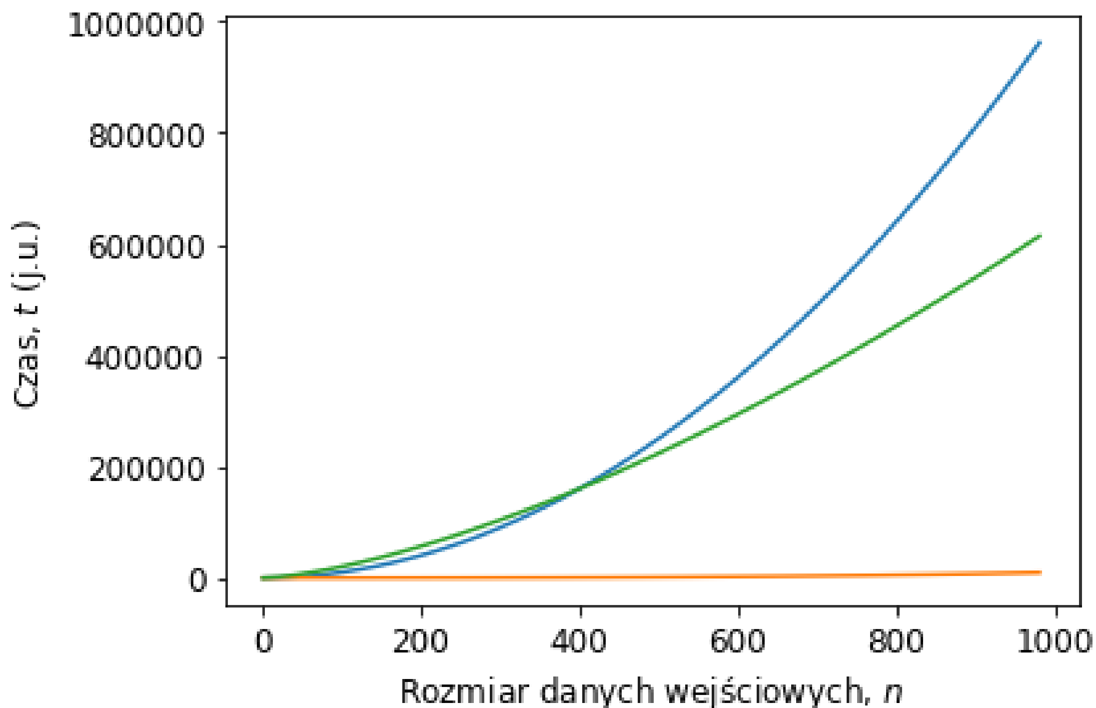
2.2 Określanie złożoności czasowej algorytmu

Aby określić złożoność czasową algorytmu, należy zmierzyć czas jego wykonania dla pewnego zakresu rozmiaru danych wejściowych. Zakres ten powinien być jak największy. Z reguły jest on ograniczony od dołu przez rozdzielczość i dokładność pomiaru czasu (o czym pisaliśmy w poprzednim punkcie), zaś od góry przez naszą cierpliwość.

Po zmierzeniu czasu należy go wykreślić w funkcji rozmiaru danych wejściowych i spróbować określić, jaką funkcją najlepiej można przybliżyć uzyskany wykres. Popatrzmy na wykres generowany przez poniższy kod (punkty danych nie pochodzą tak naprawdę z pomiaru czasu, lecz są generowane analitycznie).

[7]:
```
import matplotlib as mpl
from matplotlib import pyplot as plt
import locale
import platform
if platform.system()=='Windows':
```

```
    locale.setlocale(locale.LC_NUMERIC, "Polish")
else:
    locale.setlocale(locale.LC_NUMERIC, "pl_PL.utf8")
mpl.rcParams['axes.formatter.use_locale'] = True

n = range(1, 1000, 20)
t1 = [ x**2 for x in n ]
t2 = [ (1e-5)*x**3 for x in n ]
t3 = [ 20*x**1.5 for x in n ]

plt.plot(n, t1)
plt.plot(n, t2)
plt.plot(n, t3)

plt.xlabel('Rozmiar danych wejściowych, $n$', fontsize=12)
plt.ylabel('Czas, $t$ (j.u.)', fontsize=12)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.tight_layout()
plt.show()
```



Najczęściej pojawiające się pod takim wykresem wnioski brzmią: “Widzimy dwie parabole i funkcję stałą bliską zeru”. Skoro podejrzewamy, że mamy do czynienia z parabolami, spróbujmy to potwierdzić. Użyjemy przy tym następującego przekształcenia.

Wyraźmy czas t jako następujący jednomian:

$t \; = \; Cn^k,$

gdzie n jest rozmiarem danych, zaś C i k pewnymi stałymi. Po zlogarytmowaniu stronami przy podstawie 10 (lub dowolnej innej) otrzymujemy

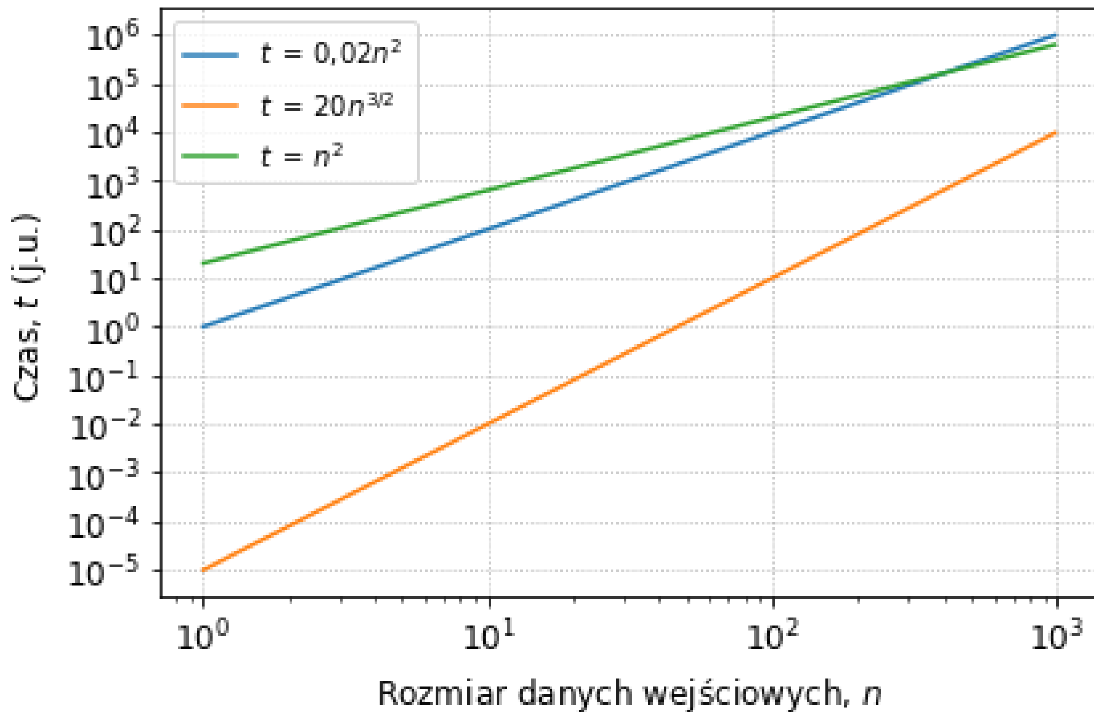
$\log_{10} t \; = \; \log_{10} C + k \log_{10} n.$

Zauważmy, że wykreślenie tej funkcji w skali podwójnie logarytmicznej (wykres typu `loglog` w bibliotece Matplotlib języka Python) da taki sam efekt, jak wykreślenie w skali liniowej wyrażenia $\log_{10} t$ w funkcji $\log_{10} n$. Jeśli więc wprowadzimy zmienne $t' = \log_{10} t$ i $n' = \log_{10} n$, to otrzymamy wykres

$t' \; = \; \log_{10} C + kn'.$

Będzie to więc linia prosta o współczynniku kierunkowym proporcjonalnym do wykładnika k i wyrazie wolnym proporcjonalnym do logarytmu z C. Wykreślmy powyższe trzy funkcje jeszcze raz, tym razem w skali podwójnie logarytmicznej na tle siatki.

[8]:
```
plt.loglog(n, t1, label='$t\:=\:0{,}02n^2$')
plt.loglog(n, t2, label='$t\:=\:20n^{3/2}$')
plt.loglog(n, t3, label='$t\:=\:n^2$')

plt.xlabel('Rozmiar danych wejściowych, $n$', fontsize=12)
plt.ylabel('Czas, $t$ (j.u.)', fontsize=12)
plt.xticks(fontsize=12)
plt.yticks([ 10**k for k in range(-5, 7) ], fontsize=12)
plt.tight_layout()
plt.legend()
plt.grid(linestyle='dotted')
plt.show()
```

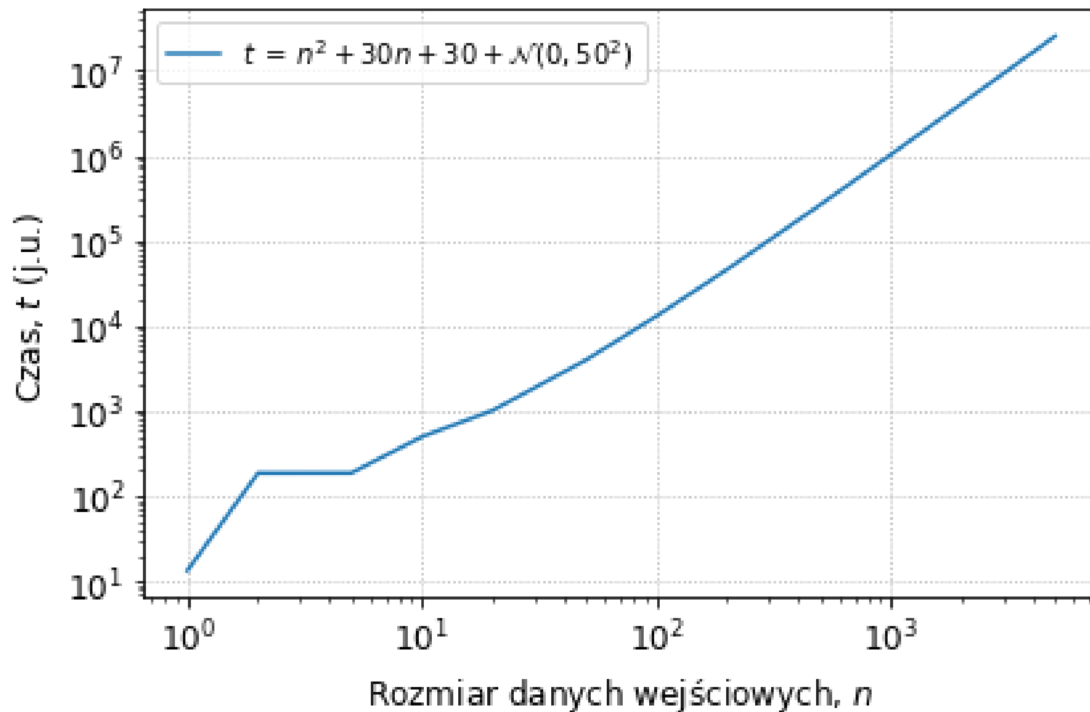


Skok siatki zarówno w pionie, jak i w poziomie, to w naszym przypadku jedna dekada. Nachylenie *niebieskiej* prostej, wyrażone w dekadach na dekadę, wynosi 2, więc taka też jest wartość wykładnika k. Zatem, jak przypuszczaliśmy, czas jest tu funkcją kwadratową rozmiaru danych n. *Pomarańczowa* funkcja, którą wzięliśmy za "stałą bliską zeru", jest w rzeczywistości proporcjonalna do n^3, zatem rośnie szybciej od niebieskiej. Choć ma bardzo mały współczynnik C, to z ostatniego wykresu można przewidzieć, że dla rozmiaru danych przekraczającego 10^5 to właśnie ona będzie dominować. Z kolei po przypatrzeniu się *zielonej* prostej widzimy, że na każde dwie dekady przesunięcia w poziomie przypadają trzy dekady wzrostu w pionie. Jest to więc funkcja potęgowa o wykładniku 3/2.

Z reguły zależność czasu wykonania algorytmu od rozmiaru problemu nie jest dana tak nieskomplikowaną funkcją, jak tym przykładzie. W dodatku każdy pomiar czasu jest obarczony pewnym błędem, wynikającym np. ze współdzielenia zasobów obliczeniowych z innymi procesami. W praktyce należy spodziewać się wykresów podobnych do poniższego.

[9]:
```
from numpy.random import randn

n = [ z*10**dec for dec in range(0,4) for z in [1,2,5] ]
t = [ x**2 + 30*x + 30 + noise for (x, noise) in zip(n, 50*randn(len(n))) ]

plt.loglog(n, t, label='$t\:=\:n^2+30n+30+\mathcal{N}(0,50^2)$')
plt.xlabel('Rozmiar danych wejściowych, $n$', fontsize=12)
plt.ylabel('Czas, $t$ (j.u.)', fontsize=12)
plt.xticks(fontsize=12)
```

```
plt.yticks(fontsize=12)
plt.grid(linestyle='dotted')
plt.legend()
plt.tight_layout()
plt.show()
```

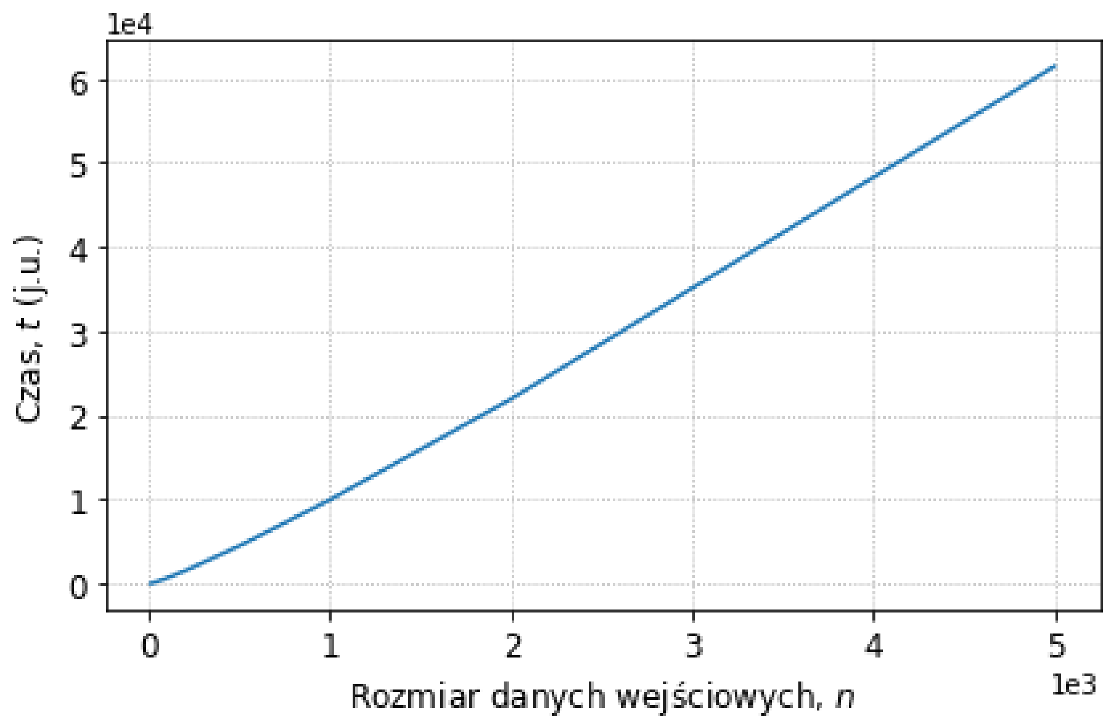


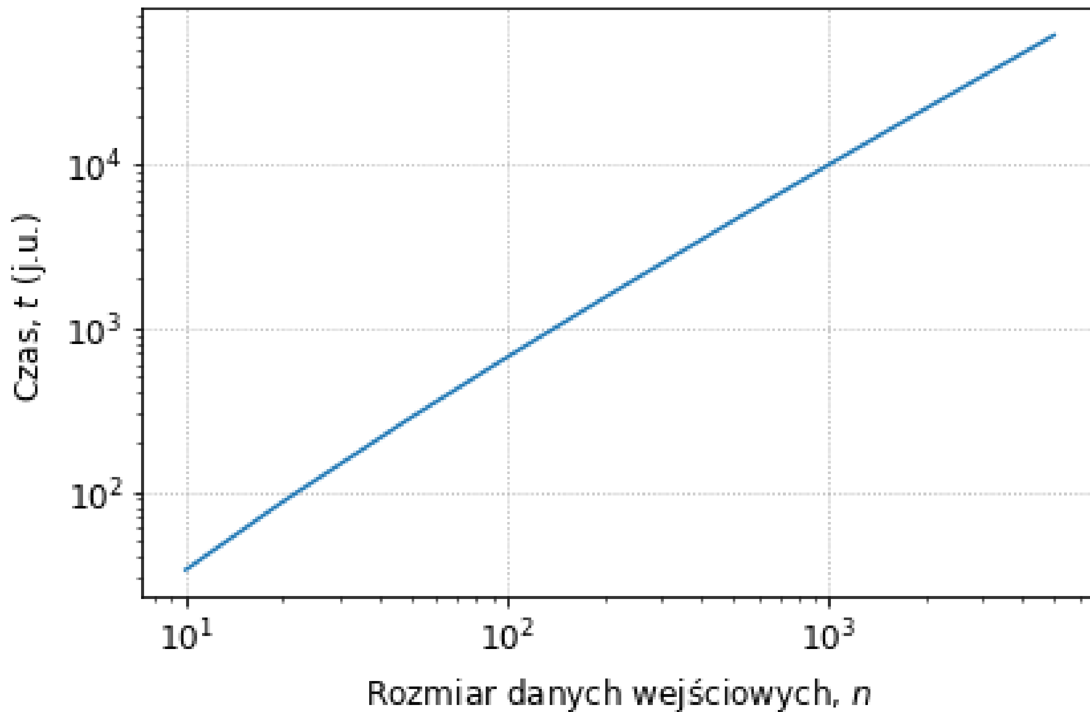
Jak widać, wykres zaczyna przypominać linię prostą dopiero dla rozmiaru danych przekraczającego 100. Dla mniejszych rozmiarów widoczny jest wpływ wyrazu liniowego, wyrazu stałego oraz addytywnego szumu gaussowskiego. Przy szacowaniu klasy złożoności należy oczywiście skupić się na liniowej części wykresu. Będzie to łatwiejsze, jeśli uda się wygenerować dodatkowe dane dla problemów przynajmniej o jeden rząd większych.

Oczywiście nie każdy algorytm charakteryzuje się złożonością wielomianową. Poniżej prezentujemy wykresy (w skali liniowej i podwójnie logarytmicznej) funkcji $n \log_2 n$, którą można opisać złożoność wielu algorytmów uważanych za efektywne.

[10]:
```
from math import log2

n = [ z*10**dec for dec in range(1,4) for z in [1,2,5] ]
t = [ x*log2(x) for x in n ]

plt.plot(n, t)
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.xlabel('Rozmiar danych wejściowych, $n$', fontsize=12)
```

```
plt.ylabel('Czas, $t$ (j.u.)', fontsize=12)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.tight_layout()
plt.grid(linestyle='dotted')
plt.show()

plt.loglog(n, t)
plt.xlabel('Rozmiar danych wejściowych, $n$', fontsize=12)
plt.ylabel('Czas, $t$ (j.u.)', fontsize=12)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.tight_layout()
plt.grid(linestyle='dotted')
plt.show()
```





2.3 Operacje na listach

UWAGA! W języku Python struktury danych typu `list` są implementowane jako tablice, **nie** jako struktury z łączami. Pociąga to za sobą wszystkie znane z wykładu konsekwencje dotyczące czasu dostępu do elementu o podanym indeksie, czasu potrzebnego na wstawienie do takiej struktury nowego elementu itp.

Na początku utworzymy listę liter alfabetu.

[11]:
```
import timeit
import string

letter_list = list(string.ascii_lowercase)
print(letter_list)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

Rozpocznijmy od wyznaczenia czasu dostępu do pierwszego elementu stworzonej listy, czyli elementu o indeksie 0. Pamiętajmy, że `repeat` wywołuje kilkukrotnie funkcję `timeit`, która domyślnie wykonuje milion razy polecenie przekazane jako `stmt`.

[12]:
```
min(timeit.repeat(stmt='letter_list[0]', globals=globals()))
```

[12]: `0.02516598699997985`

Teraz wyznaczymy czas dostępu do ostatniego elementu listy. Jest on dostępny pod indeksem –1.

[13]:
```
min(timeit.repeat(stmt='letter_list[-1]', globals=globals()))
```

[13]: 0.02349630399999114

Uzyskane wyniki pozwalają przypuszczać, że czas dostępu do elementu o podanym indeksie nie zależy od położenia tego elementu w ramach listy.

Następnie sprawdzimy wydajność operacji wyszukiwania elementu o danym kluczu. Pamiętajmy, że nasza lista zawiera litery uporządkowane alfabetycznie. Zaczynamy od litery `a`, kolejnymi poszukiwanymi znakami będą `n` i `z`.

[14]:
```
min(timeit.repeat(stmt='letter_list.index("a")', globals=globals()))
```

[14]: 0.03767511499995635

[15]:
```
min(timeit.repeat(stmt='letter_list.index("n")', globals=globals()))
```

[15]: 0.15070849699998234

[16]:
```
min(timeit.repeat(stmt='letter_list.index("z")', globals=globals()))
```

[16]: 0.2446047070000077

Na podstawie wyników można założyć, że lista jest przeszukiwana od początku do końca, więc znalezienie `z` trwa znacznie dłużej od znalezienia `a`.

2.4 Przeszukiwanie łańcucha

W następnym doświadczeniu będziemy przeszukiwać nie listę, lecz łańcuch znaków składający się z wielokrotnie powtórzonej litery `a`. W tym łańcuchów będziemy poszukiwać wzorców (napisów) wieloznakowych i sprawdzimy, jak długość poszukiwanego wzorca wpływa na czas tej operacji.

W języku Python tworzenie łańcucha składającego się z miliona identycznych znaków jest bardzo proste:

[17]:
```
full_str = 'a'*1000000
```

W podobny sposób zdefiniujemy poszukiwany wzorzec, czyli ciąg dziesięciu liter `a` i sprawdzimy, jak długo trwa jego znalezienie:

[18]:
```
substr = 'a'*10 # Poszukiwany wzorzec.
min(timeit.repeat(stmt='full_str.find(substr)', globals=globals()))
```

[18]: 0.12730844700001853

Teraz wydłużmy poszukiwany wzorzec dziesięciokrotnie i powtórzmy operację:

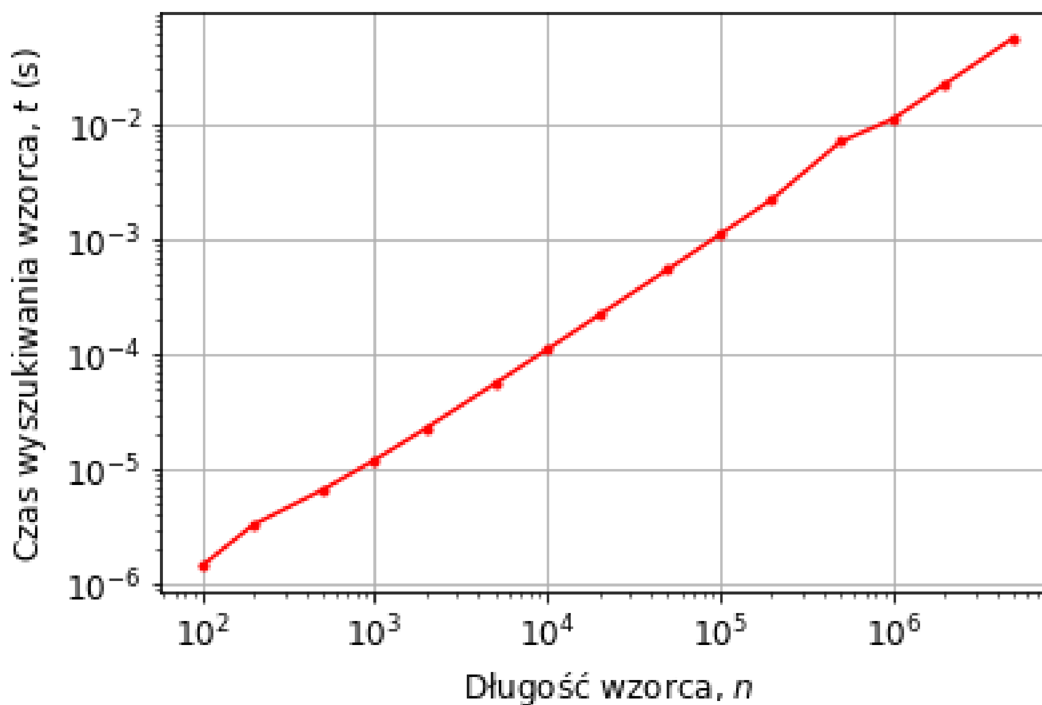
[19]:
```
substr = 'a'*100
min(timeit.repeat(stmt='full_str.find(substr)', globals=globals()))
```

[19]: 0.24926993899998706

Zapewne zaobserwowany przyrost czasu był mniejszy niż dziesięciokrotny. To dlatego, że – jak wspomnieliśmy wcześniej – samo wykonanie funkcji `timeit` wnosi pewne opóźnienie. Czas ten należałoby zmierzyć i odjąc od wyniku.

Spróbujmy podsumować wszystko, czego nauczyliśmy się do tej pory i określić zależność czasu poszukiwań wzorca od jego długości.

[20]:
```
import timeit
from matplotlib import pyplot as plt

full_str = 'a'*10000000

# Tabela rozmiarów szukanych wzorców: 100, 200, 500, 1000, 2000, ...
sizes = [ n*10**dec for dec in range(2,7) for n in [1,2,5] ]

# Lista czasów wyszukiwania coraz dłuższych wzorców
search_time = []

for s in sizes:
    pattern = 'a'*s
    # Mierzymy czas wykonania 'pustej' instrukcji 'timeit'
    time_noop = timeit.timeit(number=10)
    search_time.append(min(timeit.repeat(stmt='full_str.find(pattern)',␣
↪number=10, globals=globals()))-time_noop)

plt.loglog(sizes, search_time, 'r.-')
plt.xlabel('Długość wzorca, $n$', fontsize=12)
plt.ylabel('Czas wyszukiwania wzorca, $t$ (s)', fontsize=12)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.grid(True)
plt.show()
```



Otrzymany wykres w skali podwójnie logarytmicznej powinien przypominać linię prostą. Jeśli można na nim zaobserwować silne załamania, to warto powtórzyć eksperyment. Oszacujmy nachylenie otrzymanej prostej, wyrażone w dekadach na dekadę. Powinno wynosić 1, co oznacza, że złożoność czasowa jest liniową funkcją rozmiaru poszukiwanego wzorca.

2.4.1 Dla zainteresowanych:

- Implementacja klasy `list` w języku Python
- Zarządzanie pamięcią w języku Python
- M. Gorelick i I. Ozsvald, *Wysoko wydajny Python. Efektywne programowanie w praktyce. Wydanie 2*, Helion 2021 **UWAGA** Należy unikać Wydania 1 (niedostępnego zresztą w języku polskim), gdyż jest ono poświęcone starej wersji języka (Python2), gdzie implementacja pewnych struktur danych i algorytmów odbiega od obecnego standardu.