2. Asymptotyka: Notacja O

Grzegorz Kosiorowski

Uniwersytet Ekonomiczny w Krakowie

Motywacja

W tym rozdziale przypomnimy znaną z metod numerycznych notację umożliwiającą szacowanie czasu działania (a co za tym idzie, efektywności) algorytmów. Do tego przyda się odrobina wiedzy o ciągach.

Motywacja

W tym rozdziale przypomnimy znaną z metod numerycznych notację umożliwiającą szacowanie czasu działania (a co za tym idzie, efektywności) algorytmów. Do tego przyda się odrobina wiedzy o ciągach.
Od razu zaznaczę, że tą kwestią zajmiemy się bardzo pobieżnie i w dużym uproszczeniu - ten materiał ma jedynie dać Państwu ogólną orientację w tym zagadnieniu. Nie jest bardzo potrzebny dla całości kursu, ale wypadałoby, żeby informatycy wiedzieli przynajmniej, że notacja O istnieje i co ona oznacza.

Motywacja

Rozważmy pewien algorytm, który wymaga danych, których liczność oznaczamy przez n (np. sortowanie tablicy n-elementowej, rozwiązanie układu równań o danej liczbie równań i niewiadomych itp.).

Motywacja

Rozważmy pewien algorytm, który wymaga danych, których liczność oznaczamy przez n (np. sortowanie tablicy n-elementowej, rozwiązanie układu równań o danej liczbie równań i niewiadomych itp.). Badanemu algorytmowi przypisujemy czas jego wykonania, który wynosi $t(n)$.

Motywacja

Rozważmy pewien algorytm, który wymaga danych, których liczność oznaczamy przez n (np. sortowanie tablicy n-elementowej, rozwiązanie układu równań o danej liczbie równań i niewiadomych itp.). Badanemu algorytmowi przypisujemy czas jego wykonania, który wynosi $t(n)$. Oczywiście, 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 $n = 1, 2, \ldots$.

Motywacja

Rozważmy pewien algorytm, który wymaga danych, których liczność oznaczamy przez n (np. sortowanie tablicy n-elementowej, rozwiązanie układu równań o danej liczbie równań i niewiadomych itp.). Badanemu algorytmowi przypisujemy czas jego wykonania, który wynosi $t(n)$. Oczywiście, 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.

Motywacja

Rozważmy pewien algorytm, który wymaga danych, których liczność oznaczamy przez n (np. sortowanie tablicy n-elementowej, rozwiązanie układu równań o danej liczbie równań i niewiadomych itp.). Badanemu algorytmowi przypisujemy czas jego wykonania, który wynosi $t(n)$. Oczywiście, 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. Za chwilę jednak zobaczymy, że te czynniki 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 $t(n)$. Oczywiście, 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. Za chwilę jednak zobaczymy, że te czynniki 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.

Przykład 1 - ziemniaki

Dany jest worek ziemniaków. Powiedzmy, że liczba ziemniaków w worku wynosi n - i to są dane algorytmów, które będziemy rozważać.

Przykład 1 - ziemniaki

Dany jest worek ziemniaków. Powiedzmy, że liczba ziemniaków w worku wynosi n - i 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.

Przykład 1 - ziemniaki

Dany jest worek ziemniaków. Powiedzmy, że liczba ziemniaków w worku wynosi n - i 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 t_1, to w najgorszym wypadku (brak zgniłych ziemniaków) wykonanie tego algorytmu potrwa $n \cdot t_1$.

Przykład 1 - ziemniaki

Dany jest worek ziemniaków. Powiedzmy, że liczba ziemniaków w worku wynosi n - i 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 t_1, to w najgorszym wypadku (brak zgniłych ziemniaków) wykonanie tego algorytmu potrwa $n \cdot t_1$. 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.

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 t_2, to w najgorszym wypadku (każdy ziemniak innej wagi) wykonanie tego algorytmu potrwa $\frac{n(n-1)}{2} \cdot t_2$ (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).

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 t_2, to w najgorszym wypadku (każdy ziemniak innej wagi) wykonanie tego algorytmu potrwa $\frac{n(n-1)}{2} \cdot t_2$ (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 $\frac{n^2}{2} \cdot t_2$.

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 t_2, to w najgorszym wypadku (każdy ziemniak innej wagi) wykonanie tego algorytmu potrwa $\frac{n(n-1)}{2} \cdot t_2$ (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 $\frac{n^2}{2} \cdot t_2$. 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?

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 t_1 i t_2 (które zależą przecież też od sprzętu używanego do badania ziemniaków i innych czynników zewnętrznych).

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 t_1 i t_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.

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 t_1 i t_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,

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 t_1 i t_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.

Przykład 1 - ziemniaki

Oczywiście, porównywanie czasu wykonania algorytmów, które mają zupełnie inne zadania zazwyczaj niczemu nie służy (bo nie możemy powiedzieć, że któryś jest „lepszy").

Przykład 1 - ziemniaki

Oczywiście, porównywanie czasu wykonania algorytmów, które mają zupełnie inne zadania zazwyczaj niczemu nie służy (bo nie możemy powiedzieć, że któryś jest „lepszy”). Dlatego przykład poprzedni był skonstruowany tylko do celów ilustracyjnych.

Przykład 1 - ziemniaki

Oczywiście, porównywanie czasu wykonania algorytmów, które mają zupełnie inne zadania zazwyczaj niczemu nie służy (bo nie możemy powiedzieć, że któryś jest „lepszy”). Dlatego przykład poprzedni był skonstruowany tylko do celów ilustracyjnych. Kolejne dwa pokażą korzyści wynikające z istnienia narzędzia porównania asymptotycznego wzrostu złożoności czasowej 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.

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).

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 $(n-1)^2$ (a nawet nieco mniej) operacji porównywania.

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 $(n-1)^2$ (a nawet nieco mniej) 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, jak na wszystkich zajęciach z tego przedmiotu ma domyślną podstawę 2).

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, jak na wszystkich zajęciach z tego przedmiotu 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 się Państwo z wieloma metodami rozwiązywania równań liniowych.

Przykład 3 - układy równań liniowych

Na algebrze i metodach numerycznych zapoznali się Państwo z wieloma metodami rozwiązywania równań liniowych. Niektóre z nich są znacząco lepsze od innych.

Przykład 3 - układy równań liniowych

Na algebrze i metodach numerycznych zapoznali się Państwo z wieloma metodami rozwiązywania 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 rozkładu na iloczyn macierzy trójkątnych (np. metoda Cholesky'ego) rośnie proporcjonalnie do n^3.

Wstępne uwagi

Niech $t(n)$ oznacza czas, w jakim program poradzi sobie z listą danych długości n. Rozważamy teraz ciąg $t(n)$ ze zmieniającym się n.

Wstępne uwagi

Niech $t(n)$ oznacza czas, w jakim program poradzi sobie z listą danych długości n. Rozważamy teraz ciąg $t(n)$ ze zmieniającym się n. Na jakie aspekty tego ciągu powinniśmy zwracać uwagę?

Wstępne uwagi

Niech $t(n)$ oznacza czas, w jakim program poradzi sobie z listą danych długości n. Rozważamy teraz ciąg $t(n)$ ze zmieniającym się n. Na jakie aspekty tego ciągu powinniśmy zwracać uwagę? Pomnożenie kolejnych wyrazów t przez jakąś liczbę nie powinno nas zbytnio interesować - można to uzyskać używając szybszego lub wolniejszego komputera.

Wstępne uwagi

Niech $t(n)$ oznacza czas, w jakim program poradzi sobie z listą danych długości n. Rozważamy teraz ciąg $t(n)$ ze zmieniającym się n. Na jakie aspekty tego ciągu powinniśmy zwracać uwagę? Pomnożenie kolejnych wyrazów t przez jakąś liczbę nie powinno nas zbytnio interesować - można to uzyskać używając szybszego lub wolniejszego komputera. Zatem istotna nie jest absolutna wielkość $t(n)$, 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: $t(n)$ i $s(n)$ 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:

$$\exists_{C \in \mathbb{N}} \exists_{n_0 \in \mathbb{N}} \forall_{n > n_0} |t(n)| \leqslant C|s(n)|.$$

Definicja

Notacja O

Jeśli dane są dwa ciągi: $t(n)$ i $s(n)$ 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:

$$\exists_{C \in \mathbb{N}} \exists_{n_0 \in \mathbb{N}} \forall_{n > n_0} |t(n)| \leqslant C|s(n)|.$$

Zauważmy, że to oznaczenie jest tylko oszacowaniem od góry!

Definicja

Notacja O

Jeśli dane są dwa ciągi: $t(n)$ i $s(n)$ 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:

$$\exists_{C \in \mathbb{N}} \exists_{n_0 \in \mathbb{N}} \forall_{n > n_0} |t(n)| \leqslant C|s(n)|.$$

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!

Notacja O - przykład

Przykład

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

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 $6n^4 + 20n^2 + 2000$

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 $6n^4 + 20n^2 + 2000 \leqslant 6n^4 + 20n^4 + 2000n^4$

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 $6n^4 + 20n^2 + 2000 \leqslant 6n^4 + 20n^4 + 2000n^4 \leqslant 2026n^4$.

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 $6n^4 + 20n^2 + 2000 \leqslant 2026n^4 \leqslant 2026n^5$,

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 $O(n^5)$.

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
$6n^4 + 20n^2 + 2000 >$

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
$6n^4 + 20n^2 + 2000 > 6n^4 = 6n \cdot n^3$

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
$6n^4 + 20n^2 + 2000 > 6n^4 = 6n \cdot n^3 > Cn^3$.

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 $j < n$.

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 $j < n$.

Ze względu na to twierdzenie 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 $O(n^2)$.

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 $j < n$.

Ze względu na to twierdzenie 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 $O(n^2)$.
Powstaje teraz pytanie: w jaki sposób notacja O porządkuje czasową efektywność algorytmów?

Czasowa złożoność obliczeniowa

Czasowa złożoność obliczeniowa

Jeśli $t_A(n)$ i $t_B(n)$ oznaczają ciągi czasów wykonania odpowiednio algorytmów A i B oraz $t_A(n) = O(t_B(n))$, ale nie jest prawdą, że $t_B(n) = O(t_A(n))$, to mówimy, że B ma większą *czasową złożoność obliczeniową* niż algorytm A. W późniejszych wykładach będziemy pomijać słowo „czasowa”, gdyż to jedyny typ złożoności, który tu będziemy omawiać.

Czasowa złożoność obliczeniowa

Czasowa złożoność obliczeniowa

Jeśli $t_A(n)$ i $t_B(n)$ oznaczają ciągi czasów wykonania odpowiednio algorytmów A i B oraz $t_A(n) = O(t_B(n))$, ale nie jest prawdą, że $t_B(n) = O(t_A(n))$, to mówimy, że B ma większą *czasową złożoność obliczeniową* niż algorytm A. W późniejszych wykładach będziemy pomijać słowo „czasowa”, gdyż to jedyny typ złożoności, który tu będziemy omawiać.

Zazwyczaj takie określenie oznacza, że algorytm A jest preferowany w stosunku do B, gdyż dla dużych ilości danych działa szybciej.

Czasowa złożoność obliczeniowa

Czasowa złożoność obliczeniowa

Jeśli $t_A(n)$ i $t_B(n)$ oznaczają ciągi czasów wykonania odpowiednio algorytmów A i B oraz $t_A(n) = O(t_B(n))$, ale nie jest prawdą, że $t_B(n) = O(t_A(n))$, to mówimy, że B ma większą *czasową złożoność obliczeniową* niż algorytm A. W późniejszych wykładach będziemy pomijać słowo „czasowa”, gdyż to jedyny typ złożoności, który tu 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),

Czasowa złożoność obliczeniowa

Czasowa złożoność obliczeniowa

Jeśli $t_A(n)$ i $t_B(n)$ oznaczają ciągi czasów wykonania odpowiednio algorytmów A i B oraz $t_A(n) = O(t_B(n))$, ale nie jest prawdą, że $t_B(n) = O(t_A(n))$, to mówimy, że B ma większą *czasową złożoność obliczeniową* niż algorytm A. W późniejszych wykładach będziemy pomijać słowo „czasowa”, gdyż to jedyny typ złożoności, który tu 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.

Czasowa złożoność obliczeniowa

Czasowa złożoność obliczeniowa

Jeśli $t_A(n)$ i $t_B(n)$ oznaczają ciągi czasów wykonania odpowiednio algorytmów A i B oraz $t_A(n) = O(t_B(n))$, ale nie jest prawdą, że $t_B(n) = O(t_A(n))$, to mówimy, że B ma większą *czasową złożoność obliczeniową* niż algorytm A. W późniejszych wykładach będziemy pomijać słowo „czasowa", gdyż to jedyny typ złożoności, który tu będziemy omawiać.

Większa czasowa złożoność obliczeniowa mówi tylko, że dla n większych od pewnej wartości czas wykonania A będzie mniejszy niż czas wykonania B, ale ta „pewna wartość" może być tak duża, że w konkretnej, praktycznej implementacji może to nie mieć znaczenia.

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.

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 $1 < a < b$.

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.

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.

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.

Hierarchia - uwagi

Notacja O podaje nam jedynie górne ograniczenie na złożoność obliczeniową.

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).

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 $t(n) = 6n^4 + 20n^2 + 2000$.

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 $t(n) = O(n^4)$.

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 $t(n) = O(n!)$.

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 $t(n)$.

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 - przykład 1

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 $1 < a < b$.

W pierwszym przykładzie z tego wykładu, poszukiwanie zgniłego ziemniaka w worku miało złożoność czasową typu $O(n)$. W drugim przykładzie - porównywanie ziemniaków miało złożoność typu $O(n^2)$.

Hierarchia - przykład 1

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 $1 < a < b$.

W pierwszym przykładzie z tego wykładu, poszukiwanie zgniłego ziemniaka w worku miało złożoność czasową typu $O(n)$. W drugim przykładzie - porównywanie ziemniaków miało złożoność typu $O(n^2)$. Stąd (i z twierdzenia o hierarchii) wiemy, że pierwszy z tych algorytmów ma mniejszą złożoność obliczeniową.

Hierarchia - przykład 2

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 $1 < a < b$.

Hierarchia - przykład 2

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 $1 < a < b$.

Sortowanie bąbelkowe ma złożoność typu $O(n^2)$, a sortowanie przez scalanie - typu $O(n \log n)$.

Hierarchia - przykład 2

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 $1 < a < b$.

Sortowanie bąbelkowe ma złożoność typu $O(n^2)$, a sortowanie przez scalanie - typu $O(n \log n)$. Pierwszy z tych algorytmów ma większą złożoność obliczeniową, gdyż $n \log n$ jest przed n^2 w twierdzeniu o hierarchii.

Hierarchia - przykład 3

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 $1 < a < b$.

Hierarchia - przykład 3

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 $1 < a < b$.

Rozwiązywanie układów równań liniowych metodą Cholesky'ego ma złożoność typu $O(n^3)$, a metodą wzorów Cramera - typu $O(n!)$, gdzie n jest liczbą równań.

Hierarchia - przykład 3

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 $1 < a < b$.

Rozwiązywanie układów równań liniowych metodą Cholesky'ego ma złożoność typu $O(n^3)$, a metodą wzorów Cramera - typu $O(n!)$, gdzie n jest liczbą równań. Jak widać, pierwszy z tych algorytmów ma mniejszą złożoność obliczeniową.

Hierarchia - wzrost wielomianowy

Wzrost wielomianowy

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

Hierarchia - wzrost wielomianowy

Wzrost wielomianowy

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

Z teoretycznego punktu widzenia, tylko algorytmy, których czas rośnie wielomianowo są uważane za praktyczne, jeśli mamy do czynienia z długimi listami przy ich wykonywaniu. 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 Cholesky'ego. 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?

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 Cholesky'ego. 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 Cholesky'ego jest proporocjonalny do trzeciej potęgi liczby równań, dlatego rozwiązywanie układu 6 równań liniowych tą metodą powinno zająć około

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 Cholesky'ego. 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 Cholesky'ego jest proporocjonalny do trzeciej potęgi liczby równań, 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.

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 Cholesky'ego. 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 Cholesky'ego jest proporocjonalny do trzeciej potęgi liczby równań, 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 odpowiedni stosunek wynosi

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 Cholesky'ego. 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 Cholesky'ego jest proporocjonalny do trzeciej potęgi liczby równań, 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 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”.

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 $f(n) = O(g(n))$ i c jest stałą, to $cf(n) = O(g(n))$.

b) Jeśli $f(n) = O(g(n))$ i $h(n) = O(g(n))$, to $f(n) + h(n) = O(g(n))$.

c) Jeśli $f(n) = O(g(n))$ i $h(n) = O(k(n))$, to $f(n) + h(n) = O(\max\{g(n), k(n)\})$.

d) Jeśli $f(n) = O(a(n))$ i $g(n) = O(b(n))$, to $f(n) \cdot g(n) = O(a(n) \cdot b(n))$.

Działania na notacji O - twierdzenie

Działania na symbolach O

Zachodzą następujące zależności:
a) Jeśli $f(n) = O(g(n))$ i c jest stałą, to $cf(n) = O(g(n))$.
b) Jeśli $f(n) = O(g(n))$ i $h(n) = O(g(n))$, to
$f(n) + h(n) = O(g(n))$.
c) Jeśli $f(n) = O(g(n))$ i $h(n) = O(k(n))$, to
$f(n) + h(n) = O(\max\{g(n), k(n)\})$.
d) Jeśli $f(n) = O(a(n))$ i $g(n) = O(b(n))$, to
$f(n) \cdot g(n) = O(a(n) \cdot b(n))$.

Twierdzenie to mówi o zasadach dodawania i mnożenia przez siebie złożoności obliczeniowych różnych algorytmów.

Uwaga końcowa

Jak już wspomniałem wcześniej, notacja O nie podaje oszacowania z dołu na czas działania algorytmu.

Uwaga końcowa

Jak już wspomniałem wcześniej, 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).

Uwaga końcowa

Jak już wspomniałem wcześniej, 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 $O(n^2)$.

Uwaga końcowa

Jak już wspomniałem wcześniej, 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 $O(n^2)$. Oszacowania z dołu zapisuje się notacją podobną do O - za pomocą symbolu Ω.

Uwaga końcowa

Jak już wspomniałem wcześniej, 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 $O(n^2)$. Oszacowania z dołu zapisuje się notacją podobną do O - za pomocą symbolu Ω. Ze względu na brak czasu, nie będziemy się tym na wykładzie zajmować.