Złożoność obliczeniowa (c.d.)

Poprzedni wykład: **notacja asymptotyczna**. Formalny opis kiedy funkcja f zachowuje się asymptotycznie tak samo, jak funkcja g. Dla funkcji rosnących: kiedy f rośnie tak samo szybko, jak g (tzn. $\Theta(f) = \Theta(g)$).

Notacja Θ zaniedbuje stałe i (dla funkcji rosnących) składniki rosnące wolniej.

przykład $f(n)$	$\Theta(f(n))$	wzrost ...
$\frac{n}{2}$ $n + \sqrt{n}$	$\Theta(n)$	liniowy
$2n \log n$ $n \log(n^2 + 2n)$	$\Theta(n \log n)$	liniowo-logarytmiczny
n^2 $2n^2 + 6n + 1$	$\Theta(n^2)$	kwadratowy
$a_k n^k + \ldots + a_0$	$\Theta(n^k)$	wielomianowy
a^n	$\Theta(a^n)$	wykładniczy

Złożoność obliczeniowa (c.d.)

Na klasach $\Theta(\cdot)$ mamy naturalny porządek. Mówimy, że f rośnie wolniej niż g, gdy $\Theta(f) < \Theta(g)$ (czyli $f \in O(g) \wedge f \notin \Theta(g)$).

$$\begin{array}{|c|} \hline \Theta(1) \\ \hline \dots \\ 1 \\ 50 \\ 10^7 \\ \dots \\ \hline \end{array} < \begin{array}{|c|} \hline \Theta(n) \\ \hline \dots \\ n \\ 2n+2 \\ n+20 \\ \dots \\ \hline \end{array} < \begin{array}{|c|} \hline \Theta(n \log n) \\ \hline \dots \\ n \log n \\ n \log n + n \\ 6n \log n \\ \dots \\ \hline \end{array} < \begin{array}{|c|} \hline \Theta(n^2) \\ \hline \dots \\ n^2 \\ 2n^2 + 2n \\ n^2 + \sqrt{n} \\ \dots \\ \hline \end{array} < \dots$$

Związane z porządkiem: rodziny funkcji $O(\cdot), \Omega(\cdot)$.

Złożoność obliczeniowa (c.d.)

Dla danego algorytmu:
$T_{\text{best}}(n)$, $T_{\text{worst}}(n)$, $T_{\text{avg}}(n)$ – optymistyczny, pesymistyczny i średni czas działania algorytmu dla danych rozmiaru n (np. wejściowej liczby, długości wejściowej listy etc.).
Wyrażamy złożoność (czas działania) algorytmu poprzez notację asymptotyczną, np.

$T_{\text{avg}}(n) \in \Theta(n)$ „$T_{\text{avg}}(n)$ rośnie liniowo"
$T_{\text{worst}}(n) \in \Theta(n^2)$ „$T_{\text{worst}}(n)$ rośnie kwadratowo"

etc.

Kolokwialnie:
„Algorytm działa [optymistycznie/pesymistycznie/średnio] w czasie liniowym/kwadratowym/etc.".

Złożoność obliczeniowa (c.d.)

Czas działania algorytmu można zliczać w wybranych operacjach (np. zwykłe mnożenie macierzy $n \times n$ potrzebuje $\Theta(n^3)$ mnożeń współczynników). Domyślnie jednak zliczamy wszystkie operacje.

Proste operacje (np. arytmetyczne*, logiczne, porównania prostych obiektów, przypisania, przejścia do następnej instrukcji/bloku/iteracji) są wykonywane w czasie stałym.

Pewne operacje wymagają więcej czasu – wykonania wywoływanych funkcji i np. operacje na listach i innych strukturach danych. Czasy operacji na podstawowych obiektach w Pythonie:
https://wiki.python.org/moin/TimeComplexity

* – przynajmniej dla niezbyt dużych liczb.

Złożoność obliczeniowa (c.d.)

Przykład: badamy czasy wykonania $\mathrm{f}(n)$ i $\mathrm{g}(n)$.

```
def f(n):
    for i in range(n):
        for j in range(n):
            k = 2
```

$\mathrm{f}(n)$ wymaga $n^2 \in \Theta(n^2)$ przypisań.

```
def g(n):
    for i in range(n):
        for j in range(i):
            k = 2
```

$\mathrm{g}(n)$ wymaga $0 + 1 + 2 + \ldots n - 1 \in \Theta(n^2)$ przypisań.

Złożoność obliczeniowa (c.d.)

Przykład: badamy czas wykonania $\mathrm{h}(n)$.

```
def h(n):
    for i in range(n):
        if i % 2 == 0:
            k = 2
        else:
            k = 2
            l = 3
```

Fragment wewnątrz pętli wykonuje się w czasie stałym, zatem $\mathrm{h}(n)$ działa w czasie $\Theta(n)$.

Złożoność obliczeniowa (c.d.)

Przykład: badamy czas wykonania $\mathrm{f}(n)$.

```
def g(n):
    for i in range(n):
        k = 2

def f(n):
    for i in range(n):
        g(i)
```

$\mathrm{g}(n)$ działa w czasie $\Theta(n)$.
Zatem wykonanie każdego wywołania $\mathrm{g}(i)$ działa w czasie proporcjonalnym do i.
Stąd $\mathrm{f}(n)$ działa w czasie $\sum_{i=0}^{n-1} i \in \Theta(n^2)$.

Złożoność obliczeniowa (c.d.)

Przykład: badamy czas wykonania $\mathrm{g}(n)$.

```
def g(n):
    lst = list(range(n)) # Θ(n)
    while len(lst) > 0:
        lst.pop(0) # Θ(len(lst))
```

Stworzenie $\mathrm{range(n)}$: $\Theta(1)$.
Stworzenie listy długości n: $\Theta(n)$.
Usunięcie elementu z początku listy długości k: $\Theta(k)$.

Czas wykonania: $\Theta(n^2)$.

Czasy operacji listowych wzięte z: https://wiki.python.org/moin/TimeComplexity

Złożoność obliczeniowa (c.d.)

Złożoność asymptotyczna nie jest jedynym wyznacznikiem przydatności algorytmu.

Uwaga I: Paralelizacja: wiele algorytmów potrafi wykorzystywać więcej niż jeden logiczny procesor (rdzeń).
Uwaga II: stałe „ukryte" w notacji Θ mają czasem znaczenie.
Przykład: algorytmy mnożenia macierzy $n \times n$:

- Zwykły („naiwny"): $\Theta(n^3)$,
- Strassen (1969): $\Theta(n^{\log 7}) \approx \Theta(n^{2.81})$,
- Coppersmith–Winograd (1990): $O(n^{2.375477})$,
- Stothers (2010): $O(n^{2.374})$, Vassilevska (2011): $O(n^{2.3728642})$, Le Gall (2014): $O(n^{2.3728639})$, Alman-Vassilevska (2020): $O(n^{2.3728596})$, Duan-Wu-Zhou (2022): $O(n^{2.37188})$ (?).

W typowej implementacji, algorytm Strassena działa szybciej od naiwnego dla macierzy rozmiaru powyżej 100×100 (tak duże macierze pojawiają się powszechnie). Pozostałe algorytmy działają szybciej jedynie dla macierzy astronomicznych rozmiarów.

Struktury danych

Struktura danych – sposób organizacji danych w pamięci.

Pozwalają na przechowywanie obiektów oraz wykonywanie na nich pewnych operacji (np. dodawanie i usuwanie elementów, iteracja, indeksowanie), dając gwarancje na (asymptotyczne) czasy działania tych operacji.

Np. wbudowane struktury danych w Pythonie: listy, krotki, słowniki, zbiory.

Struktury danych – stos

Stos – struktura danych „z życia", reprezentująca obiekty ułożone jeden nad drugim. Obiekty można dodawać na szczyt stosu i ściągać ze szczytu stosu.

Stos obsługuje (co najmniej) operacje:

- $\mathrm{push(x)}$ – kładzie obiekt x na szczycie stosu.
- $\mathrm{pop()}$ – usuwa obiekt ze szczytu stosu i zwraca go[1].

Dla wygody, implementacje stosów często obsługują też poniższe operacje:

- $\mathrm{peek()}$ – zwraca (bez usuwania) obiekt ze szczytu stosu[1].
- $\mathrm{size()}$ – zwraca ilość obiektów na stosie.
- $\mathrm{is\_empty()}$ – orzeka, czy stos jest pusty.

Przykład struktury typu LIFO (last in, first out) – im później dodany obiekt, tym wcześniej zostanie ściągnięty.

[1] O ile stos nie jest pusty.

Struktury danych – stos

Stos zdefiniowaliśmy przez jego *interfejs* – zestaw operacji i ich znaczenia, bez wspominania o konkretnej implementacji.

Implementacja stosu – klasa Stack[1], z metodami odpowiadającymi operacjom.

Wewnętrznie Stack przechowuje obiekty na (początkowo pustej) liście. Wybór: czy szczyt stosu jest na początku, czy na końcu listy?

Stack – szczyt na końcu listy.
Stack1 (alternatywna implementacja) – szczyt na początku listy.

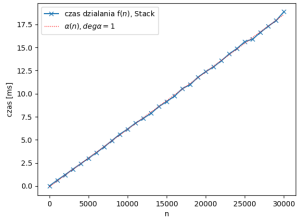
Operacja	Stack	Stack1
push(x)	list.append(x), $\Theta(1)$	list.insert(0, x), $\Theta(n)$
pop()	list.pop(), $\Theta(1)$	list.pop(0), $\Theta(n)$

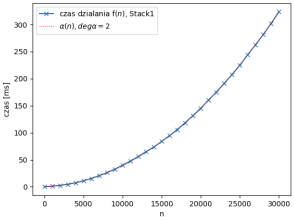
[1] stack.py, materiały do wykładu.

Struktury danych – stos

Na Stack i Stack1 można wykonywać identyczne operacje o takim samym znaczeniu, ale szczegóły implementacji decydują o czasie ich wykonania: Stack jest ściśle lepszy od Stack1.

```
def f(n):
    s = Stack() # Stack1()
    for i in range(n):
        s.push(i)
    while not s.is_empty():
        s.pop()
```





Struktury danych – stos

Przykładowe zastosowania stosu:

- Edytor tekstu – cofnij/powtórz.
- Przeglądarka – wstecz/dalej.
- Problem nawiasowania – czy w wyrażeniach złożonych z nawiasów każdy otwierający nawias jest zamknięty odpowiadającym mu nawiasem.
- Parsowanie plików HTML, XML, LaTeX, ... – uogólnienie poprzedniego przykładu.
- Parsowanie wyrażeń arytmetycznych.
- Reprezentacja liczby w danym systemie liczbowym.
- Wiele, wiele innych – podstawowa cegiełka różnych algorytmów.

Na wykładzie: problem nawiasowania; później parsowanie wyrażeń.