Złożoność obliczeniowa

Teoria złożoności obliczeniowej zajmuje się badaniem *zasobów* (czasu i pamięci) potrzebnych na wykonanie algorytmu **w zależności od *rozmiaru danych***, w sposób **niezależny od jego implementacji**.

Pozwala na obiektywny opis wydajności algorytmów, oraz porównywanie wydajności algorytmów, niezależnie od języka i sprzętu, w którym są zaimplementowane.

Motywujący przykład

Rozważmy dwa algorytmy rozwiązywania (trywialnego) problemu znajdowania sumy $0 + 1 + 2 + \ldots + n$, wyrażone w Pythonie:

Algorytm A – wprost, przez zsumowanie:

```
def sum1(n):
    total = n
    for i in range(n + 1):
        total += i
    return total
```

Algorytm B – przez formułę:

```
def sum2(n):
    return n * (n + 1) // 2
```

Motywujący przykład

Wyznaczamy czas pracy obu funkcji przy pewnych założeniach:

- Uwzględniamy jedynie czasy wykonywania operacji arytmetycznych – są to operacje opisane w algorytmie. W praktyce oba programy wykonują też inne operacje (np. stworzenie obiektu `range` etc.).
- Zakładamy, że czas operacji dodawania nie zależy od sumowanych liczb, podobnie dla mnożenia i dzielenia.
- Rozróżniamy jednak (na razie) czasy różnych operacji arytmetycznych.

Niech $t_a, t_m, t_d > 0$ to czasy (np. w nanosekundach lub cyklach procesora) wykonywania dodawań, mnożeń i dzieleń odpowiednio. Wtedy czasy wykonania obu funkcji to:

A) $n \cdot t_a$,
B) $t_a + t_m + t_d$.

Motywujący przykład

Czasy t_a, t_m, t_d zależą od platformy (sprzętu, wersji interpretera, etc.). Dla małych n nie można jednoznacznie powiedzieć, który z algorytmów jest szybszy. Zauważamy jednak, że czas działania A jest proporcjonalny do n, a czas działania B jest stały.

Stąd, niezależnie od wartości stałych t_a, t_m, t_d, dla dostatecznie dużych n, A będzie działał wolniej niż B (najmniejsze n dla którego B jest szybszy niż A może być bardzo duże, np. jeśli czas mnożenia i dzielenia jest znacznie większy, niż czas dodawania).

Intuicyjnie zatem, algorytm B jest „lepszy" niż A. Teoria złożoności obliczeniowej precyzuje takie intuicje.

Notacja wielkiego O

Od teraz funkcje, które rozważamy są elementami $\mathbb{R}^{\mathbb{N}}_{\geqslant 0}$. Dla $f \in \mathbb{R}^{\mathbb{N}}_{\geqslant 0}$ definiujemy następujący zbiór:

$$O(f) = \{g \in \mathbb{R}^{\mathbb{N}}_{\geqslant 0} : \exists_{M>0}\ \exists_{n_0 \in \mathbb{N}}\ \forall_{n \geqslant n_0}\ g(n) \leqslant M \cdot f(n)\}.$$

Fragment „$\exists_{n_0 \in \mathbb{N}}\ \forall_{n \geqslant n_0}\ \ldots$" to znane z analizy „dla dostatecznie dużych n ...".

Funkcja g jest więc elementem $O(f)$, jeśli istnieje stała $M > 0$ taka, że dla dostatecznie dużych n zachodzi $g(n) \leqslant M \cdot f(n)$. Opisowo: $g(n)$ da się ograniczyć **z góry** przez pewne przeskalowanie $f(n)$ przez stałą, być może poza małymi n.

Gdy $g \in O(f)$, mówimy że g rośnie (asymptotycznie) **nie szybciej** niż f.

Notacja wielkiego O

Zapis: zamiast $g \in O(f)$ możemy też pisać $g(n) \in O(f(n))$, lub podstawiać wyrażenia opisujące $f(n), g(n)$, na przykład:

- $n \in O(n)$,
- $\frac{n}{2} \in O(n)$,
- $100n \in O(2n)$,
- $n \in O(n^2)$,
- $n^2 + n + 100 \in O(n^2)$,
- $100 \in O(1)$.

Ponadto, wszystkie powyższe należenia są prawdziwe.

Notacja wielkiego O

Analogicznie, dla $f \in \mathbb{R}^{\mathbb{N}}_{\geqslant 0}$ definiujemy zbiór:

$$\Omega(f) = \{g \in \mathbb{R}^{\mathbb{N}}_{\geqslant 0} : \exists_{m>0}\ \exists_{n_0 \in \mathbb{N}}\ \forall_{n \geqslant n_0}\ m \cdot f(n) \leqslant g(n)\}.$$

Funkcja g jest elementem $\Omega(f)$, jeśli istnieje stała $m > 0$ taka, że dla dostatecznie dużych n zachodzi $m \cdot f(n) \leqslant g(n)$; a zatem $g(n)$ da się ograniczyć **z dołu** przez pewne przeskalowanie $f(n)$ przez stałą, być może poza małymi n.

Gdy $g \in \Omega(f)$, mówimy że g rośnie (asymptotycznie) **nie wolniej** niż f.

Notacja wielkiego O

Wreszcie definiujemy:

$$\Theta(f) = O(f) \cap \Omega(f).$$

Mamy: $g \in \Theta(f)$ gdy istnieją dwa przeskalowania f, które ograniczają g z dołu i z góry (poza małymi n).

Gdy $g \in \Theta(f)$, wtedy g rośnie nie szybciej i nie wolniej niż f. Mówimy wtedy, że g rośnie **tak samo** szybko, jak f.

Dla $\Omega(\cdot)$ i $\Theta(\cdot)$ stosujemy te same konwencje, co dla $O(\cdot)$, np.:

- $n + 1 \in \Theta(2n)$,
- $n^2 \in \Omega(n)$.

Notacja wielkiego O

Pytanie: na ile uprawniona jest opisana terminologia, w której funkcje rosną „nie wolniej", „nie szybciej", etc.?

Fakt

Niech $f, g, h \in \mathbb{R}^{\mathbb{N}}_{\geqslant 0}$. Wtedy:

1. $f \in \Theta(f)$ *(f rośnie nie szybciej i nie wolniej niż f).*
2. $g \in O(f) \iff f \in \Omega(g)$ *(g rośnie nie szybciej niż f wtedy i tylko wtedy, gdy f rośnie nie wolniej niż g).*
3. *Jeśli $f \in O(g)$ i $g \in O(h)$, to $f \in O(h)$ (jeśli f rośnie nie szybciej niż g i g rośnie nie szybciej niż h, to f rośnie nie szybciej, niż h). Analogicznie dla Ω w miejscu O.*

Dowód: skrypt/wykład.

Notacja wielkiego O

Zbiory postaci $\Theta(f)$ są klasami abstrakcji pewnej relacji równoważności, a na zbiorze tych klas abstrakcji możemy wprowadzić porządek częściowy. Konkretniej, definiujemy relację $\sim$ na $\mathbb{R}^{\mathbb{N}}_{\geqslant 0}$:

$$f \sim g \iff f \in \Theta(g).$$

Fakt

Relacja $\sim$ jest relacją równoważności.

Fakt

Dla wszystkich $f \in \mathbb{R}^{\mathbb{N}}_{\geqslant 0}$ mamy $[f]_{\sim} = \Theta(f)$.

Notacja wielkiego O

Przykłady $\sim$-klas i niektórych funkcji, które do niej należą:

- $\Theta(1)$ – zawiera wszystkie funkcje stałe (niezerowe).
- $\Theta(n)$ – zawiera wszystkie funkcje liniowe (poza stałymi).
- $\Theta(n^k)$ – zawiera wszystkie (nieujemne) wielomiany stopnia dokładnie k.
- $\Theta(\log n)$ – zawiera wszystkie logarytmy.

Klasy zawierają też inne funkcje. Przykładowo, $10 + \sin(n) \in \Theta(1)$.

Przynależenie funkcji do danej klasy pozwala mówić o sposobie, w jaki ta funkcja rośnie: np. funkcje z $\Theta(n)$ „rosną liniowo", funkcje z $\Theta(n^2)$ „rosną kwadratowo" (nawet jeśli nie są wielomianami drugiego stopnia).

Porządek na klasach Θ

Na klasach postaci $\Theta(f)$ wprowadzamy relację:

$$\Theta(f) \leqslant \Theta(g) \iff f \in O(g).$$

Fakt (ćw.)

Relacja $\leqslant$ jest dobrze określona i zadaje porządek częściowy na zbiorze klas.

Przykładowo:

$$\Theta(1) < \Theta(n) < \Theta(n^2).$$

Kolokwialnie te nierówności znaczą, że funkcje stałe rosną wolniej niż funkcje liniowe, a funkcje liniowe rosną wolniej niż kwadratowe.

(bardziej ściśle: funkcje rosnące tak samo szybko jak funkcje stałe rosną wolniej niż liniowo, a funkcje które rosną liniowo rosną wolniej niż kwadratowo).

Czasy działania algorytmów

Jak badać *złożoność czasową* algorytmu:

1. Opisać czas działania algorytmu jako funkcję $T(n)$, gdzie n to *rozmiar danych*.
2. Zbadać asymptotyczne zachowanie tej funkcji: opisać klasę $\Theta(T)$ (*).
3. Porównać tę klasę z klasami innych algorytmów (rozwiązujących ten sam problem).

Problemy: co to jest „rozmiar danych"? O jaki czas działania chodzi i jak go liczyć?

Czasy działania algorytmów

Danymi na których działa algorytm (jego „wejściem") mogą być dowolne obiekty – pojedyncza liczba, lista, plik etc. *Rozmiar danych* to cecha danych, względem której wyznaczamy czas działania algorytmu.

Przykład 1. Daną algorytmu A liczącego $0 + 1 + \ldots + n$ jest pojedyncza liczba n. Możemy po prostu przyjąć jej wartość jako rozmiar danych. Wtedy czas działania algorytmu (licząc wyłącznie operacje arytmetyczne) $T(n)$ wyraża się jako $T(n) = t_a \cdot n \in \Theta(n)$, zatem powiemy, że czas działania algorytmu jest liniowy w zależności od n.

Przykład 2. Dla algorytmu B i tak samo określonego rozmiaru danych, jego czas działania jest stały ($T(n) = t_a + t_m + t_d$), czyli jest w $\Theta(1)$.

Przykład 3. Wyszukiwanie maksimum na liście parami porównywalnych obiektów:

```
def maximum(lst):
    if len(lst) == 0: raise ValueError
    m = lst[0]
    for element in lst:
        if element > m:
            m = element
    return m
```

Kluczową operacją w algorytmie jest porównywanie obiektów z listy. Porównań tych będzie tyle, ile elementów na liście. Zatem przyjmujemy że rozmiarem danych jest długość listy ($n = \mathrm{len}(\mathrm{lst})$), a wtedy czas działania algorytmu (zliczając tylko porównania) w zależności od n jest klasy $\Theta(n)$ (algorytm działa liniowo względem długości listy).

Czasy działania algorytmów

Które operacje zliczamy? Szersza odpowiedź za tydzień. Na razie: zliczamy niektóre *operacje elementarne* (m.in. porównania, operacje arytmetyczne, przypisania).

Zakładamy, też, że wszystkie takie operacje wykonują się w czasie 1. Takie założenie *generalnie* nie wpływa na wyznaczenie złożoności algorytmu w notacji O.

Przykład zamiast uzasadnienia: Załóżmy, że dla danych rozmiaru n algorytm wykonuje n^2 mnożeń, $2n^2 + n$ dodawań i $n + 10$ dzieleń. Wtedy

$$T(n) = (t_m + 2t_a)n^2 + (t_a + t_d)n + 10t_d,$$

a taka funkcja jest klasy $\Theta(n^2)$ niezależnie od wartości stałych. Zatem bez straty ogólności możemy założyć $t_a = t_d = t_m = 1$.

Czasy działania algorytmów

Dalszy problem: rozmiar danych może nie być dostateczną informacją do (sensownego) określenia czasu działania algorytmu.

Przykład 4. Algorytm przyjmuje listę lst długości n złożoną z liczb $0, 1, 2, \ldots, n-1$ (każda występuje w lst raz) i szuka indeksu pierwszej *„transpozycji"* (termin ad hoc), mianowicie pary kolejnych liczb z listy, które nie są uporządkowane rosnąco:

```
def find_transposition(lst):
    n = len(lst)
    for i in range(0, n - 1):
        if lst[i] > lst[i + 1]:
            return i
    return None
```

Czasy działania algorytmów

Załóżmy, że zliczamy tylko porównania elementów na liście. Jeśli rozmiarem danych n jest długość listy, nie możemy jednoznacznie określić ile takich porównań się wykona:

- Jeśli lista jest np. postaci $[1, 0, ...]$, wtedy wykona się tylko jedno porównanie (niezależnie od długości).
- Jeśli lista jest posortowana, wtedy wykona się maksymalna liczba porównań: $n - 1$.
- Dla „przypadkowej" listy: trudno powiedzieć: między 0 a $n - 1$.

Powyższe listy reprezentują trzy możliwe przypadki danych tego samego rozmiaru: najlepszy (gdy potrzeba najmniej operacji), najgorszy (gdy potrzeba ich najwięcej), oraz „średni". Możemy użyć tych przypadków do dookreślenia czasu działania algorytmu.

Czasy działania algorytmów

Niech będzie dany algorytm wraz z ustalonym pojęciem „rozmiaru danych" (gdzie dla jednego rozmiaru istnieje wiele danych tego rozmiaru). Dla konkretnej danej wejściowej δ, niech $T(\delta)$ oznacza czas działania algorytmu dla δ. Niech Δ_n oznacza zbiór wszystkich danych wejściowych rozmiaru n (zakładamy o nim, że jest skończony). Definiujemy:

- Optymistyczny czas działania algorytmu:

$$T_{\text{best}}(n) := \min_{\delta \in \Delta_n} T(\delta).$$

- Pesymistyczny czas działania algorytmu:

$$T_{\text{worst}}(n) := \max_{\delta \in \Delta_n} T(\delta).$$

- Średni czas działania algorytmu:

$$T_{\text{avg}}(n) := \frac{1}{|\Delta_n|} \sum_{\delta \in \Delta_n} T(\delta).$$

Czasy działania algorytmów