

Uważne porównywanie jest szczególnie istotne w przypadku kosztowniejszych algorytmów. Autor artykułu poprawiał kiedyś pewien program, który był używany przez parę lat, przy czym za każdym uruchomieniem dla typowych danych obliczenia trwały około pół godziny. Tymczasem przesunięcie jednej instrukcji w kodzie spowodowało, że czas działania zmniejszył się do 4 sekund! Po prostu jeden z iloczynów skalarnych był obliczany niepotrzebnie w najbardziej zagnieżdżonej z trzech pętli i wystarczyło jego obliczenie przenieść na wyższy poziom, zmniejszając koszt z  $O(n^4)$  do  $O(n^3)$  – proszę zgadnąć, ile wynosiło  $n$  dla typowych danych.

\*Instytut Informatyki,  
Uniwersytet Warszawski

Czasami sprawa jest prosta: pod każdym względem algorytm A przewyższa algorytm B. Obliczanie rozwiązań układu równań liniowych za pomocą wzorów Cramera praktycznie zawsze będzie gorsze od metody eliminacji Gaussa. Zazwyczaj jednak sytuacja jest inna. Dla jednych danych lepszy jest algorytm A, a dla innych B. Informatycy radzą sobie z tymi problemami, wprowadzając pojęcie *złożoności średniej*. Zakładamy zatem pewien rozkład prawdopodobieństwa na przestrzeni danych (najczęściej jednostajny) i jako miarę złożoności algorytmu przyjmujemy wartości zmiennej losowej wyrażającej liczbę operacji, które wykonamy dla tych danych. Wtedy *wartość oczekiwana* tej zmiennej losowej jest poszukiwaną funkcją kosztu – nazywamy ją kosztem średnim algorytmu (lub problemu, gdy bierzemy jeszcze minimum po wszystkich średnich kosztach algorytmów rozwiązujących dany problem).

Rozważmy tę kwestię na bardzo praktycznym przykładzie. Mamy dwie wersje niesamowicie ważnego algorytmu, zwanego algorytmem wyszukiwania binarnego – można powiedzieć – będącego podstawą informatyki. Za pomocą wyszukiwania binarnego przeszukujemy większość baz danych. Każda oszczędność jest tu ważna. Podstawowa idea jest jasna: w posortowanym niemalejąco ciągu danych  $A[l..p]$  szukamy zadanej wartości  $x$ , a dokładniej indeksu, pod którym występuje. Szukamy zatem takiego  $i$ ,  $l \leq i \leq p$ , żeby nasza poszukiwana wartość  $x$  była równa  $A[i]$ . Badamy więc „środek” przedziału  $s = \lfloor (l+p)/2 \rfloor$  i jeśli  $x \leq A[s]$ , to koncentrujemy się na lewej „połowie” przedziału, a jeśli  $x > A[s]$ , to na prawej. Powtarzamy to tak długo, aż przedział zrobi się jednoelementowy, i wtedy sprawdzamy, czy szukana wartość tam właśnie się znajduje. Kod w Pascalu podanego algorytmu wyglądając może tak:

```
l:=1; p:=n;
while (l < p) do
  begin
    s:=(l+p) div 2;
    if (x > A[s]) then l:=s+1
      else p:=s;
  end;
if (x = A[l]) then wynik:=l
  else wynik:=-1; // w ten sposób
  // zakomunikujemy porażkę
```

## Średnio lepiej

Piotr CHRZAŚTOWSKI\*

Każdy problem algorytmiczny można rozwiązać na wiele sposobów. Gdy zależy nam na wyborze jednego z takich rozwiązań, musimy odpowiedzieć sobie na pytanie, co to znaczy, że jeden algorytm jest lepszy od drugiego, czyli znaleźć wspólną *miarę* oceny ich jakości. Trudność polega na tym, że nie do końca wiadomo, co brać pod uwagę. Algorytm szybki, czyli wykonujący stosunkowo niewiele działań, może być wymagający pamięciowo, czyli potrzebować dużo pamięci komputera. Algorytm działający wolno dla dużych danych może świetnie poradzić sobie z danymi małymi. Dalej, nie tylko rodzaj mierzonej wielkości może mieć wpływ na naszą ocenę algorytmu, ale i rodzaj danych. Może się chociażby zdarzyć, że dla większości danych jeden z algorytmów zdecydowanie przeważa nad drugim, i tylko dla stosunkowo niewielkiej ich liczby jest gorszy.

Mówienie o „środku” i „połowach” jest drobnym, ale dopuszczalnym nadużyciem: w rzeczywistości zazwyczaj będzie to prawie środek i prawie połowa (z dokładnością do jedynki).

Możemy jednak zadać pytanie: w każdym obrocie pętli już mamy element  $A[s]$  w ręce, więc może lepiej byłoby sprawdzić, czy przypadkiem zachodzi  $x = A[s]$ ? Głupio byłoby czekać, aż przedział zrobi się jednoelementowy.

Zatem konkurencyjna wersja algorytmu wyszukiwania binarnego jest taka:

```
l:=1; p:=n; s:=(l+p) div 2;
while (l < p) and (A[s] <> x) do
  begin
    if (x > A[s]) then l:=s+1
      else p:=s-1; // tu wiemy, że
      // A[s]<>x
    s:=(l+p) div 2;
  end;
if (x = A[s]) then wynik:=s
  else wynik:=-1; // w ten sposób
  // zakomunikujemy porażkę
```

Poza nieistotną z punktu widzenia złożoności kolejnością wyznaczania środka przedziału kod ten od poprzedniego różni się w zasadzie tylko tym, że w dozorze pętli mamy dodatkowo jeszcze jeden warunek do sprawdzenia i koniunkcję do wykonania. Teraz pętla zakończy wykonywać się wcześniej, jeśli tylko dostatecznie szybko wpadniemy na  $x$  znajdujące się gdzieś w środku przedziału  $[l..p]$  – wydaje się więc, że możemy na tym sporo zyskać. Czy mniejsza liczba wykonanych pętli zrekompensuje nam jednak lekkie zwiększenie kosztu wykonania pojedynczego obrotu związane ze skomplikowaniem dozoru pętli?

Ustalmy najpierw, co jest tu miarą danych i algorytmu. Rozmiar danych to po prostu  $n$ , czyli liczebność zbioru, w którym poszukujemy naszego  $x$ . Niewątpliwie, gdy elementu  $x$  nie ma w tablicy, to druga wersja jest gorsza, po prostu będzie niepotrzebnie sprawdzała jeden warunek więcej, a i tak zakończy działanie, gdy długość przedziału spadnie do 1. Jeśli jednak  $x$  jest, i to w dodatku tylko w jednym miejscu, to być może, mając szczęście, nie wykonamy ani jednego obrotu pętli i od razu natrafimy na szukaną wartość. Ale tak zdarzy się tylko w jednym przypadku na  $n$ , przy założeniu, że każdej wartości

szukamy z jednakowym prawdopodobieństwem. Jeśli  $x$ -ów jest więcej, to oczywiście szansa natrafienia na którąś z pozycji zawierającą  $x$  rośnie i drugi algorytm może okazać się lepszy (a na pewno lepszy będzie, gdy wszystkie wartości są równe  $x$ ).

W każdym razie złożoność algorytmu jest zależna od liczby wykonań pętli. Przyjmijmy zatem jako miarę złożoności liczbę sprawdzeń dozoru pętli. Przeanalizujemy sytuację, w której zadany ciąg w tablicy jest ściśle rosnący, a poszukiwane wartości pojawiają się z jednakowym prawdopodobieństwem. Czy warto wtedy dokładać sobie pracy przy każdorazowym sprawdzaniu warunku wyjścia z pętli, aby zmniejszyć liczbę jej obrotów? Rachunki w ogólnym przypadku są dość skomplikowane. Wykonamy teraz upraszczające założenie. Przyjmiemy mianowicie, że  $n = 2^k - 1$  dla pewnego  $k$ . Dzięki temu po każdym obrocie pętli w drugim algorytmie będziemy mieli przedział długości  $2^{k'} - 1$  dla pewnego  $k' \leq k$ . Czy wolno robić takie założenia? W zasadzie mogłyby one nieco wypaczyć wynik, ale w naszym przypadku nie będzie tak źle. Jak duży błąd możemy popełnić? Otóż możemy pomylić się nie więcej niż o 1. W końcu po pierwszym strzale mamy do zbadania dla dowolnego  $n$  przedział nie dłuższy niż  $\frac{n}{2}$ , zatem nasz wynik będzie się różnił od prawdziwego nie więcej niż o 1, bo w przedziale  $[\frac{n}{2}..n]$  musi znajdować się jakaś liczba postaci  $2^k - 1$ , a dla niej liczba kroków jest co najmniej taka jak dla  $\frac{n}{2}$  – wszak liczba obrotów pętli nie może urosnąć dla mniejszych danych. I można przypuszczać, że różnica w obliczeniach dla obu algorytmów będzie proporcjonalna, zatem obliczenia wykonane dla  $n$  będących prawie potęgami dwójki będą dawały reprezentatywne wyniki.

Jaki jest zatem średni koszt pierwszego algorytmu? Tam – zależnie od szukanej wartości  $x$  – otrzymamy zawsze albo  $k$ , albo  $k + 1$  sprawdzeń dozoru pętli. Mniej się nie da: musimy zjechać z długością przedziału do jedynki, odrzucając za każdym razem albo połowę, albo o jeden mniej niż połowę elementów. W przypadku drugiego algorytmu sytuacja nieco się komplikuje. Żeby wyznaczyć średnią liczbę sprawdzeń dozoru pętli, przyjmijmy, że kolejno wyszukujemy wartości  $1, 2, \dots, n$  w tablicy zawierającej tylko te liczby, dodajmy uzyskane liczby sprawdzeń dozoru pętli dla każdej z tych wartości i podzielmy wynik przez  $n$  – otrzymamy w ten sposób średnią liczbę sprawdzeń dozoru pętli.

Jest tylko jedna wartość, dla której pętla nie wykona się ani razu, czyli dozór będzie sprawdzony raz – znajduje się ona w połowie tablicy. Są dwie wartości, dla których sprawdzenie wykona się dwa razy – są one umieszczone mniej więcej w jednej czwartej i w trzech czwartych tablicy, cztery, dla których wykona się trzy razy, itd. W końcu dla prawie połowy wartości (a dokładniej dla wszystkich nieparzystych wartości) wykonamy maksymalną liczbę  $k$  sprawdzeń dozoru. Łącznie zatem wysumowana liczba wszystkich sprawdzeń dozoru pętli dla wszystkich danych to

$$S = \sum_{j=1}^k j2^{j-1}$$

(sumowanie przebiega względem liczby sprawdzeń dozoru pętli).

Ile wynosi ta liczba dokładnie, można obliczyć na wiele sposobów. Jednym z najciekawszych jest niejako utrudnienie tego problemu i obliczenie wartości wielomianu

$$W(x) = \sum_{j=1}^k jx^{j-1}$$

w punkcie  $x = 2$ . Widzimy, że

$$W(x) = V'(x) \quad \text{dla} \quad V(x) = \sum_{j=1}^k x^j.$$

Ale wyrazy wielomianu  $V(x)$  tworzą zwykły ciąg geometryczny z pierwszym wyrazem, jak i ilorazem, równym  $x$ . Korzystając ze wzoru na sumę pierwszych  $k$  wyrazów ciągu geometrycznego, otrzymujemy

$$V(x) = x \frac{x^k - 1}{x - 1}.$$

A pochodna to

$$V'(x) = \frac{((k+1)x^k - 1)(x-1) - x^{k+1} + x}{(x-1)^2}.$$

Wygląda dość obskurnie, ale takie jest już życie! Na szczęście chcemy znać jej wartość w dość przyjaznym punkcie  $x = 2$ , więc szybko otrzymujemy

$$S = W(2) = V'(2) = 2^k(k-1) + 1.$$

Ostatecznie średnia liczba obrotów pętli będzie równa

$$\frac{S}{n} = \frac{2^k(k-1) + 1}{2^k - 1} > k - 1.$$

Zatem okazuje się, że w porównaniu z pierwszym algorytmem zyskujemy średnio nie więcej niż dwa obroty pętli!

Czy zysk ten może zrekompensować stratę związaną ze sprawdzaniem dodatkowego warunku przy każdym obrocie pętli? Tak, ale tylko dla małych danych. Przeprowadzone eksperymenty pokazały, że dla danych rzędu paru milionów pierwszy, a zarazem prostszy, algorytm daje wyniki o kilka-kilkanaście procent lepsze. W trakcie testowania nie korzystaliśmy tu z funkcji optymalizacyjnych kompilatora, takich jak tzw. leniwe wyliczanie warunków logicznych, co mogłoby nieco poprawić działanie drugiego algorytmu.

Leniwe wyliczanie polega na tym, że kompilator przerywa obliczanie warunku w pierwszym momencie, w którym orientuje się, że wynik już i tak zna. Na przykład, gdy oblicza koniunkcję dwóch warunków i pierwszy z nich okazuje się fałszem, uznaje, że nie ma po co obliczać drugiego, i przerywa obliczenia, od razu przekazując wynik.

Zanim zaczniemy ulepszać nasze rozwiązania, warto czasem trochę przeliczyć, czy się to w ogóle opłaca. A tak czy siak, warto potem przetestować nowe pomysły z zegarkiem w rękę, szczególnie w przypadku algorytmów ważnych, które mają istotny wpływ na końcową złożoność. Reasumując: dla typowych danych, kiedy z dużym prawdopodobieństwem dany element  $x$  występuje w tablicy co najwyżej jednokrotnie, zazwyczaj lepiej stosować pierwszy, prostszy algorytm i nie zawracać sobie głowy wcześniejszym przerywaniem pętli. Czasami w programowaniu leniwość się opłaca!