O co chodzi w złożoności czasowej

Jakub RADOSZEWSKI

Jeśli mam przybliżyć komuś pojęcie złożoności czasowej, zazwyczaj opowiadam mu następujący problem. Dany jest ciąg liczb całkowitych $a_1, \ldots, a_n$, a naszym celem jest znaleźć jego fragment – czyli spójny podciąg – którego suma elementów jest jak największa. Dokładniej, należy napisać program, który wczytuje ciąg liczb całkowitych i wyznacza sumę elementów w takim maksymalnym fragmencie. Zadanie to można zinterpretować, na przykład, tak: dla określonego ciągu transakcji na koncie (wpłaty, wypłaty) chcemy wyznaczyć przedział czasu, w którym bilans transakcji był możliwie najkorzystniejszy.

Przykładowo, maksymalnym fragmentem ciągu 2 −6 4 −2 3 4 −1 −3 jest 4 −2 3 4 o sumie 9

Nasz ciąg najwygodniej reprezentować w tablicy. Pierwsze przychodzące na myśl rozwiązanie może polegać na przejrzeniu wszystkich fragmentów ciągu i sprawdzeniu, który z nich ma maksymalną sumę. W języku Pascal taka funkcja może wyglądać następująco:

```
function max_fragment(var a : array of LongInt;
                          n : LongInt) : LongInt;
var i, j, k, wyn, suma : LongInt;
begin
  wyn := 0;
  for i := 1 to n do
    for j := i to n do begin
      {rozważamy fragment a[i],...,a[j]}
      suma := 0;
      for k := i to j do
        suma := suma + a[k];
      if suma > wyn then
        wyn := suma;
    end;
  max_fragment := wyn;
end;
```

Czy to jest dobre rozwiązanie? Można je chwilę potestować i stwierdzić, że dla kilku przykładów daje poprawne wyniki. Ale dobre to nie tylko znaczy poprawne. Czy ten program jest szybki? Sprawdźmy!

Na potrzeby tego zadania wygenerowaliśmy trzy testy zawierające ciągi o długościach 100, 10 000 i 1 000 000. Gdy uruchamiamy na tych testach program oparty na powyższej funkcji, to na pierwszym z nich działa w ułamku sekundy, ale dla drugiego i trzeciego trudno doczekać się końca jego działania.

Nie trzeba było jednak uruchamiać tego programu, żeby domyślić się, że tak właśnie będzie. Spróbujmy ustalić, ile operacji wykonuje ten program dla danych wejściowych rozmiaru n. W programie występuje cała gama różnych operacji – przypisania, operacje arytmetyczne, warunki, pętle... – i trudno byłoby to tak dokładnie policzyć. Warto więc ustalić, która operacja jest *operacją dominującą*, czyli którą operację wykonujemy najczęściej – i tę operację zliczać. Łatwo zauważyć, że w tym przypadku będzie to zwiększanie wartości zmiennej `suma`. Ile razy ma to miejsce? Dla każdego fragmentu $a_i, \ldots, a_j$ wykonujemy $j - i + 1$ takich operacji, a zatem łącznie będzie ich:

$$\sum_{i=1}^{n} \sum_{j=i}^{n} (j - i + 1).$$

Po pracowitym przeliczeniu tej sumy, którego oglądania darujemy Czytelnikowi, wychodzi:

$$\tfrac{1}{6}n^3 + \tfrac{1}{2}n^2 + \tfrac{1}{3}n.$$

Nietrudno teraz sprawdzić, że dla $n = 100, 10\,000, 1\,000\,000$ otrzymujemy, odpowiednio, 171 700, mniej więcej $1{,}7 \cdot 10^{11}$ i mniej więcej $1{,}7 \cdot 10^{17}$ operacji. Biorąc pod uwagę, że obecnie komputery mogą wykonać maksymalnie 10^9 bardzo prostych operacji na sekundę, widzimy, dlaczego nasz program działa tak wolno.

W takim razie warto zastanowić się nad tym, czy nie ma szybszego rozwiązania. Najbardziej znaczący składnik w powyższym wzorze to, oczywiście, składnik z n^3

Rozwiązanie zadania M 1499.
Tak!
Niech k będzie taką liczbą naturalną, że $10^k > 2016^{2016}$. Rozważmy liczby $1234567890 \cdot 10^k + l$ dla $l = 1, 2, \ldots, 2016^{2016}$. Zapis każdej z nich rozpoczyna się od 1234567890, więc zawiera wszystkie dziesięć cyfr. Jednocześnie jest to 2016^{2016} kolejnych liczb naturalnych, więc jedna z nich jest wielokrotnością liczby 2016^{2016}.

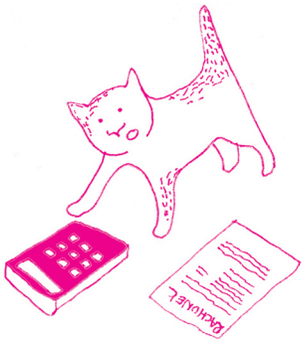
i to on w decydującym stopniu wpływa na to, że nasz program działa tak wolno. Z tego względu w analizie liczby operacji wykonywanej przez program (czy raczej algorytm) pomija się składniki *niższego rzędu*, co w przypadku wielomianów oznacza, że interesuje nas tylko jednomian o najwyższym wykładniku. Dalszym uproszczeniem jest pominięcie, zazwyczaj niedużych, stałych czynników.
W ten sposób uzyskujemy funkcję, która z niezłym przybliżeniem określa czas działania wynikowego programu. Powiemy więc, że nasz algorytm ma *złożoność czasową* $O(n^3)$. Przykładowo, dla trzech podanych wartości n funkcja n^3 przyjmuje wartości 10^6, 10^{12} i 10^{18}, co nieźle przybliża dokładnie obliczone liczby operacji. W dalszej części artykułu spróbujemy przekonać się, że pojęcie złożoności czasowej pomaga w projektowaniu szybkich programów.

Złożoność czasową nazywa się także *złożonością obliczeniową* lub po prostu *złożonością*. Formalnie, notacja $O(f(n))$ oznacza, że istnieje taka stała M, że dla każdego n algorytm wykonuje co najwyżej $Mf(n)$ operacji.

Intuicyjnie złożoność $O(n^3)$ wzięła się w powyższym programie stąd, że rozpatrujemy wszystkie fragmenty ciągu, których jest z grubsza n^2, i obliczamy sumę każdego z nich, co wymaga maksymalnie n dodawań – łącznie, mniej więcej, n^3 dodawań. Gdyby dało się szybciej obliczać sumę fragmentu, udałoby nam się wykonywać jedynie mniej więcej n^2 operacji...

Okazuje się, że jest to możliwe. W rozwiązaniu przydaje się pomysł z zakresu księgowości: aby łatwo sprawdzać, na ile dobry był dany okres na koncie, wystarczy po każdej operacji przechowywać łączny *bilans* z wszystkich dotychczas wykonanych operacji. W ten sposób do określenia sumy danego okresu wystarczy nam znajomość bilansu na koniec tego okresu oraz bilansu tuż przed jego rozpoczęciem. Kod odpowiedniej funkcji znajduje się poniżej.

Takie bilanse dla ciągu nazywa się w algorytmice *sumami częściowymi* lub *prefiksowymi* ciągu.



```
function max_fragment2(var a : array of LongInt;
                           n : LongInt) : LongInt;
var i, j, wyn, suma : LongInt;
  bilans : array[0 .. MAX_N] of LongInt;
begin
  bilans[0] := a[0];
  for i := 1 to n do
    bilans[i] := bilans[i - 1] + a[i];
  wyn := 0;
  for i := 1 to n do
    for j := i to n do begin
      suma := bilans[j] - bilans[i - 1];
      if suma > wyn then
        wyn := suma;
    end;
  max_fragment2 := wyn;
end;
```

Zbadajmy, na ile szybkie jest to rozwiązanie. Uruchomienie go na naszych trzech testach wykazuje znaczącą poprawę: dla pierwszych dwóch testów wyniki otrzymujemy prawie natychmiast, choć dla trzeciego znów nie udaje się doczekać na odpowiedź. A co wykazuje analiza „teoretyczna”? Operacją dominującą będzie teraz dowolna z operacji wykonywanych wewnątrz dwóch zagnieżdżonych pętli. Każda z nich wykonywana jest tyle razy, ile jest możliwych wyborów indeksów $1 \leqslant i \leqslant j \leqslant n$, czyli

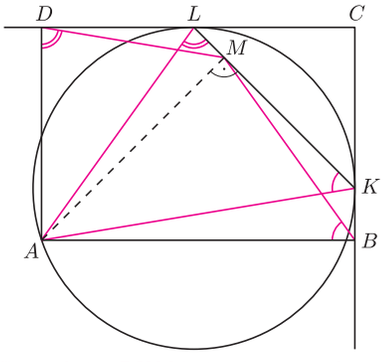
$$\binom{n}{2} = \tfrac{1}{2}n^2 + \tfrac{1}{2}n$$

razy, co odpowiada złożoności czasowej $O(n^2)$. Obliczenia wstępne polegają na wykonaniu zaledwie $n-1$ dodawań, więc można je pominąć w opisie złożoności. Dla $n = 10^2, 10^4, 10^6$ mamy zatem mniej więcej 10^4, 10^8 i 10^{12} operacji.

Nie moglibyśmy jednak tego wszystkiego pozostawić, nie przedstawiając rozwiązania, które poradzi sobie z naszym największym testem. Znów, intuicyjnie, złożoność czasowa $O(n^2)$ powyższego rozwiązania bierze się stąd, że rozpatrujemy wszystkie początki i końce fragmentów, a każdych z nich jest n. Postawmy śmiałe pytanie: czy dałoby się zamiast tego rozważać np. tylko początki albo tylko końce fragmentów?... Okazuje się, że tak!

W tym celu wystarczy zapytać, jak dla ustalonego końca fragmentu dobrać początek fragmentu, tak aby suma fragmentu była maksymalna. Będzie to,

Rozwiązanie zadania M 1500.
Prosta AM jest prostopadła do KL, więc jest nachylona do boków prostokąta pod kątem 45°. Ponadto kąt środkowy oparty na cięciwie KL jest prosty, a stąd $\sphericalangle KAL = \sphericalangle 45^\circ = \sphericalangle BAM$.



Czworokąt $ABKM$ jest opisany na okręgu o średnicy AK. W takim razie kąty ABM i AKM są równe.
Otrzymujemy, że trójkąty ABM i AKL są podobne. Analogicznie trójkąty AMD i AKL są podobne. W takim razie mamy podobieństwo trójkątów ABM i AMD, a stąd

$$\frac{AB}{AM} = \frac{AM}{AD}.$$

Bezpośrednio stąd wynika, że pole prostokąta $ABCD$ jest równe $AB \cdot AD = AM^2 = 1$.

oczywiście, taki indeks operacji, przed wykonaniem której bilans na koncie jest możliwie *najmniejszy*!

```
function max_fragment3(var a : array of LongInt;
                           n : LongInt) : LongInt;
var i, j, wyn, suma, min_bilans : LongInt;
  bilans : array[0 .. MAX_N] of LongInt;
begin
  bilans[0] := a[0];
  for i := 1 to n do
    bilans[i] := bilans[i - 1] + a[i];
  wyn := 0;
  min_bilans := 0;
  for j := 1 to n do begin
    suma := bilans[j] - min_bilans;
    if suma > wyn then
      wyn := suma;
    if bilans[j] < min_bilans then
      min_bilans := bilans[j];
  end;
  max_fragment3 := wyn;
end;
```

Tym razem główna pętla wykonuje jedynie kilka prostych operacji, więc standardowo z pominięciem stałej określamy złożoność czasową ostatecznego rozwiązania jako $O(n)$. I rzeczywiście, jak można się już domyślić, to rozwiązanie bez problemu radzi sobie z wszystkimi trzema testami.

W tym artykule spróbowaliśmy w kilku słowach opowiedzieć, na czym polega analiza złożoności czasowej algorytmów. Celowo pominęliśmy kwestie takie jak dobór odpowiednich typów danych (czy typ `LongInt` wystarcza?) oraz *złożoność pamięciową*, czyli – znów przybliżone – określenie zużycia pamięci przez program.

Programy i testy opisane w artykule są dostępne na stronie `deltami.edu.pl`

Hexapawn, czyli czego można nauczyć pudełka

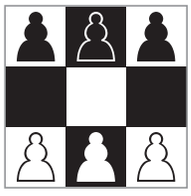
Kamila ŁYCZEK

Zamiast analizować, czy gra jest sprawiedliwa, zamiast szukać najlepszych strategii graczy, można stworzyć pewną maszynę, która część tej pracy wykona za nas. Trzeba jej objaśnić zasady, a potem z nią grać, niekoniecznie najlepiej – w końcu jeszcze nie przeanalizowaliśmy gry. Maszyna, grając, zapamiętując i wyciągając wnioski z przegranych oraz wygranych (co śmiało można zakwalifikować jako *uczenie się*), prędzej czy później zorientuje się, jak grać możliwie najlepiej, a więc ogrywać nas, o ile to tylko możliwe.

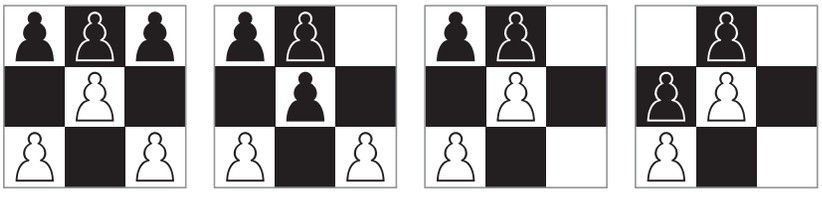
Maszyną górnolotnie będzie nazywany zestaw pudełek opatrzonych w etykietki, koraliki oraz stosowną instrukcję obsługi.

Uczenie się oznacza adaptacyjne zmiany w systemie, w tym sensie, że w miarę postępu procesu te same zadania wykonywane są przynajmniej tak samo dobrze jak we wcześniejszych etapach.

Instrukcja gry *Hexapawn* dla ludzkich graczy. Gra rozgrywa się na szachownicy 3×3. Początkowe ustawienie przedstawia rysunek 1. Dwaj gracze (pierwszy – biały, drugi – czarny) ruszają się na przemian. W każdym ruchu gracz rusza się jednym ze swoich pionków jedno pole do przodu lub, jeżeli ma taką możliwość, może (nie musi) wykonać bicie pionka przeciwnika jedno pole po skosie do przodu. Wygrywa ten gracz, który jako pierwszy dotrze swoim pionkiem na przeciwną stronę szachownicy (na jedno z pól, z których grę rozpoczynał przeciwnik) lub który uniemożliwi jakikolwiek ruch przeciwnikowi (zbije wszystkie jego pionki bądź go zablokuje).



Rys. 1. Początkowe ustawienie w grze *Hexapawn*.



Rys. 2. Przykładowa rozgrywka zakończona wygraną gracza czarnego.