O notacji asymptotycznej w analizie algorytmów

Łukasz Kowalik

14 września 2007

Załóżmy, że mamy dwa algorytmy dla tego samego problemu i oba są poprawne. Który z nich jest szybszy? Od razu powstaje pytanie: jak zmierzyć czas działania algorytmu? Choć informacja że nasz algorytm działa tyle a tyle minut na pewnym komputerze dla pewnych danych wejściowych daje nam jakieś pojęcie o jego skuteczności, jest to jednak informacja bardzo niepełna. Po pierwsze nie mamy gwarancji że na innych danych (nawet tego samego rozmiaru) które mogą się pojawić nasz algorytm nie będzie działał znacznie wolniej. Po drugie, nie mamy żadnego wyobrażenia na temat zachowania się algorytmu dla większych danych. Po trzecie wreszcie, wraz z postępem technologii pojawiają się kolejne generacje komputerów i tego typu informacja szybko się dezaktualizuje.

Rozważmy konkretny przykład algorytmu. Dana jest tablica liczb naturalnych $a[1..n]$, uporządkowana niemalejąco. Należy sprawdzić, czy a zawiera daną liczbę x. Użyjemy klasycznego algorytmu wyszukiwania binarnego. Z grubsza rzecz biorąc, naśladuje on wyszukiwanie numeru w książce telefonicznej, ale takiej, w której nie mamy żadnej pewności czy istnieją np. jakiekolwiek nazwiska zaczynające się na a. Zaczynamy od porównania x z $a[\lfloor n/2 \rfloor]$. Jeśli mieliśmy szczęście, znaleźliśmy x i kończymy. Jeśli $x < a[\lfloor n/2 \rfloor]$, wiemy że możemy ograniczyć poszukiwania do $a[1..\lfloor n/2 \rfloor - 1]$, a w przeciwnym przypadku do $a[\lfloor n/2 \rfloor + 1..n]$. Powiedzmy, że zdarzył się ten ostatni przypadek. Mamy do czynienia z takim samym problemem, tylko o połowę mniejszym. Postępujemy więc tak samo jak poprzednio, czyli porównujemy x z elementem środkowym ciągu $a[\lfloor n/2 \rfloor + 1], \ldots, a[n]$. Ponownie albo znajdujemy x, albo redukujemy przetrzeń poszukiwań o połowę, itd. W końcu może się zdarzyć, że przestrzeń poszukiwań stanie się pusta — wtedy kończymy ze stwierdzeniem, że x nie ma w tablicy.

Jeśli chcemy uniezależnić naszą miarę czasu działania algorytmu od konkretnych danych, to innymi słowy chcemy rozważyć wszystkie możliwe zestawy danych. Zwykle algorytmy działają dłużej dla większych danych. Stąd naturalny pomysł: jako miarę złożoności (szybkości) algorytmu przyjmiemy maksymalny czas działania algorytmu dla danych rozmiaru n. Jest to tzw. złożoność pesymistyczna. Widzimy, że jest to funkcja postaci $T : \mathbf{N} \to \mathbf{N}$. Spróbujmy znaleźć funkcję T dla wyszukiwania binarnego, ale zamiast czasu będziemy zliczać porównania x i elementów tablicy a. Jest dosyć jasne, że porównań jest najwięcej, gdy x nie ma w tablicy — algorytm kończy działanie dopiero gdy przedział jest pusty. Widzimy więc, że spełniona jest zależność rekurencyjna $T(n) = 1 + T(\lfloor n/2 \rfloor)$ oraz $T(0) = 0$. Podstawmy $n = 2^k$. (Gdy n nie jest potęgą 2 bierzemy pierwszą taką potęgę większą od n). Mamy wtedy $T(2^k) = 1 + T(2^{k-1}) = 1 + 1 + T(2^{k-2}) = \ldots = k + 1$. Dostaliśmy zatem $T(n) = \log_2 n + 1$ gdy n jest potęgą 2 i $T(n) = \lceil \log_2 n \rceil + 1$ w ogólności.

Funkcja $T(n)$ którą określiliśmy przed chwilą, unika wszystkich trzech wymienionych wad „podejścia ze stoperem”: nie zależy od konkretnych danych, możemy obliczyć jej wartość dla dowolnych, nawet bardzo dużych wartości n i nie zależy od właściwości sprzętu. Pozostał do wyjaśnienia jeden problem: czy ma jakiś związek z „prawdziwym” czasem wykonania algorytmu na konkretnym komputerze? Na szczęscie ma. Na każdy program uruchamiany w komputerze możemy patrzeć jak na ciąg tzw. instrukcji maszynowych (np. porównanie, dodanie dwóch liczb, przypisanie itd). Każda taka instrukcja maszynowa ma ściśle określony czas wykonania (powiedzmy, w nanosekundach). Patrząc na algorytm wyszukiwania binarnego widzimy, że pomiędzy dowolnymi dwoma porównaniami wykonywana jest pewna ograniczona liczba instrukcji maszynowych. Ich czas możemy więc ograniczyć przez pewną stałą (powiedzmy, że dla naszego komputera jest to 1,23 milisekundy). Otrzymalibyśmy jako wniosek, że „prawdziwy” czas działania algorytmu nie przekracza $1,23 \cdot (\lceil \log_2 n \rceil + 1)$ milisekund. Nasza funkcja $T(n)$ reprezentuje więc „prawdziwy” czas, tyle że z dokładnością do stałej przez którą trzeba ją przemnożyć. Ta stała jest z wielu powodów kłopotliwa – jak widzieliśmy, zależy od użytego sprzętu a jej dokładne obliczenie byłoby niezwykle żmudne.

W wielu przypadkach stała przed funkcją złożoności jest po prostu nieistotna. Wyobraźmy sobie, że wyszukiwanie binarne porównujemy z naiwnym algorytmem, który po prostu sprawdza kolejno komórki tablicy od $a[1]$ do $a[n]$. Powiedzmy, że udało nam się oszacować czas algorytmu naiwnego na co najmniej $0,3n$ milisekund. Choć $0,3 < 1,23$ to np. dla $n = 1000$ mamy $1,23 \cdot (\lceil \log_2 n \rceil + 1) = 1,23 \cdot 11 < 0,3 \cdot 1000$. Być może dla pewnych małych wartości n algorytm naiwny jest lepszy od wyszukiwania binarnego, te wartości nas jednak nie interesują — wtedy oba algorytmy działają bardzo szybko. Dlatego podając złożoność algorytmu, stałą tę pomija się. Z tych samych powodów pomijamy także składnik „+1" (nawet gdyby było to „+100").

W podręcznikach algorytmiki złożoność czasową wyszukiwania binarnego zapisuje się jako $O(\log n)$. Jest to *notacja asymptotyczna* lub *notacja dużego O*. Chyba czas już na dokładną definicję. Mówimy, że $f(n) = O(g(n))$, gdy istnieją takie stałe c i n_0, że dla $n \geq n_0$ zachodzi $f(n) \leq c \cdot g(n)$. Dla przykładu, $1,23 \cdot (\lceil \log_2 n \rceil + 1) = O(\log_2 n)$, gdyż możemy wziąć np. $c = 5$ i $n_0 = 2$. Oto więcej przykładów: $\frac{1}{5}n^2 + 100n = O(n^2)$, $n = O(n^2)$, $100 \log_2 n + 10 = O(n)$, $\log_{10} n = O(\log_2 n)$, $n2^n = O(2.001^n)$. Podumowując, pisząc, że złożoność czasowa jakiegoś algorytmu jest $O(g(n))$ mamy na myśli, że czas jego wykonania dla danych rozmiaru n nie przekracza $f(n)$ sekund, dla pewnej funkcji f spełniającej $f(n) = O(g(n))$.

Czy można wyszukiwać liczby w posortowanej tablicy lepiej niż wyszukiwaniem binarnym? W dowolnej chwili wykonania algorytmu istnieje pewien przedział $[i, j]$ indeksów a, w którym liczba x może się znajdować. Po wykonaniu porównania x z pewnym elementem tablicy $a[k]$ ten przedział zmiejsza się do $[i, k-1]$ lub $[k+1, j]$. Widzimy jednak, że dla konkretnego algorytmu i tablicy a możemy tak złośliwie dobrać x, żeby przy każdym porównianiu x był w większym z tych dwóch przedziałów, a więc przedział, w którym x może się znajdować zmniejszał się co najwyżej dwukrotnie. Ta argumentacja prowadzi do dowodu, że każdy algorytm musi wykonać co najmniej $\log_2 n + 1$ porównań w najgorszym przypadku. Wiele osób błędnie pisze w takiej sytuacji „złożoność dowolnego takiego algorytmu jest co najmniej $O(\log n)$", choć przecież w definicji „dużego O" mowa jest o górnym ograniczeniu. Takie zwyczaje irytowały Donalda Knutha, dlatego w 1976 roku wprowadził analogiczne notacje Ω i Θ. Mianowicie $f(n) = \Omega(g(n))$, gdy istnieją takie stałe c i n_0, że dla $n \geq n_0$ zachodzi $f(n) \geq c \cdot g(n)$. Poprawnie jest więc powiedzieć, że każdy algorytm wyszukujący w posortowanej tablicy daną liczbę za pomocą porównań ma złożoność $\Omega(\log n)$. Możemy też powiedzieć, że istnieją dane, dla których algorytm naiwny działa w czasie $\Omega(n)$. Wreszcie, w sytuacji gdy $f(n) = O(g(n))$ i $f(n) = \Omega(g(n))$, piszemy $f(n) = \Theta(g(n))$. Możemy więc powiedzieć, że złożoność pesymistyczna wyszukiwania binarnego wynosi $\Theta(\log n)$.