3b. Asymptotyka: Notacja O

Grzegorz Kosiorowski

Uniwersytet Ekonomiczny w Krakowie

1. Motywacja i przykłady
2. Formalna definicja i twierdzenie
3. Hierarchia typowych ciągów
4. Działania na notacji O

Motywacja

W tym rozdziale wprowadzimy notację umożliwiającą szacowanie czasu działania (a co za tym idzie, szacowanie efektywności) algorytmów. Do tego przyda się odrobina wiedzy o ciągach.
Od razu zaznaczę, że tą kwestią zajmiemy się bardzo pobieżnie i w dużym uproszczeniu.

Motywacja

Rozważmy pewien algorytm, który wymaga danych, których liczność oznaczamy przez n (np. sortowanie tablicy n-elementowej, rozwiązanie układu n równań itp.). Badanemu algorytmowi przypisujemy czas jego wykonania, który wynosi $t(n)$ lub prościej, t_n. Dla większości algorytmów czas działania zależy od liczby danych, na których algorym ma wykonać dane działania, dlatego t_n nie jest pojedynczą liczbą, ale raczej ciągiem, dla $n = 1, 2, \ldots$. Poza tym, czas wykonywania zależy też od czynników niezależnych od konstrukcji algorytmu jak np. sprawności komputera na którym algorytm będzie wykonywany i innych kwestii technicznych. Jednak te kwestie można wyeliminować i, przy pomocy tzw. notacji O, mierzyć i porównywać efektywność różnych algorymów niezależnie od czynników zewnętrznych.

Motywacja

Badanemu algorytmowi przypisujemy czas jego wykonania, który wynosi $t(n)$ lub prościej, t_n. Dla większości algorytmów czas działania zależy od liczby danych, na których algorym ma wykonać dane działania, dlatego t_n nie jest pojedynczą liczbą, ale raczej ciągiem, dla $n = 1, 2, \ldots$. Poza tym, czas wykonywania zależy też od czynników niezależnych od konstrukcji algorytmu jak np. sprawności komputera na którym algorytm będzie wykonywany i innych kwestii technicznych. Jednak te kwestie można wyeliminować i, przy pomocy tzw. notacji O, mierzyć i porównywać efektywność różnych algorymów niezależnie od czynników zewnętrznych. Jest to szczególnie ważne, gdy rozważamy dwa algorytmy, których efekt działania jest taki sam: cenna jest wiedza, który będzie działać szybciej, o ile używamy dużej ilości danych.

Przykład 1 - ziemniaki

Dany jest worek ziemniaków. Powiedzmy, że liczba ziemniaków w worku wynosi n: to są dane algorytmów, które będziemy rozważać. Pierwszy algorytm ma za zadanie sprawdzić, czy w worku nie ma zgniłych ziemniaków. Robi to w sposób najprostszy z możliwych: pojedynczo sprawdza, czy każdy kolejny ziemniak jest zgniły, czy nie. Oczywiście, jeśli wykonanie operacji: „zbadaj zgniłość ziemniaka" zajmuje czas τ_1, to w najgorszym wypadku (konieczność zbadania wszystkich ziemniaków) wykonanie tego algorytmu potrwa $n \cdot \tau_1$. Łatwo zauważyć, że czas wykonania algorytmu rośnie liniowo wraz ze wzrostem liczby danych: 2 razy więcej ziemniaków oznacza 2 razy dłuższy czas działania.

Przykład 1 - ziemniaki

Dany jest worek ziemniaków. Powiedzmy, że liczba ziemniaków w worku wynosi n - i to są dane algorytmów, które będziemy rozważać. Drugi algorytm ma za zadanie sprawdzić, czy w worku jest para ziemniaków ważąca dokładnie tyle samo. Znów robi to w sposób najprostszy z możliwych: po kolei sprawdza wszystkie możliwe pary ziemniaków, aż natrafi na ważącą tyle samo. Jeśli wykonanie operacji: „porównaj wagę pary ziemniaków" zajmuje czas τ_2, to w najgorszym wypadku (każdy ziemniak innej wagi) wykonanie tego algorytmu potrwa $\frac{n(n-1)}{2} \cdot \tau_2$ (każdy ziemniak można porównać z $n-1$ innymi, ale wynik trzeba podzielić przez 2, by nie liczyć dwa razy np. porównywania ziemniaka pierwszego z drugim i drugiego z pierwszym). Dla dużych n możemy ten czas oszacować od góry przez $\frac{n^2}{2} \cdot \tau_2$. Oznacza to, że czas wykonania algorytmu rośnie w przybliżeniu kwadratowo wraz ze wzrostem liczby danych: 2 razy więcej ziemniaków oznacza mniej więcej 4 razy dłuższy czas działania.

Przykład 1 - ziemniaki

Co jest istotą porównania czasu działania tych algorymów? Dla ustalonego n, trudno powiedzieć, który algorytm wykonuje się dłużej, gdyż nie znamy czasów τ_1 i τ_2 (które zależą przecież też od sprzętu używanego do badania ziemniaków i innych czynników zewnętrznych). Dlatego istotne jest inne pytanie: dla którego z algorytmów czas wykonania wydłuża się bardziej w miarę zwiększania się ilości danych. I teraz widzimy, że dla odpowiednio dużych „worków", algorytm pierwszy zakończy działanie szybciej niż drugi: np. jeśli dla $n = 10$ drugi algorytm wykonuje się maksymalnie w 2 sekundy, a pierwszy w 5 sekund, to już dla $n = 100$ pierwszy algorytm zakończy się najpóźniej po 50 sekundach, a drugi może zająć nawet ponad 3 minuty.

Przykłady

Oczywiście, porównywanie czasu wykonania algorytmów, które mają zupełnie inne zadania ma głównie teoretyczną wartość (bo nie możemy powiedzieć, że któryś jest „lepszy”). Dlatego przykład poprzedni był skonstruowany tylko do celów ilustracyjnych. Kolejne dwa przykłady pokażą korzyści wynikające z istnienia narzędzia mierzącego asymptotyczny wzrost czasu wykonania algorytmów: możliwości porównania dwóch algorytmów, które mogłyby posłużyć do tego samego celu.

Przykład 2 - sortowania

Drugim przykładem będą algorytmy sortowania doskonale znane z podstaw programowania. Sortowanie bąbelkowe listy n elementowej w najprostszej postaci polega na $(n-1)$-krotnym przejściu przez listę, a w każdym z tych przejść musimy wykonać nie więcej niż $(n-1)$ porównań elementów (i ewentualnie zamienić je miejscami). Dlatego ostatecznie, wystarczy wykonać $(n-1)^2$ (najwyżej) operacji porównywania. Zatem ciąg t_n dla tego algorytmu rośnie proporcjonalnie do kwadratu długości listy.

Przykład 2 - sortowania

Z kolei można udowodnić, że sortowanie przez scalanie (merge sort) zajmuje czas proporcjonalny do $n \log n$ (gdzie n to długość listy, a log ma domyślną podstawę 2). By rozstrzygnąć, który z tych ciągów ma mniejszy przyrost zużycia czasu, wystarczy się dowiedzieć, czy szybciej rośnie ciąg n^2, czy też $n \log n$ (co za chwilę się okaże).

Przykład 3 - układy równań liniowych

Na algebrze i metodach numerycznych zapoznali lub zapoznają się Państwo z wieloma metodami rozwiązywania układów równań liniowych. Niektóre z nich są znacząco lepsze od innych. Na przykład, jeśli przez n oznaczymy wymiar macierzy głównej układu, to czas zużywany przez metodę wyznacznikową, opartą o wzory Cramera, rośnie proporcjonalnie do $n!$, a czas zużywany przez metody operacji elementarnych (np. metoda Gaussa-Jordana) rośnie proporcjonalnie do n^3.

Wstępne uwagi

Niech t_n oznacza czas, w jakim program poradzi sobie z danymi wielkości n. Na jakie aspekty ciągu $(t_n)_{n=1}^{\infty}$ powinniśmy zwracać uwagę? Nie wszystkie oczywiste zmiany wielkości t_n mają znaczenie:

- Pomnożenie kolejnych wyrazów ciągu t przez stałą: można to uzyskać używając szybszego lub wolniejszego sprzętu.
- Dodanie jakiejś stałej zmienia jeszcze mniej niż przemożenie.
- Istotna nie jest sama wartość t_n, ale szybkość, z jaką ta wielkość rośnie dla dużych n: czy rośnie tak szybko jak n, n^2, 2^n, a może $\log n$ lub jakaś inna funkcja.

Musimy zatem „tylko” dokładnie określić, co mamy na myśli mówiąc „rośnie tak szybko jak...”.

Definicja

Notacja O

Jeśli dane są dwa ciągi: $(t_n)_{n=1}^{\infty}$ i $(s_n)_{n=1}^{\infty}$ o wartościach rzeczywistych nieujemnych to mówimy, że $t_n = O(s_n)$ (czytamy: „t_n jest O od s_n") jeśli dla dużych n wartości ciągu t są nie większe niż wartości ciągu s pomnożonego przez pewną stałą. Bardziej formalnie:

$$\exists_{C>0}\exists_{n_0\in\mathbb{N}}\forall_{n>n_0}|t_n| \leqslant C|s_n|.$$

Zauważmy, że to oznaczenie jest tylko oszacowaniem od góry! Można taki zapis nieformalnie odczytywać jako „t_n rośnie nie szybciej niż s_n", ale t_n może rosnąć dowolnie wolno!

Niefortunna notacja

Tradycyjna notacja $t_n = O(s_n)$ jest niezbyt przemyślana, gdyż znak $=$ może wprowadzać w błąd. Oznacza on tutaj „jest”, ale w żadnym sensie nie oznacza równości. Na przykład, nie wyraża symetrii: wyrażenie $O(s_n) = t_n$ nie ma sensu, a jest możliwe (i raczej typowe), że $t_n = O(s_n)$, ale $s_n \neq O(t_n)$.
Dlatego, bardziej poprawny matematycznie i mniej mylący byłby zapis $t_n \in O(s_n)$. Jednakże, notacja $t_n = O(s_n)$ utarła się już w literaturze (zwłaszcza informatycznej), więc będziemy jej używać.

Notacja O - przykład

Przykład

$t_n = 6n^4 + 20n^2 + 2000$ jest $O(n^4)$.

Żeby to udowodnić, wystarczy zauważyć, że dla $n \geqslant 1$ zachodzi $6n^4 + 20n^2 + 2000 \leqslant 6n^4 + 20n^4 + 2000n^4 \leqslant 2026n^4$.
Zauważmy przy okazji, że $6n^4 + 20n^2 + 2000 \leqslant 2026n^4 \leqslant 2026n^5$, więc t_n jest też $O(n^5)$. Z drugiej strony, t_n nie jest np. $O(n^3)$, bo dla każdego C, jeśli $n > \frac{C}{6}$ to $6n^4 + 20n^2 + 2000 > 6n^4 = 6n \cdot n^3 > Cn^3$.

Twierdzenie o granicach i notacji O

Granice i notacja O

Załóżmy, że $(t_n)_{n=1}^{\infty}$ i $(s_n)_{n=1}^{\infty}$ są ciągami o wartościach nieujemnych, takimi, że $\lim_{n\to\infty} \frac{t_n}{s_n}$ istnieje. Wtedy:

I. Jeżeli $\lim_{n\to\infty} \frac{t_n}{s_n} = 0$, to $t_n = O(s_n)$, ale $s_n \neq O(t_n)$.

II. Jeżeli $\lim_{n\to\infty} \frac{t_n}{s_n} = \infty$, to $s_n = O(t_n)$, ale $t_n \neq O(s_n)$.

III. Jeżeli $\lim_{n\to\infty} \frac{t_n}{s_n} \in (0, \infty)$, to $t_n = O(s_n)$ i $s_n = O(t_n)$.

Twierdzenie wynika natychmiast z definicji notacji O i z definicji granicy ciągu.

Notacja O - przykład

Przykład

$\log n = O(n)$, ale $n \neq O(\log n)$.

Wystarczy zauważyć, że $\log n = \frac{\ln n}{\ln 2}$ i skorzystać z faktu obliczonego w poprzedniej prezentacji: $\lim_{n\to\infty} \frac{\ln n}{n} = 0$, a następnie z twierdzenia o granicach i notacji O.

Przykład

$t_n = 6n^4 + 20n^2 + 2000$ jest $O(n^4)$.

Prostszy dowód na podstawie twierdzenia o granicach i notacji O: wystarczy zauważyć, że

$$\lim_{n\to\infty} \frac{6n^4 + 20n^2 + 2000}{n^4} = 6 \in (0, \infty).$$

Notacja O - przykład uogólniony

Złożoność wielomianów

Jeśli t_n jest wielomianem stopnia k, to t_n jest $O(n^k)$ i nie jest $O(n^j)$ dla $j < k$.

Dowód jest natychmiastowy na podstawie znajomości granic typu „wielomian przez wielomian".
Ze względu na powyższą własność w początkowych przykładach algorytmy o czasach wykonania $\frac{n(n-1)}{2} \cdot t_2$ i $(n-1)^2$ traktowałem jako „algorytmy o kwadratowym wzroście zużycia czasu", czyli formalnie jako algorytmy o czasie wykonania typu $O(n^2)$.
Powstaje teraz pytanie: w jaki sposób notacja O porządkuje czasową efektywność algorytmów?

Czasowa złożoność obliczeniowa

Czasowa złożoność obliczeniowa

Jeśli a_n i b_n oznaczają ciągi czasów wykonania odpowiednio algorytmów A i B oraz $a_n = O(b_n)$, ale nie jest prawdą, że $b_n = O(a_n)$, to mówimy, że B ma większą *czasową złożoność obliczeniową* niż algorytm A.

W przyszłości będziemy pomijać słowo „czasowa”, gdyż to jedyny typ złożoności, który w ramach tego wykładu będziemy omawiać.
Zazwyczaj takie określenie oznacza, że algorytm A jest preferowany w stosunku do B, gdyż dla dużych ilości danych działa szybciej.
Oczywiście, w konkretnych zastosowaniach, algorytm o większej czasowej złożoności obliczeniowej nie musi być gorszy (bo może nadrabiać innymi zaletami), a nawet nie musi być wolniejszy dla małych zbiorów danych.

Przykład - ziemniaki

W pierwszym przykładzie z tego wykładu, poszukiwanie zgniłego ziemniaka w worku miało czas wykonania $n\tau_1$ (dla pewnego $\tau_1 \in \mathbb{R}_+$). W drugim przykładzie - porównywanie ziemniaków miało czas wykonania $\frac{n(n-1)}{2}\tau_2$ (dla pewnego $\tau_2 \in \mathbb{R}_+$). Obliczamy granicę:

$$\lim_{n\to\infty} \frac{n\tau_1}{\frac{n(n-1)}{2}\tau_2} = \lim_{n\to\infty} \frac{2\tau_1}{(n-1)\tau_2} = 0.$$

Stąd $n\tau_1 = O(\frac{n(n-1)}{2}\tau_2)$, ale $\frac{n(n-1)}{2}\tau_2 \neq O(n\tau_1)$, czyli pierwszy z tych algorytmów ma mniejszą złożoność obliczeniową.

Przykład - sortowanie

Sortowanie bąbelkowe ma maksymalny czas wykonania $t_1(n-1)^2$, a sortowanie przez scalanie - w przybliżeniu $t_2 n \log n$, $t_1, t_2 \in \mathbb{R}_+$. Pierwszy z tych algorytmów ma większą złożoność obliczeniową, gdyż

$$\lim_{n\to\infty} \frac{t_1(n-1)^2}{t_2 n \log n} = \frac{t_1}{t_2} \lim_{x\to\infty} \frac{(x-1)^2}{x \log x} \overset{H}{=} \frac{t_1}{t_2} \lim_{x\to\infty} \frac{2x-2}{\log x + \frac{1}{\ln 2}} \overset{H}{=}$$

$$\overset{H}{=} \frac{t_1}{t_2} \lim_{x\to\infty} \frac{2}{\frac{1}{x \ln 2}} = \infty.$$

Zatem dla dużych zbiorów danych sortowanie przez scalanie będzie wykonywane szybciej.

Przykład - układy równań liniowych

Rozwiązywanie układów równań liniowych metodą Gaussa-Jordana jest wykonywane w przybliżeniu w czasie an^3, a metodą wzorów Cramera - w czasie $bn!$, gdzie n jest liczbą równań, a $a, b \in \mathbb{R}$. Obliczamy granicę:

$$\lim_{n\to\infty} \frac{an^3}{bn!} = \lim_{n\to\infty} \frac{an^3}{bn(n-1)(n-2)(n-3)(n-4)!} = 0,$$

bo już sama granica $\lim_{n\to\infty} \frac{an^3}{bn(n-1)(n-2)(n-3)} = 0$. Zatem dla dużych układów równań, rozwiązywanie metodą Gaussa-Jordana jest dużo szybsze.

Przechodniość czasowej złożoności obliczeniowej

Przechodniość czasowej złożoności obliczeniowej

Jeśli $f(n) = O(g(n))$ i $g(n) = O(h(n))$, to $f(n) = O(h(n))$.
W szczególności, algorytm B ma większą czasową złożoność obliczeniową niż algorytm A, a algorytm C ma większą czasową złożoność obliczeniową niż algorytm B, to algorytm C ma większą czasową złożoność obliczeniową niż algorytm A.

To twierdzenie mówi, że porównywanie algorytmów przez czasową złożoność obliczeniową ma sens i ustawia je w pewnej hierarchii.

Twierdzenie o hierarchii

Twierdzenie o hierarchii

Oto hierarchia najbardziej znanych ciągów uporządkowanych w ten sposób, że każdy z nich jest O od wszystkich ciągów na prawo od niego:

$$1, \log n, n^{\alpha_1}, n^{\alpha_2}, n, n \log n, n^{\beta_1}, n^{\beta_2}, a^n, b^n, n!, n^n,$$

dla dowolnych rzeczywistych dodatnich $\alpha_1 < \alpha_2 < 1 < \beta_1 < \beta_2$ i $1 < a < b$.

1 oznacza tu ciąg stale równy 1, który w ogóle nie rośnie. Złożoność $O(1)$ odpowiada algorytmom, które działają w czasie stałym, czyli takim, których czas działania nie zależy od liczby danych, z którymi pracujemy.

Wzór Stirlinga

Uporządkowanie większości ciągów z twierdzenia o hierarchii łatwo uzasadnić dotychczasowymi metodami. Dla ciągów o dużej złożoności obliczeniowej (przy prawym końcu hierarchii), przydaje się sławny wzór Stirlinga.

Wzór Stirlinga

Dla dużych n mamy przybliżone oszacowanie:

$$n! \approx \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$$

Formalnie:

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\ \left(\frac{n}{e}\right)^n} = 1.$$

Hierarchia - przykład

Wzór Stirlinga

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1.$$

Na przykład, z tego wzoru wynika, że

$$\lim_{n\to\infty} \frac{n!}{n^n} = \lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} \cdot \lim_{n\to\infty} \frac{\sqrt{2\pi n}}{e^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}}{e^n} \overset{H}{=} 0.$$

Zatem $n! = O(n^n)$, ale $O(n^n) \neq n!$.

Hierarchia - uwagi

Notacja O podaje nam jedynie górne ograniczenie na złożoność obliczeniową. W takim wypadku zawsze interesuje nas ograniczenie jak najmniejsze (bo wszystkie większe wynikają z niego). Wróćmy do przykładu z $t_n = 6n^4 + 20n^2 + 2000$. Wiemy, że $t_n = O(n^4)$. Zgodnie z twierdzeniem o hierarchii jest też $t_n = O(n^5)$ i ogólnie $t_n = O(n^k)$ dla $k > 4$, a tym bardziej $t_n = O(2^n)$ i $t_n = O(n!)$. Jednak te wszystkie ograniczenia wynikają z ograniczenia najmniejszego, czyli $O(n^4)$ i dlatego właśnie $t_n = O(n^4)$ jest najbardziej interesującą odpowiedzią na pytanie o czasową złożoność obliczeniową algorytmu o czasie wykonania t_n.
W takiej sytuacji, im bardziej „na lewo" w przedstawionej wcześniej hierarchii da się umieścić ciąg czasów wykonania algorytmu, tym lepiej (tj. tym mniejsza jest czasowa złożoność obliczeniowa).

Hierarchia - wzrost wielomianowy

Wzrost wielomianowy

Ciąg t_n *rośnie wielomianowo* gdy $t_n = O(n^m)$ dla pewnego $m \in \mathbb{N}$.

W uproszczeniu, tylko algorytmy, których czas rośnie wielomianowo są uważane za praktyczne, jeśli możemy mieć do czynienia z dużymi zbiorami danych przy ich wykonywaniu (to kryterium jest znane jako teza Cobhama-Edmondsa). W praktyce, najbardziej pożądany (choć nie zawsze osiągalny) jest czas działania rzędu $O(n)$ lub $O(n \log n)$.

Wzrost wielomianowy i niewielomianowy

Przykład

Załóżmy, że dla kogoś maksymalny czas rozwiązania nieosobliwego układu 3 równań liniowych wynosi 5 minut metodą wzorów Cramera i 10 minut metodą operacji elementarnych. Zakładając, że jego tempo wykonywania odpowiednich operacji jest stałe (i się nie myli), jak można oszacować jego maksymalny czas rozwiązania nieosobliwego układu 6 równań liniowych obydwiema metodami?

Maksymalny czas wykonania metody operacji elementarnych jest $O(n^3)$, dlatego rozwiązywanie układu 6 równań liniowych tą metodą powinno zająć około $\frac{6^3}{3^3} = 8$ razy dłużej niż układu 3 równań, czyli 80 minut.Z kolei dla metody wzorów Cramera ($O(n!)$) odpowiedni stosunek wynosi $\frac{6!}{3!} = 120$, więc na rozwiązanie takiego układu trzebaby przeznaczyć 10 godzin!

Działania na notacji O

W dalszej części wykładu nie będziemy się zajmować obliczaniem złożoności obliczeniowej poszczególnych algorytmów - generalnie będę podawał ich czasową złożoność „na wiarę".
Na koniec chciałbym tylko podać jeszcze twierdzenie, które umożliwia wykonywanie działań na notacji O.

Działania na notacji O - twierdzenie

Działania na symbolach O

Zachodzą następujące zależności:

a) Jeśli $b_n = O(a_n)$ i k jest stałą, to $kb_n = O(a_n)$.

b) Jeśli $b_n = O(a_n)$ i $c_n = O(a_n)$, to $b_n + c_n = O(a_n)$.

c) Jeśli $b_n = O(a_n)$ i $d_n = O(c_n)$, to $b_n + d_n = O(\max\{a_n, c_n\})$.

d) Jeśli $b_n = O(a_n)$ i $d_n = O(c_n)$, to $b_n \cdot d_n = O(a_n \cdot c_n)$.

Działania na notacji O - twierdzenie

Działania na symbolach O

a) Jeśli $b_n = O(a_n)$ i k jest stałą, to $kb_n = O(a_n)$.
b) Jeśli $b_n = O(a_n)$ i $c_n = O(a_n)$, to $b_n + c_n = O(a_n)$.

Z punktu a) wynika, że jeśli jakaś procedura polega na wykonaniu jakiegoś algorytmu skończoną liczbę razy (k) to złożoność całej procedury jest taka, jak złożoność powtarzanego algorytmu.
Z punktu b) wynika, że jeśli dwa algorytmy mają tę samą złożoność obliczeniową i procedura polega na wykonaniu tych dwóch algorytmów, to złożoność procedury jest taka sama jak złożoność tych algorytmów.

Działania na notacji O - twierdzenie

Działania na symbolach O

c) Jeśli $b_n = O(a_n)$ i $d_n = O(c_n)$, to $b_n + d_n = O(\max\{a_n, c_n\})$.
d) Jeśli $b_n = O(a_n)$ i $d_n = O(c_n)$, to $b_n \cdot d_n = O(a_n \cdot c_n)$.

Z punktu c) wynika, że jeśli jakaś procedura polega na wykonaniu dwóch algorytmów, to jej złożoność jest równa złożoności bardziej złożonego z tych algorytmów.
Z punktu d) wynika, że jeśli procedura polega na wykonaniu algorytmu w takiej liczbie powtórzeń, jaką wyznacza drugi algorytm (np. jeden algorytm determinuje ilość wykonań pętli, w której każdym kroku wykonywany jest drugi algorytm), to złożoność procedury jest iloczynem złożoności algorytmów.

Uwaga końcowa

Notacja O nie podaje oszacowania z dołu na czas działania algorytmu. W szczególności, jeśli jakiś algorytm ma czas działania typu $O(n^2)$, to nic nie wyklucza faktu, że może też być $O(n)$ (co w wielu wypadkach bardzo by nas cieszyło). Wszystkie programy będące $O(n)$ są też $O(n^2)$. Istnieje notacja podobna do O, opisująca ograniczenia na złożoność od dołu za pomocą symbolu Ω. W ramach tego wykładu nie będziemy się nią zajmować.