Algorytmy

Algorytm jest przepisem na wykonanie jakichś obliczeń. Celem tych obliczeń jest uzyskanie jakiegoś wyniku, oraz zwykle algorytm operuje na jakichś danych, które muszą być dostępne aby możliwe było wykonanie algorytmu i otrzymanie tego wyniku.

Przykłady algorytmów stosowanych w różnych dziedzinach:

algorytmy arytmetyczne — stosowane w komputerach do obliczania popularnych funkcji matematycznych, takich jak: pierwiastki, logarytmy, funkcje trygonometryczne, itp.

algorytmy grafowe — zastosowania w znajdowaniu ścieżek dla: nawigacji, znajdowania tras w sieciach komputerowych, projektowaniu układów VLSI, grach komputerowych, itp.

algorytmy tekstowe — wyszukiwanie haseł, indeksowanie, kompresja, itp.

algorytmy analizy matematycznej — przetwarzanie sygnałów, filtracja, transformacja obrazów

algorytmy uczenia maszynowego — przetwarzanie wielkich zbiorów danych w celu automatycznej budowy systemów sztucznej inteligencji, ale również analizy danych, itp.

Pseudokod

Zawarty w algorytmie przepis musi być podany w sposób jasny, precyzyjny, i jednoznaczny. W literaturze związanej z informatyką algorytmy zwyczajowo zapisuje się w tzw. **pseudokodzie**, czyli w pewnym abstrakcyjnym języku, podobnym do popularnych języków programowania.

Na przykład, pseudokod algorytmu sumowania elementów tablicy:

```
SUM-ARRAY(A, n)
1  sum = 0
2  for i = 1 to n
3        sum = sum + A[i]
4  return sum
```

Zwróćmy uwagę:

- bloki instrukcji uwidocznione wcięciami
- instrukcje nie muszą (ale mogą) być zakończone średnikiem
- zakładamy numerowanie elementów tablicy od 1 do N (ale gdy wygodne, możemy również użyć numeracji od 0 do N-1, odnotowując to za pomocą komentarzy)
- w pseudokodzie mogą pojawiać się obiektowe atrybuty w stylu A.length
- zwykłe, skalarne, parametry procedur są przekazywane przez wartość, ale tablice i obiekty jako parametry procedur przekazywane są przez wskaźnik

Zagadnienie sortowania

Jednym z ważnych zagadnień, dla których istnieje szereg ważnych algorytmów jest zagadnienie sortowania. Problem polega, dla danej na wejściu sekwencji liczb, na wygenerowaniu na wyjściu algorytmu innej sekwencji, będącej niemalejącą permutacją sekwencji wejściowej. Permutacja jest sekwencją tych samych liczb, ale być może w innym porządku.

Konieczność sortowania zachodzi w wielu zastosowaniach praktycznych, i polega zwykle na uporządkowaniu zbioru bardziej złożonych obiektów, z których każdy charakteryzuje pewna wielkość, zwana **kluczem sortowania**, według którego ciąg powinien być uporządkowany.

Popularny, jeden z najprostszych algorytmów sortowania:

```
BUBBLE-SORT(A, n)
1  for i = 1 to n − 1
2      for j = n downto i + 1
3          if A[j] < A[j − 1]
4              // zamień wartości A[j] i A[j − 1]
5              tmp = A[j − 1]
6              A[j − 1] = A[j]
7              A[j] = tmp
```

Poprawność algorytmu sortowania

Dla każdego algorytmu ważne jest w pełni rygorystyczne wykazanie, że algorytm jest **poprawny**, to znaczy w każdym przypadku realizuje swoje zadanie. W przypadku sortowania poprawność algorytmu sprowadza się do zapewnienia dwóch własności:

- ciąg wynikowy musi zawierać te same liczby co wejściowy,
- ciąg wynikowy musi być uporządkowany niemalejąco.

Aby udowodnić pierwszą własność w odniesieniu do algorytmu sortowania bąbelkowego wystarczy zauważyć, że algorytm jedynie zamienia ze sobą w miejscu pewne pary wartości w sekwencji (tablicy), natomiast nie usuwa, nie dodaje, ani nie nadpisuje żadnej wartości w sekwencji wartością, której wcześniej w niej nie było.

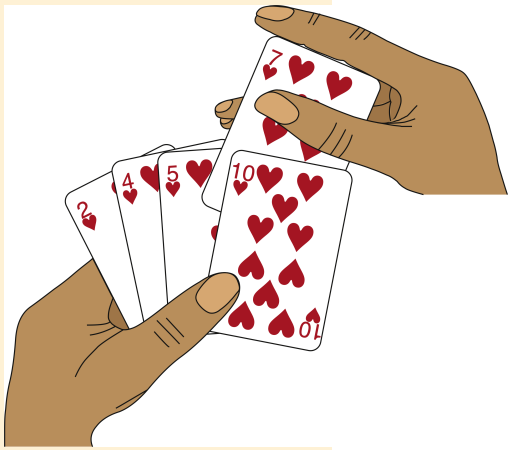
Udowodnienie uporządkowania jest również proste. Można zauważyć, że każdy przebieg pętli wewnętrznej przechodząc elementy od n-tego do $(i+1)$-ego w dół zapewnia, że najmniejszy element w podsekwencji od pozycji i do n znajdzie się na pozycji i.

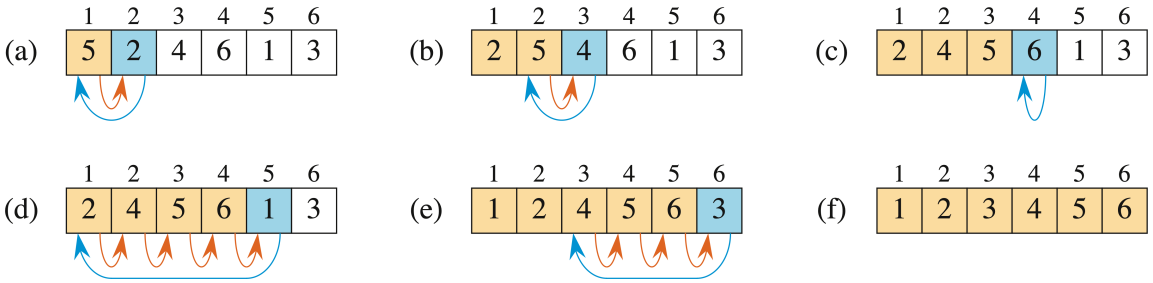
Gdy zatem ta pętla wewnętrzna zostanie wykonana $(n-1)$ razy dla wartości i od 1 do $(n-1)$ to na tych pozycjach w sekwencji znajdzie się $(n-1)$ kolejno najmniejszych elementów. Wtedy, konsekwentnie, element n-ty będzie największy w sekwencji, zatem cała sekwencja będzie uporządkowana.

Sortowanie przez wstawianie

Przyjrzyjmy się z kolei innemu popularnemu algorytmowi sortowania tablicy elementów, zwanego **sortowaniem przez wstawianie**:

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // wstaw A[i] do posortowanej podtablicy A[1 : i − 1]
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```





Jak sugeruje rysunek po prawej, algorytm jest typowo wykorzystywany przez ludzi grających w gry karciane typu brydża, gdzie gracze otrzymują pewną liczbę kart, i zwykle dla wygody „porządkują" je sobie w ręce.

Niezmienniki pętli

Aby udowodnić poprawność sortowania przez wstawianie zauważmy, że algorytm sprawdza kolejne podsekwencje $A[1:i]$, i traktując podsekwencję od pierwszego do $(i-1)$-ego jako uporządkowaną, przesuwa niesprawdzony jeszcze element i-ty (zmienna key) na swoje miejsce we wcześniejszej podsekwencji.

W dowodach poprawności algorytmów przydatne bywa pojęcie **niezmiennika pętli**. W przypadku sortowania przez wstawianie, niezmiennikiem zewnętrznej pętli **for** jest własność, że na początku każdego jej wykonania część tablicy w zakresie $A[1:i-1]$ zawiera te same elementy co tablica wejściowa, ale uporządkowane niemalejąco.

Dowód poprawności z wykorzystaniem niezmiennika wykorzystuje indukcję matematyczną. Należy wykazać, że własność niezmiennika zachodzi przed pierwszym wykonaniem pętli, a następnie, zakładając, że niezmiennik jest spełniony przed dowolnym kolejnym jej wykonaniem, to będzie również spełniony po tym wykonaniu.

Po przeprowadzeniu tego dowodu mamy zagwarantowane, że własność niezmiennika będzie zachowana po ostatnim wykonaniu pętli, a co za tym idzie, zachodzi dla wynikowej struktury danych. W przypadku sortowania przez wstawianie, pętla kończy się gdy wartość i wynosi $(n+1)$, zatem zakres elementów tablicy $A[1:n]$ zawiera oryginalne elementy, ale uporządkowane.

Efektywność algorytmów

Poprawność algorytmu stanowi jego być może najważniejszą własność, ale nie jest jedyną cechą ważną w zastosowaniach praktycznych. W oczywisty sposób, aspekty praktyczne, takie jak zasoby obliczeniowe niezbędne do jego wykonania, są niemal równie ważne. Te zasoby to głównie ilość wykorzystywanej pamięci oraz czas zużyty na obliczenia, a w niektórych przypadkach także inne zasoby takie jak intensywność komunikacji (ilość przesyłanych danych), zużycie energii, itp.

Wykorzystanie tych zasobów w trakcie wykonywania danego algorytmu stanowi o jego **efektywności**, a badanie tej efektywności nazywa się **analizą** algorytmu.

Aby wiarygodnie oszacować efektywność algorytmu potrzebujemy informacji o parametrach i wymaganiach komputera, na którym algorytm będzie wykonywany. To jest ogólnie trudne do przewidzenia, zatem w analizie algorytmów przyjmuje się pewien abstrakcyjny model komputera. W wielu przypadkach jest nim jednoprocesorowy model komputera zwany **RAM** (*Random-Access Machine*). Model RAM zakłada, że wykonanie każdej instrukcji zajmuje tyle samo czasu, że instrukcje wykonywane są ściśle sekwencyjnie, i że istnieją w nim podstawowe typy danych, takie jak liczby całkowite i zmiennoprzecinkowe. Model nie określa dokładnie zestawu instrukcji, ale domyślnie są to podstawowe instrukcje arytmetyczne, logiczne, przesyłanie danych, skoki, i wykonywanie procedur.

Ograniczenia modelu RAM

brak założenia o zakresie liczb integer

brak założenia o zakresie i precyzji liczb zmiennoprzecinkowych

brak uwzględnienia efektów pamięci buforowej (*cache*) i pamięci wirtualnej

Pomimo tych i innych ograniczeń uproszczonego modelu RAM komputera, analiza algorytmów dokonywana na jego bazie zwykle dostarcza bardzo dobrych predykcji efektywności uzyskiwanej na rzeczywistych komputerach.

Często jednak taka analiza okazuje się całkiem skomplikowana, pomimo uproszczeń modelu RAM.

Analiza czasu wykonania algorytmu

Gdybyśmy chcieli określić czas wykonania algorytmu takiego jak jeden z przedstawionych algorytmów sortowania, to moglibyśmy zaimplementować dany algorytm w konkretnym języku programowania, skompilować program na jakimś wybranym komputerze, wykorzystując wybrany kompilator, który wykorzystuje swoje biblioteki, zawierające implementacje podstawowych funkcji wysokopoziomowego języka programowania, oraz ostatecznie uruchomić go dla jakichś przykładowych danych mierząc czasy wykonania.

W oczywisty sposób można oczekiwać, że te czasy będą różne, w zależności od powyższych czynników biorących udział w eksperymencie, a nawet wielokrotne uruchamianie tego samego programu dla tych samych danych może dać różne wyniki.

Zamiast tego, można przeanalizować sam zapis algorytmu w pseudokodzie, licząc wykonanie każdej instrukcji, ewentualnie uwzględniając przyjęte w modelu czasy wykonania różnych instrukcji pseudokodu.

Jednak nawet przy takich założeniach nie możemy wyznaczyć konkretnego czasu wykonania algorytmu, takiego jak algorytm sortowania przez wstawianie, albowiem ten czas będzie zależał on od konkretnych danych.

Obliczenie czasu wykonania sortowania przez wstawianie

Przede wszystkim, czas wykonania algorytmu takiego jak sortowanie przez wstawianie będzie niemal na pewno zależał od długości sortowanej sekwencji. Łatwo sobie wyobrazić, że algorytm wykona mniej operacji, a więc skończy pracę szybciej, dla sekwencji pięciu liczb, niż, na przykład, dla sekwencji pięciu tysięcy.

Można oczekiwać, że czas wykonania algorytmu będzie pewną funkcją długości sortowanej sekwencji. Zamiast więc obliczać ten konkretny czas, naszym celem będzie wyznaczenie kształtu tej funkcji. Zakładając, że sekwencja wejściowa ma długość n, będzie to funkcja zmiennej n.

Ale pojawia się dodatkowa komplikacja. Algorytm INSERTION-SORT wykonuje swoją pętlę wewnętrzną **while** — która przesuwa nowo wstawianą liczbę w lewo, na wcześniejsze pozycje — tylko tak daleko jak to jest potrzebne. Ta liczba wykonań pętli **while** będzie zmienna, i zależna nie tylko od długości sekwencji danych, ale również od konkretnych liczb w sekwencji.

W przypadku sortowania przez wstawianie możemy określić przypadki skrajne.

Sortowanie przez wstawianie — przypadek najlepszy

Najlepszym możliwym przypadkiem jest, gdy sekwencja wejściowa jest od razu poprawnie posortowana. W takim przypadku pętla wewnętrzna **while** nigdy nie będzie wykonywana, a dokładniej będzie tylko jednorazowo sprawdzany jej warunek zakończenia. Zatem będzie jedynie wykonywana pętla zewnętrzna **for**, i wykona ona trzy instrukcje przypisania:

$$T(n) = 3 * n * c_{ass} + n * c_{cmp} + (n-1) * c_{for}$$

gdzie c_{ass} oznacza czas wykonania instrukcji przypisania wartości zmiennej, dla uproszczenia ujednolicony dla maszyny RAM, c_{cmp} jest czasem sprawdzenia warunku pętli **while**, a c_{for} jest czasem wznowienia każdej iteracji pętli **for**.

Ponieważ nie znamy dokładnych wartości czasów wykonania poszczególnych jednostkowych operacji modelu RAM, możemy ten wzór przedstawić w następującej postaci:

$$T(n) = (3 * c_{ass} + c_{cmp} + c_{for}) * n - c_{for} \approx c_1 * n$$

Końcowa aproksymacja przyjmuje $c_1 = (3 * c_{ass} + c_{cmp} + c_{for})$ i pomija jednostkowy czas c_{for}, który przy wielokrotnym wykonaniu pętli sortowania będzie na ogół pomijalny. Zatem czas sortowania jest liniowo zależny od liczby elementów sekwencji.

Sortowanie przez wstawianie — przypadek najgorszy

Najgorszym możliwym przypadkiem dla sortowania przez wstawianie jest, gdy sekwencja wejściowa jest początkowo posortowana malejąco, czyli w kolejności odwrotnej do docelowej. W takim przypadku pętla wewnętrzna **while** będzie zawsze wykonywana w maksymalnym zakresie instrukcji. Pętla zewnętrzna będzie wykonywana dokładnie tak samo jak w przypadku najlepszym.

Czas wykonywania pętli wewnętrznej **while** jest w tym przypadku zmienny, rosnąc od pojedynczej operacji do przejścia całej sekwencji $(n-1)$ liczb na końcu. Oznaczając jako c_{while} czas wykonania całej zawartości pętli **while**, to znaczy dwóch instrukcji przypisania, sprawdzenia warunku, i obliczania indeksów tablicy, całkowity czas wszystkich wykonań tej pętli będzie wynosił

$$T_{while} = (\sum_{i=2}^{n}(i-1)) * c_{while} = (\frac{(n-2)*(n-1)}{2} - 1) * c_{while} = c_1 * n^2 + c_2 * n + c_3$$

czyli będzie kwadratową funkcją (wielomianem) parametru n z pewnymi stałymi.

Ponieważ w zewnętrznej pętli **for** dodatkowo wykonywane są $(n-1)$ razy trzy operacje przypisania, zatem cały czas wykonania algorytmu również jest funkcją kwadratową długości wejścia, z trochę innymi stałymi.[1]

[1] Nieco dokładniejsze obliczenie czasu działania tego algorytmu można znaleźć w podręczniku cytowanym na końcu tej prezentacji.

Znaczenie czasu wykonania algorytmu w różnych przypadkach

Obliczyliśmy czas wykonywania algorytmu sortowania przez wstawianie w dwóch skrajnych przypadkach, i otrzymaliśmy istotnie różne wyniki.

Ale co można powiedzieć o pozostałych przypadkach? Czy te skrajne wyniki mają w ogóle jakieś znaczenie?

Jeśli wziąć pod uwagę różne możliwe zastosowania procedury sortowania w praktyce, to najgorszy możliwy przypadek ma bardzo istotne znaczenie techniczne w systemach typu sterowania, zwłaszcza w systemach czasu rzeczywistego, takich jak sterowanie silnikami, procesami przemysłowymi, elektrowniami atomowymi, itp. W takich systemach ważne jest nie tylko, że zaimplementowany i wdrożony system informatyczny działa poprawnie w czasie testów, ale również ważne jest zapewnienie, że będzie on nadal działał poprawnie nawet przy najgorszym możliwym splocie okoliczności, który może być mało prawdopodobny, ale gdy się zdarzy, może maksymalnie utrudnić (czytaj: wydłużyć) czas zadziałania systemu sterowania.

Zatem analiza zachowania algorytmu w przypadku najgorszym ma sens i bardzo często jest to podstawowa właściwość algorytmów badana i podawana w ich opisach.

Przypadek najlepszy ma dużo mniejsze znaczenie w praktyce. Pomyślmy jaki byłby efekt, gdyby jakiś autor programu, lub firma, reklamowała nowy, innowacyjny program, który *„w najlepszym, skrajnie korzystnym przypadku, działa w czasie pojedynczych milisekund”* (na przykład). Czy to byłaby dobra reklama takiego produktu?
A co w przypadku choćby minimalnie gorszym od tego optymistycznego?

Jednak są sytuacje, w których minimalny czas działania programu w najbardziej optymistycznym przypadku może mieć znaczenie, aczkolwiek te sytuacje są znacznie rzadsze. Wyobraźmy sobie nowoczesny algorytm szyfrowania, którego procedura złamania jest znana, ale jest tak skomplikowana obliczeniowo, że nawet w najlepszym przypadku musi zająć bardzo dużo obliczeń (np. kilka lat czasu procesora). Wtedy ten wynik jest bardzo ważnym parametrem algorytmu szyfrowania, i jego znajomość byłaby cenna dla jego autora/ów. (Jest to jednak sytuacja szczególna, bo zła efektywność czasowa algorytmu łamania szyfru ma znaczenie dla algorytmu szyfrowania.)

A co w pozostałych sytuacjach? Dla wielu zastosowań praktycznych znaczenie ma czas wykonania algorytmu w przypadku średnim, czyli czas pracy uśredniony dla różnych danych, np. generowanych losowo. Taki czas można traktować jako wartość oczekiwaną czasu działania algorytmu, i ta wartość ma znaczenie np. w systemach typu serwisowego. Na przykład, gdyby jakaś firma oferowała obsługę serwisową pewnych zdarzeń, i miała podpisać umowę na świadczenie tych usług z klientem, to dla tego klienta mógłby nie mieć znaczenia żaden pojedynczy czas zadziałania serwisu, a jedynie ich czas sumaryczny, związany ze średnim czasem pojedynczego zadziałania.

Rząd wzrostu funkcji

Określając czas działania algorytmu sortowania przez wstawianie zignorowaliśmy różnice między czasami pojedynczych operacji, zastępując ich sumę jakąś symboliczną stałą. Czas wykonywania algorytmu w przypadku najgorszym można wtedy wyrazić wzorem: $an^2 + bn + c$. W ten sposób nie tylko nie będziemy w stanie obliczyć ostatecznego czasu działania w jakichkolwiek jednostkach czasu, ale również gubimy różnice pomiędzy (stałymi) czasami wykonywania różnych instrukcji, i otrzymany wzór będzie tylko zgrubnym przybliżeniem rzeczywistej wartości.

Zrobimy kolejny krok upraszczający, i we wzorze na czas wykonywania algorytmu pominiemy wszystkie składniki poza elementem o najwyższym rzędzie wielkości, czyli an^2. Oraz ostatecznie pominiemy również stały mnożnik, pozostawiając jedynie wyraz n^2. Ten główny wyraz ze wzoru na czas działania algorytmu, nazywamy jego **rzędem wielkości** lub **rzędem wzrostu** i oznaczamy symbolem greckim Θ. Mówimy, że algorytm ma czas pracy najgorszego przypadku $\Theta(n^2)$ (czytaj: duże Θ od n kwadrat).

Powodem takiego uproszczenia jest fakt, że dla długich sekwencji danych, dla których czas działania algorytmu ma największe znaczenie, ten główny czynnik najlepiej określa czas działania algorytmu.

Na przykład, rozważmy czas sortowania sekwencji miliona liczb algorytmem INSERT-SORT w dwóch przypadkach: najlepszym (sekwencja już poprawnie uporządkowana), i najgorszym (sekwencja uporządkowana odwrotnie). W pierwszym przypadku rząd wielkości $\Theta(n)$ daje wartość miliona 10^6, a w drugim $\Theta(n^2)$ daje wartość 10^{12}.

Dla potrzeb konkretnego obliczenia przyjmijmy bardzo zgrubne przybliżenie czasu wykonania pojedynczej instrukcji pseudokodu jako dziesięciu mikrosekund $10 * 10^{-6}s = 10^{-5}s$, i całkowite czasy w obu przypadkach przemnóżmy jeszcze przez 10 aby odzyskać zgubione mnożniki stałe. Dla pierwszego przypadku uzyskujemy czas działania $10 * 10^6 * 10^{-5}s = 100s$. Natomiast dla drugiego przypadku uzyskujemy $10 * 10^{12} * 10^{-5}s = 10^8s \approx 27778g \approx 1157d \approx 3l$. Różnica wynikająca z różnych rzędów wielkości tych funkcji jest tak duża, że gdybyśmy chcieli dokładniej uwzględnić różnice wartości czasów wykonywania poszczególnych operacji, to nawet gdyby wnosiły one mnożniki typu $10\times, 20\times, 50\times$, to nie zniwelują one dramatycznej różnicy czasów wykonania algorytmu w tych dwóch przypadkach.

Ten przykład ilustruje, dlaczego ma sens wyrażanie i porównywanie czasu pracy algorytmów jako rzędu wielkości Θ, z pominięciem czynników stałych i wyrazów niższego rzędu. Są jednak przypadki, kiedy te elementy mają znaczenie, a mianowicie praca na mniejszych zbiorach danych. Gdy rozważamy taki ograniczony zbiór danych, to algorytm o niższym rzędzie Θ może mieć wyższy rzeczywisty czas pracy.

Uzupełnienie: czas pracy sortowania przez wstawianie w przypadku średnim

Fakt: W przypadku średnim sortowanie przez wstawianie ma czas wykonania będący kwadratem wielkości zbioru, czyli $\Theta(n^2)$, a więc dokładnie takim jak w przypadku najgorszym.

Jakkolwiek w pełni formalny i rygorystyczny dowód tego faktu można znaleźć w literaturze, to na dość nieprecyzyjnym poziomie możemy zauważyć, że w przypadku średnim algorytm wykona w każdym przebiegu pętli zewnętrznej część, nie całość, pętli wewnętrznej, bo takiego rozrzutu wartości możemy oczekiwać w takim przypadku. Jednak ta część będzie zawsze ułamkiem całkowitej liczby elementów, a nie jakąś niewielką, stałą liczbą kroków.

A zatem otrzymujemy wzór na liczbę kroków w postaci: $n * \frac{k}{l}n$ i po wyeliminowaniu stałych otrzymamy zawsze wyrażenie n^2.

Efektywność asymptotyczna i notacja asymptotyczna

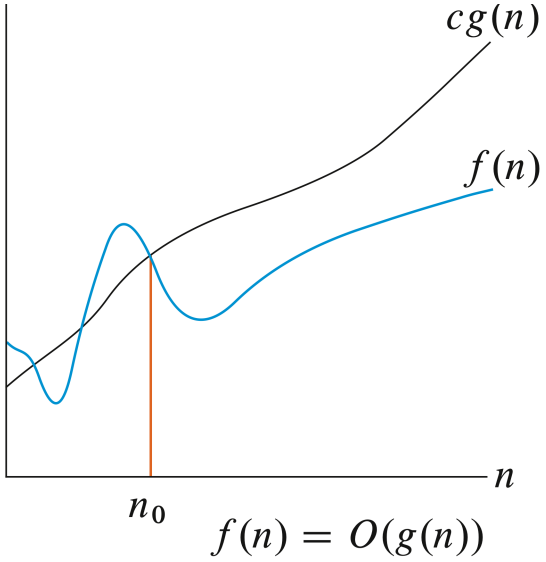
Stwierdziliśmy, że ma sens porównywanie algorytmów ze względu na główny czynnik wyrażenia określającego czas pracy algorytmu, co wiarygodnie określa czas działania algorytmu jedynie dla dostatecznie dużych zbiorów danych. W ten sposób bierzemy pod uwagę **asymptotyczną efektywność** algorytmu, czyli jego zachowanie gdy rozmiar zbioru danych dąży do nieskończoności. Θ-wyrażenia służą do zapisu tej asymptotycznej efektywności i ten zapis nazywamy **notacją asymptotyczną**.

Istnieją jeszcze inne rodzaje notacji asymptotycznej przydatnych w analizie algorytmów. Notacja duże-O wyraża ograniczenie górne asymptotycznego wzrostu funkcji. Za pomocą notacji duże-O możemy określić funkcje, które rosną asymptotycznie co najmniej tak samo szybko jak dana funkcja. A zatem dla algorytmu, którego rząd wzrostu czasu działania wynosi n^2 możemy stwierdzić, że jego asymptotyczna efektywność wynosi $O(n^2)$, ale jednocześnie wynosi ona $O(n^3)$, $O(n^5)$, $O(n^{15})$, itd.

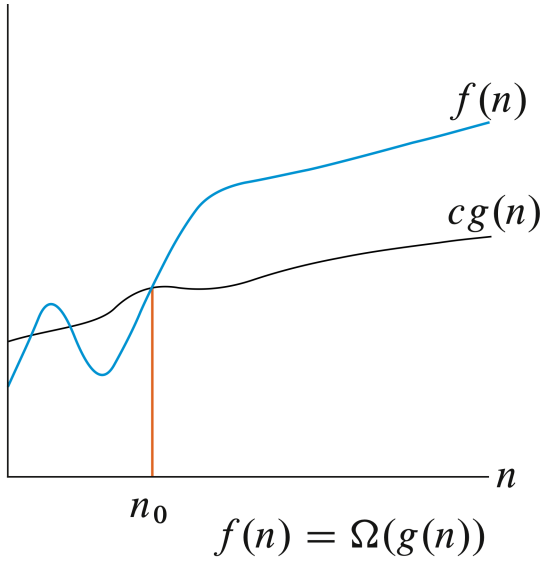
Analogicznie, notacja duże-Ω określa ograniczenie dolne asymptotycznego wzrostu funkcji. A więc dla algorytmu, którego rząd wzrostu czasu działania wynosi n^2 jego asymptotyczna efektywność wynosi $\Omega(n^2)$, ale jednocześnie wynosi $\Omega(n)$, $\Omega(\log n)$, itp.

W odróżnieniu od tych dwóch ograniczeń (O i Ω), wprowadzona wcześniej notacja duże-Θ wyraża dokładną funkcję wzrostu danej wartości. Funkcja n^2 jest tylko $\Theta(n^2)$.

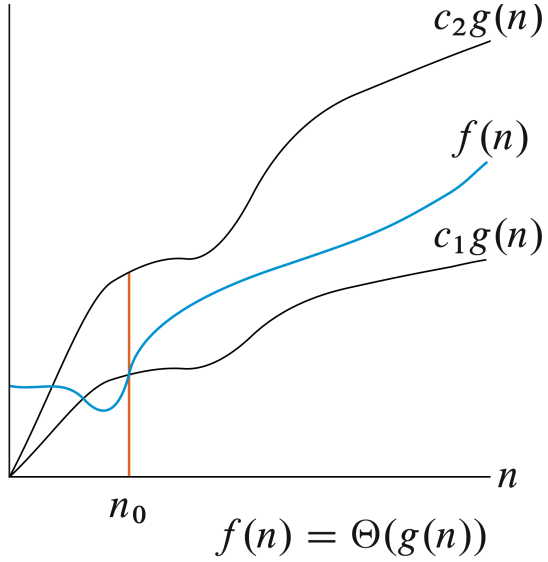
Precyzyjne definicje notacji asymptotycznych



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$



$f(n) = \Theta(g(n))$

$$
\begin{aligned}
O(g(n)) &= \{f(n) : \exists_{c>0\ n_0>0}\ \forall_{n\geq n_0}\ 0 \leq f(n) \leq cg(n)\} \\
\Omega(g(n)) &= \{f(n) : \exists_{c>0\ n_0>0}\ \forall_{n\geq n_0}\ 0 \leq cg(n) \leq f(n)\} \\
\Theta(g(n)) &= \{f(n) : \exists_{c_1>0\ c_2>0\ n_0>0}\ \forall_{n\geq n_0}\ 0 \leq c_1g(n) \leq f(n) \leq c_2g(n)\}
\end{aligned}
$$

Wyrażenie duże-O nazywamy asymptotycznym ograniczeniem górnym.
Wyrażenie duże-Ω nazywamy asymptotycznym ograniczeniem dolnym.
Wyrażenie duże-Θ nazywamy asymptotycznym ograniczeniem ciasnym.

Uwaga: pomimo iż formalnie definiujemy notacje asymptotyczne jako zbiory, potem utożsamiamy je z pojedynczymi funkcjami, np. $2n^2 + 3n + 5 = \Theta(n^2)$.

Własności notacji asymptotycznych

Przechodniość:

$$f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$

Zwrotność:

$$f(n) = \Theta(f(n))$$
$$f(n) = O(f(n))$$
$$f(n) = \Omega(f(n))$$

Symetria

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$
$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

Zachodzi również następujące twierdzenie:

Dla dowolnych funkcji $f(n)$ i $g(n)$:

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

Krótkie podsumowanie — pytania sprawdzające

1. Rozważ algorytm liniowego przeszukiwania tablicy w zakresie $A[1 : n]$. Napisz pseudokod procedury implementującej ten algorytm. Argumentami będą: tablica A, rozmiar n, i poszukiwana liczba x. Procedura zwróci indeks znalezionego elementu, lub wartość -1 gdy go nie ma.

2. Sformułuj niezmiennik pętli algorytmu z poprzedniego pytania, i z jego wykorzystaniem udowodnij poprawność algorytmu.

3. Sformułuj niezmiennik pętli wewnętrznej dla algorytmu Bubble-Sort, i udowodnij jego zachowanie w algorytmie.

4. Następnie wykorzystując warunek zakończenia niezmiennika z poprzedniego pytania, sformułuj niezmiennik pętli zewnętrznej algorytmu Bubble-Sort, i z jego wykorzystaniem udowodnij poprawność całego algorytmu.

5. Rozważ algorytm przeszukiwania liniowego tablicy z pytania numer 1. Ile elementów tablicy procedura przeszuka w najgorszym przypadku? A ile w przypadku średnim, zakładając losowe rozmieszczenie elementów, i zakładając, że element znajduje się w tablicy? Zapisz Θ-wyrażenia czasu działania algorytmu w obu przypadkach.

Literatura i materiały pomocnicze

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest, Clifford Stein: Wprowadzenie do algorytmów, PWN, 2024