Algorytmy i struktury danych – Matematyka III sem.

30 godz. wykł. / 15 godz. ćw. / 15 godz. projekt
dr inż. Paweł Syty, 413GB, sylas@mif.pg.gda.pl, http://sylas.info

Literatura

T.H. Cormen i inni, *Wprowadzenie do algorytmów*, WNT 2005
J. Bentley, *Perełki oprogramowania*, wyd. II, WNT 2001
D. Harel, *Rzecz o istocie informatyki. Algorytmika*, WNT 2001

Materiały dydaktyczne

http://moodle.pg.gda.pl
Kurs „Algorytmy i struktury danych (Mat)”

Algorytm – definicja, cechy, poprawność

Obliczenie – znalezienie rozwiązania danego zagadnienia w oparciu o dostępne dane i z użyciem algorytmu.

Algorytm – poddający się interpretacji skończony zbiór instrukcji wykonania zadania mającego określony stan końcowy dla każdego zestawu danych wejściowych. Innymi słowy – algorytm jest ciągiem kroków obliczeniowych, prowadzących do przekształcenia danych wejściowych w wyjściowe.

Własności algorytmu

- może korzystać z danych wejściowych
- prowadzi do jednej lub większej liczby danych wyjściowych
- wskazana własność ogólności
- rozwiązanie zawsze osiągnięte i to w skończonej liczbie kroków
- każdy możliwy przypadek przewidziany
- każdy krok jednoznacznie i precyzyjnie zdefiniowany
- korzysta z operacji podstawowych (plus iteracje i struktury warunkowe)

Poprawność algorytmów

Algorytm nazywamy poprawnym, jeżeli dla dowolnych poprawnych danych wejściowych, osiąga on punkt końcowy i otrzymujemy poprawne wyniki.

Cechy algorytmu poprawnego

- Częściowa poprawność. Algorytm nazywamy częściowo poprawnym, gdy prawdziwa jest następująca implikacja: jeżeli algorytm osiągnie koniec dla dowolnych poprawnych danych wejściowych, to dane wyjściowe będą spełniać warunek końcowy.

- Własność określoności obliczeń. Algorytm posiada tę własność, jeżeli dla dowolnych poprawnych danych wejściowych, działanie algorytmu nie zostanie przerwane.

- Własność stopu. Algorytm posiada tę własność, jeżeli dla dowolnych poprawnych danych wejściowych, algorytm nie będzie działał w nieskończoność.

Dowodzenie poprawności algorytmów – metoda niezmienników Floyda

- wyróżnić newralgiczne punkty w algorytmie
- określić warunki (niezmienniki), jakie muszą być spełnione w każdym wyróżnionym punkcie
- udowodnić poprawność kolejnych warunków, zakładając poprawność warunków poprzedzających
- własność stopu udowodnić np. metodą liczników iteracji lub metodą malejących wielkości

Przykłady sformułowania problemów algorytmicznych

Sortowanie
Wejście: tablica T zawierająca n elementów (a_1, a_2, . . . , a_n) typu porządkowego.
Wyjście: tablica o tych samych elementach, ale uporządkowana niemalejąco (czyli uporządkowana permutacja ciągu wejściowego).

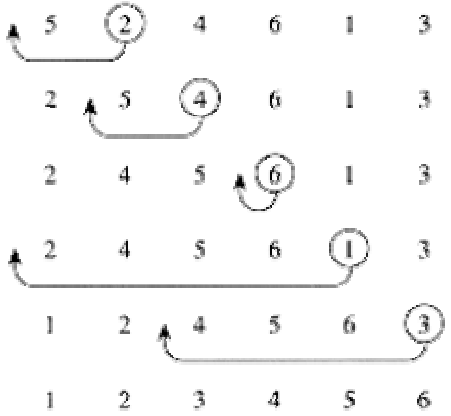
Wyszukiwanie
Wejście: posortowana, n-elementowa tablica liczbowa T oraz liczba p.
Wyjście: liczba naturalna, określająca pozycję elementu p w tablicy T, bądź zero, jeżeli element w tablicy nie występuje.

Generowanie podciągu
Wejście: dwie liczby całkowite m i n, gdzie $m <= n$.
Wyjście: posortowana lista m losowych liczb całkowitych z przedziału $1 . . . n$, wśród których żadna nie powtarza się dwukrotnie.

Problem komiwojażera
Wejście: n punktów (miast), odległości pomiędzy miastami ($d_{ij} \mid i, j = 1, 2, . . . , n$);
Wyjście: trasa komiwojażera przez wszystkie miasta (przy czym dopuszczalna jest tylko jedna wizyta w każdym mieście – permutacja miast) o najmniejszej sumie odległości.

Przykład algorytmu sortującego – sortowanie przez wstawianie

- efektywny algorytm sortowania niewielkiej liczby elementów
- działa na zasadzie porządkowania talii kart



Schemat działania algorytmu:

1. utwórz zbiór elementów posortowanych i przenieś do niego dowolny element ze zbioru nieposortowanego
2. weź dowolny element ze zbioru nieposortowanego
3. wyciągnięty element porównuj z kolejnymi elementami zbioru posortowanego póki nie napotkasz elementu równego lub mniejszego, lub nie znajdziesz się na początku zbioru uporządkowanego
4. wyciągnięty element wstaw w miejsce gdzie skończyłeś porównywać
5. jeśli zbiór elementów nieuporządkowanych jest niepusty, wróć do punktu 2

Pseudokod

```
// wejście: n-elementowa tablica T[1... n]
pętla od i = 2 do n
    // niezmiennik: fragment T[1... i-1] jest posortowany
    // cel: przesunąć element T[i] w dół na właściwe miejsce
    pętla od j = i do 2
        jeżeli T[j] < T[j-1]
            zamień T[j] z T[j-1]
            // realizacja zamiany: tmp = T[j]; T[j] = T[j-1]; T[j-1] = tmp
```

Analiza algorytmów

Rozmiar danych wejściowych

- dla sortowania: liczba elementów do posortowania (*n*)
- mnożenie liczb całkowitych: całkowita liczba bitów
- operacje na grafach: liczba wierzchołków

Czas działania algorytmu (złożoność czasowa)

- liczba elementarnych operacji (np. podstawienie, porównanie, prosta operacja arytmetyczna), potrzebnych do wykonania algorytmu
- najczęściej jest funkcją rozmiaru danych wejściowych

Dla sortowania przez wstawianie:

```
1 pętla od i = 2 do n
2   pętla od j = i do 2
3       jeżeli T[j] < T[j-1]
4           tmp = T[j]
5           T[j] = T[j-1]
6           T[j-1] = tmp
7       w przeciwnym wypadku zakończ pętlę wewnętrzną
```

Oznaczmy przez t liczbę porównań w wierszu 3. Maksymalnie będzie ich $(n-1)(n-1)$ (bo mamy maksymalnie $n-1$ wykonań pętli zewnętrznej pomnożone przez $n-1$ (w uproszczeniu; ściśle – w tej implementacji algorytmu będzie ich mniej, jest to jednak sprawa drugorzędna) wykonań pętli wewnętrznej; ten przypadek wystąpi dla „pechowego przypadku" tablicy wejściowej posortowanej malejąco). Minimalna liczba porównań to $n-1$ (bo mamy $n-1$ wykonań pętli zewnętrznej pomnożone przez jedno, negatywne porównanie; ten przypadek wystąpi dla tablicy wejściowej już posortowanej rosnąco – wewnętrzna pętla w takim przypadku nie musi nic robić, poza jednym porównaniem).
Liczba wykonań instrukcji w wierszach 4–6 jest zależna od t. W skrajnym przypadku może to być aż $(n-1)(n-1)$ instrukcji na każdy wiersz, z drugiej strony instrukcje te się mogą się w ogóle nie wykonać, w przypadku wszystkich negatywnych porównań (tablica wstępnie posortowana rosnąco).

Załóżmy dla uproszczenia, że pojedyncze instrukcje porównania i przypisania wykonują się w tym samym czasie c. Procedura sortująca wykona się maksymalnie w czasie $4c(n-1)(n-1)$ (przypadek pesymistyczny – dane posortowane odwrotnie; funkcja kwadratowa względem n), natomiast minimalny czas wykonania to $c(n-1)$ (przypadek optymistyczny – dane posortowane; funkcja liniowa względem n).

Przypadek średni (oczekiwany) będzie w tym przypadku zbliżony do przypadku pesymistycznego (też będzie to funkcja kwadratowa).

Wymagana pamięć (złożoność pamięciowa)

- liczba podstawowych komórek pamięci, wykorzystywanych przez algorytm
- najczęściej jest funkcją rozmiaru danych wejściowych

Dla sortowania przez wstawianie: wymagana pamięć wynosi *n* (elementy tablicy) + 1 (zmienna pomocnicza `tmp`) – zależność liniowa względem *n*.

Oszacowania asymptotyczne

Notacja Θ (Theta)

$f=\Theta(g)$, jeżeli $\exists\, c_1,c_2\in R^+\ \exists\, x_0\in R^*\ \forall\, x \geq x_0 \quad c_1\cdot g(x) \leq f(x) \leq c_2\cdot g(x)$ (mówimy, że ***f jest theta od g***)

Inaczej: *Funkcje f i g są tego samego rzędu.*

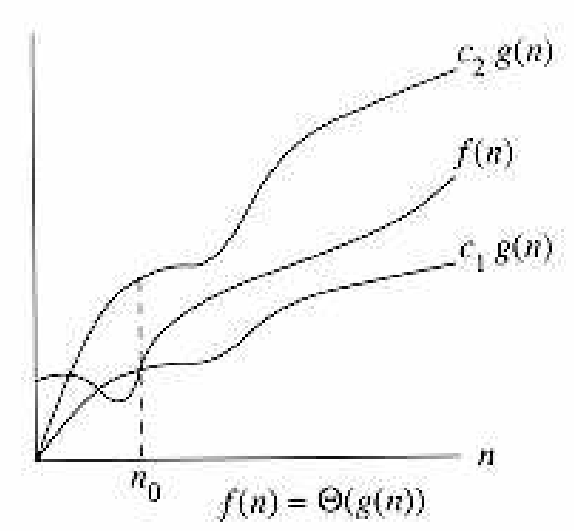
Jeszcze inaczej: $\lim_{x\to\infty} \frac{f(x)}{g(x)} = c \neq 0$

R^+- zbiór liczb rzeczywistych dodatnich
$R^* = R^+ \cup \{0\}$

Niech $g:R^*\to R^*$ i $f:R^*\to R$ będą funkcjami.

Przykład: $\frac{1}{2} n^2 - 3n = \Theta(n^2)$.
Uzasadnienie:
Szukamy stałych c_1 i c_2 oraz n_0 takich, że $c_1 n^2 <= \frac{1}{2} n^2 - 3n <= c_2 n^2$ dla każdego $n > n_0$.
Dzieląc przez n^2 otrzymujemy: $c_1 <= \frac{1}{2} - 3/n <= c_2$.
Nierówność ta jest spełniona dla wszystkich n>6, np. gdy c_1=1/14 i c_2= ½. Zatem : $\frac{1}{2} n^2 - 3n = \Theta(n^2)$.

Przykład: $6n^3 \neq \Theta(n^2)$.
Uzasadnienie:
Załóżmy, że istnieją stałe c_2 oraz n_0 takie, że $6n^3 <= c_2 n^2$ dla każdego $n > n_0$. Ale wtedy $6n <= c_2/6$ co nie może być prawdą dla dowolnie dużych *n*, ponieważ c_2 jest stałą.



Notacja Θ asymptotycznie ogranicza funkcję od góry i od dołu.

Oszacowania Θ używamy dla określenia pesymistycznej złożoności obliczeniowej algorytmów. Na przykład – pesymistyczny czas wykonania sortowania przez wstawianie (czyli pesymistyczna złożoność obliczeniowa tego algorytmu) jest rzędu $\Theta(n^2)$.

Intuicyjnie, składniki niższego rzędu mogą być pominięte, gdyż są mało istotne dla dużych *n*. Składniki wyższego rzędu są wtedy dominujące.
Przykład: dowolna funkcja kwadratowa jest rzędu $\Theta(n^2)$, tzn. $an^2 + bn + c = \Theta(n^2)$.
Ogólnie, dowolny wielomian $p(n) = \sum_{i=0}^{d} a_i n^i = \Theta(n^d)$, o ile a_i są stałymi oraz $a_d > 0$.

Funkcję stałą określamy jako $\Theta(n^0)$ lub $\Theta(1)$.

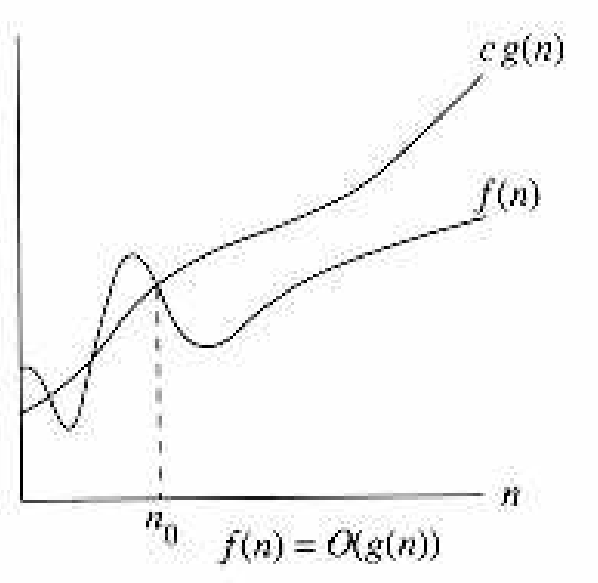
Notacja O (dużego O)

$f=\mathbf{O}(g)$, jeżeli $\exists\, c\in R^{+}\ \exists\, x_0\in R^{+}\ \forall\, x \geq x_0 \quad f(x) \leq c\cdot g(x)$ (mówimy, że ***f jest duże o od g***)

Inaczej: *Funkcja f nie rośnie szybciej niż g.*

Jeszcze inaczej: $\lim_{x\to\infty} \frac{f(x)}{g(x)} = c \geq 0$

Przykład: ½ n^2 - 3n = O(n^2), ale również np. 5n +6 = O(n^2).



Notacja O określa asymptotyczną granicę górną. Korzystamy z niej, żeby oszacować funkcję z góry, z dokładnością do stałego współczynnika.

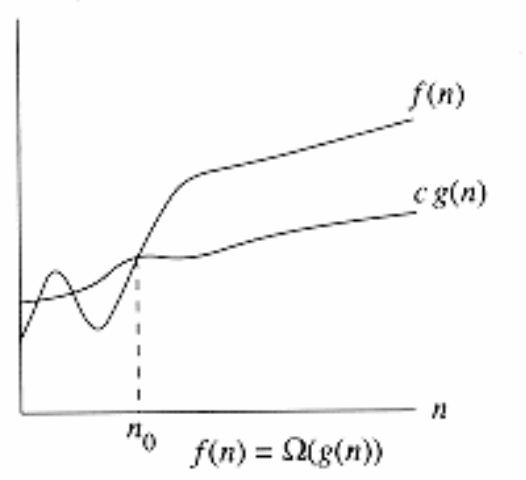
Można powiedzieć, że czas działania algorytmu sortowania przez wstawianie jest rzędu O(n^2) – czyli algorytm ten nie zostanie nigdy wykonany wolniej niż w czasie kwadratowym (ale może być wykonany szybciej – np. w czasie liniowym).

Notacja Ω (Omega)

$f=\Omega(g)$, jeżeli $\exists\, c\in R^+ \exists\, x_0\in R^* \forall\, x \geq x_0 \quad g(x) \leq c\cdot f(x)$ (mówimy, że ***f jest duże omega od g***)

Inaczej: *Funkcja f rośnie tak samo lub szybciej niż g.*

Jeszcze inaczej: $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \begin{cases} c > 0 \\ \infty \end{cases}$



Notacja Ω określa asymptotyczną granicę dolną.

Można powiedzieć, że czas działania algorytmu sortowania przez wstawianie jest rzędu $\Omega(n)$ – czyli algorytm ten nie zostanie nigdy wykonany szybciej niż w czasie linowym.

Notacja o (małego o)

$f=\mathbf{o}(g)$, jeżeli $\forall\ c\in R^+\ \exists\ x_0\in R^+\ \forall\ x \geq x_0 \quad f(x) < c\cdot g(x)$ (mówimy, że ***f jest małe o od g***)

Inaczej: *Funkcja f rośnie znacznie wolniej niż g.*

Jeszcze inaczej: $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$

Przykład: $2n = o(n^2)$, ale $n^2 \neq o(n^2)$.

Notacja ω (małego omega)

$f=\boldsymbol{\omega}(g)$, jeżeli $\forall\ c\in R^+\ \exists\ x_0\in R^+\ \forall\ x \geq x_0 \quad g(x) < c\cdot f(x)$ (mówimy, że ***f jest małe omega od g***)

Inaczej: *Funkcja f rośnie o wiele szybciej niż g.*

Jeszcze inaczej: $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty$

Przykład: $n^2/2 = \omega(n)$, ale $n^2/2 \neq \omega(n^2)$.

Własności oszacowań

Twierdzenie. Dla każdych dwóch funkcji $f(n)$ i $g(n)$ zachodzi zależność $f(n) = \Theta(g(n))$ wtedy i tylko wtedy, gdy $f(n) = O(g(n))$ i $f(n) = \Omega(g(n))$.

Przykład: Z tego, że $\frac{1}{2} n^2 - 3n = \Theta(n^2)$ wynika, że $\frac{1}{2} n^2 - 3n = O(n^2)$ oraz $\frac{1}{2} n^2 - 3n = \Omega(n^2)$.

Przechodniość:
$f(n) = \Theta(g(n))$ i $g(n) = \Theta(h(n))$ implikuje $f(n) = \Theta(h(n))$
$f(n) = O(g(n))$ i $g(n) = O(h(n))$ implikuje $f(n) = O(h(n))$
$f(n) = \Omega(g(n))$ i $g(n) = \Omega(h(n))$ implikuje $f(n) = \Omega h(n))$
$f(n) = o(g(n))$ i $g(n) = o(h(n))$ implikuje $f(n) = o(h(n))$
$f(n) = \omega(g(n))$ i $g(n) = \omega(h(n))$ implikuje $f(n) = \omega(h(n))$

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

Symetria:
$f(n) = \Theta(g(n))$ wtedy i tylko wtedy, gdy $g(n) = \Theta(f(n))$

Symetria transpozycyjna:
$f(n) = O(g(n))$ wtedy i tylko wtedy, gdy $g(n) = \Omega(f(n))$
$f(n) = \Omega(g(n))$ wtedy i tylko wtedy, gdy $g(n) = O(f(n))$

Notacja asymptotyczna w równaniach

Gdy notacja asymptotyczna pojawia się po prawej stronie równania, tak jak do tej pory (np. $n = O(n^2)$), oznacza to przynależność: $n \in O(n^2)$.

Z kolei, np. równanie: $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ oznacza, że $\Theta(n)$ jest pewną anonimową funkcją (o pomijalnej nazwie), tzn, $2n^2 + 3n + 1 = 2n^2 + f(n)$, gdzie $f(n)$ jest funkcją należącą do zbioru $\Theta(n)$. W tym przypadku $f(n) = 3n+1 = \Theta(n)$.

Użycie notacji asymptotycznej pozwala więc na uproszczenie równań poprzez wyeliminowanie nieistotnych jego składników.