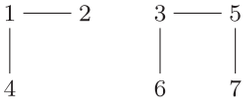
Jak znaleźć *second min*?

Jakub RADOSZEWSKI

Jednym z pierwszych zadań, z jakimi musi zmierzyć się każdy uczący się algorytmów lub programowania, jest znajdowanie minimum w tablicy (ciągu liczb). Oznaczmy taką tablicę przez $A[1..n]$. Poszukiwanie zaczynamy, naturalnie, od elementu $A[1]$, który staje się kandydatem na minimum. Jeśli teraz $A[1] \leqslant A[2]$, to $A[1]$ pozostaje kandydatem na minimum, a jeśli, przeciwnie, $A[1] > A[2]$, to kandydatem na minimum staje się $A[2]$. Łatwo zobaczyć, co będzie dalej: każdy kolejny element tablicy, $A[i]$, porównujemy z obecnym kandydatem na minimum i aktualizujemy tegoż kandydata, jeśli $A[i]$ okazał się od niego mniejszy (patrz pseudokod na marginesie).

```
min := A[1];
for i := 2 to n do
  if A[i] < min then
    min := A[i];
return min;
```

Łatwo oszacować, ile operacji wykonujemy w tym algorytmie: mamy dokładnie $n-1$ porównań oraz co najwyżej n przypisań. Co prawda, powyższy algorytm jest naprawdę prosty, ale zawsze można dla pewności spytać: czy dałoby się lepiej? Na przykład, czy można znaleźć minimum w tablicy, wykonując mniej niż $n-1$ porównań?

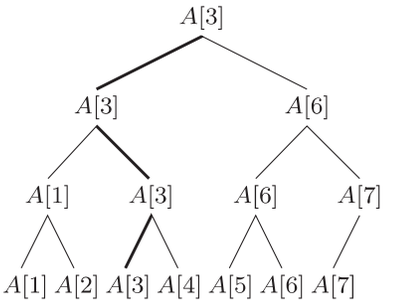


Rys. 1. Graf porównań odpowiadający tablicy $A[1..7]$ po wykonaniu porównań: $A[1]$ z $A[2]$, $A[1]$ z $A[4]$, $A[3]$ z $A[5]$, $A[3]$ z $A[6]$ i $A[5]$ z $A[7]$.

Intuicyjnie widać, że nie, czyli że trzeba wykonać co najmniej $n-1$ porównań. Tę intuicję można także wesprzeć formalnym uzasadnieniem. Ustalmy pewien algorytm wyznaczający minimum w tablicy $A[1..n]$. Dla danego przebiegu (wykonania) tego algorytmu możemy skonstruować *graf porównań*, którego wierzchołki $1, 2, \ldots, n$ reprezentują numery elementów w tablicy, a krawędź łączy dwa wierzchołki wtedy, gdy odpowiadające im elementy zostały porównane w tym przebiegu algorytmu (patrz rys. 1). Wystarczy teraz zauważyć, że jeśli na końcu takiego przebiegu graf porównań nie jest *spójny*, to rozważany algorytm nie może być poprawny: bez dodatkowych porównań nie jesteśmy w stanie stwierdzić, w której spójnej składowej faktyczne minimum się znajduje. Ponieważ graf spójny o n wierzchołkach musi mieć co najmniej $n-1$ krawędzi, więc rzeczywiście potrzebnych jest $n-1$ porównań.

Innym ciekawym uogólnieniem początkowego problemu byłoby wyznaczanie jednocześnie minimum i maksimum w tablicy. Okazuje się, że potrzeba i wystarczy do tego $\lceil 3n/2 \rceil - 2$ porównań. Jak mógłby wyglądać algorytm wykonujący taką liczbę porównań? I dlaczego nie da się lepiej?

Skoro wiemy już wszystko o wyznaczaniu minimum, to może teraz zastanowimy się, jak efektywnie znaleźć drugi co do wielkości element tablicy? Będzie to właśnie tytułowe *second min*.



Rys. 2. Turniej związany z tablicą $A[1..7]$. Zwycięzcą (czyli minimum) jest $A[3]$.

Najprościej wykorzystać poprzedni algorytm: znajdujemy minimum, usuwamy je z tablicy (np. odpowiednio je oznaczając), po czym znajdujemy minimum wśród pozostałych elementów tablicy. W ten sposób wykonujemy $n-1+n-2 = 2n-3$ porównania. Można to jednak zrobić lepiej, jeśli tylko wykorzystamy inny algorytm wyszukiwania minimum.

Pokażemy, jak znaleźć second min za pomocą $n + \lceil \log_2 n \rceil - 2$ porównań. Zaczynamy od zbudowania *turnieju*, czyli „drabinki meczów" między elementami tablicy A: w każdym „meczu" mniejszy element wygrywa i w ten sposób wyłaniany jest zwycięzca, czyli minimum. Gdyby n było potęgą dwójki, ów turniej moglibyśmy reprezentować jako pełne drzewo binarne; w ogólnym przypadku na ostatnim poziomie drzewa mogą występować luki (patrz rys. 2).

Do wyznaczenia second min nie zawsze potrzeba $n + \lceil \log_2 n \rceil - 2$ porównań – a to dlatego, że algorytm może mieć szczęście. Czasem może wystarczyć $n-1$ porównań – na przykład wtedy, gdy przy wyznaczaniu min za pomocą pierwszego opisanego algorytmu okaże się, że $A[1]$ jest minimum z elementów $A[1], \ldots, A[n-1]$, po czym $A[1]$ przegra w ostatnim porównaniu z elementem $A[n]$. Wtedy od razu wiemy, że $A[1]$ jest elementem second min. Jest to jednak wyjątkowo optymistyczny wariant; gdyby okazało się chociażby, że $A[1] < A[n]$, to po tym porównaniu o second min nie wiedzielibyśmy nic.

Widzimy, że za pomocą turnieju znów udaje nam się wyznaczyć minimum w tablicy za pomocą $n-1$ porównań (meczów). Rzeczywiście, po każdym porównaniu jeden element odpada z turnieju. A gdzie w drabince turniejowej znajduje się element second min? Otóż mógł on przegrać w turnieju jedynie z elementem min – no bo z nikim innym. Na każdym poziomie drabinki jest co najwyżej jeden element, który przegrał z minimum, czyli łącznie mamy co najwyżej $\lceil \log_2 n \rceil$ kandydatów na second min. Przykładowo, w turnieju z rysunku 2 kandydatami na second min są $A[4]$, $A[1]$ i $A[6]$. Wystarczy zatem wybrać minimum spośród tych właśnie elementów, do czego, jak wiemy, potrzeba już tylko $\lceil \log_2 n \rceil - 1$ porównań. Łącznie wykonaliśmy $n + \lceil \log_2 n \rceil - 2$ porównań, czyli tyle, ile chcieliśmy.

Naturalnie nasuwa się pytanie, czy da się lepiej, tzn. czy aby na pewno przy wyznaczaniu second min trzeba wykonać $n + \lceil \log_2 n \rceil - 2$ porównań. W związku z uwagą na marginesie warto tu zadać bardziej precyzyjne pytanie: czy każdy algorytm znajdujący second min wykonuje *w pesymistycznym przypadku* co najmniej $n + \lceil \log_2 n \rceil - 2$ porównań? Innymi słowy, czy każdemu takiemu algorytmowi

możemy tak złośliwie dobrać dane wejściowe, żeby musiał wykonać podaną liczbę porównań? Odpowiedź na to pytanie jest twierdząca, co spróbujemy uzasadnić. Odtąd założymy dla uproszczenia, że wszystkie elementy w tablicy A są różne; poniższe rozumowanie można lekko zmodyfikować tak, aby działało także w przypadku powtarzających się elementów w A.

Przede wszystkim musimy wyjaśnić jedną rzecz: każdy algorytm wyznaczający second min musi przy okazji znaleźć element min, tzn. na końcu jego działania musi być jednoznacznie wyznaczone, który element tablicy jest minimalny. Faktycznie, gdyby na końcu algorytmu wciąż byli dwaj kandydaci na minimum (czyli mniejsi od wszystkich elementów, z którymi byli porównywani), to któryś z nich mógłby równie dobrze być elementem second min.

Aby dokończyć rozumowanie, musimy pokazać pewną kluczową i trochę nieintuicyjną własność. Otóż w każdym algorytmie wyznaczającym minimum w tablicy (a więc także w każdym wyznaczającym second min) element min jest porównywany w pesymistycznym przypadku co najmniej $\lceil \log_2 n \rceil$ razy.

Zanim to pokażemy, zastanówmy się, co nam to da. Przyjrzyjmy się grafowi porównań algorytmu wyznaczającego second min. Wiemy – a dokładniej, będziemy wiedzieli, jak udowodnimy – że w pesymistycznym przypadku z elementu min wychodzi w tym grafie co najmniej $\lceil \log_2 n \rceil$ krawędzi. Usuńmy z grafu element min; reszta musi pozostać spójna, gdyż w przeciwnym przypadku mielibyśmy w niej więcej niż jednego kandydata na second min. Stąd, reszta grafu musi zawierać co najmniej $n-2$ krawędzie, czyli łącznie musi być co najmniej tyle krawędzi, ile należało pokazać.

Pozostał nam już tylko dowód tajemniczej własności wszystkich algorytmów wyznaczających minimum. Teraz całą sytuację przedstawimy jako grę...

Mamy dwoje graczy. Pierwszy gracz będzie symulował działanie algorytmu wyszukującego minimum, czyli będzie zadawał pytania postaci: „Czy $A[i] < A[j]$?”. Zawartość tablicy A nie jest jednak znana *a priori*. Drugi gracz odpowiada na pytania pierwszego, przy czym jego odpowiedzi nie mogą nigdy być sprzeczne – to znaczy, że po każdej sekwencji jego odpowiedzi musi istnieć zawartość tablicy $A[1..n]$ zgodna ze wszystkimi odpowiedziami, jakich udzielił. Celem pierwszego gracza jest wyznaczyć minimum w tablicy, wykonując mniej niż $\lceil \log_2 n \rceil$ porównań dotyczących min, a celem drugiego – spowodować, żeby pierwszy gracz musiał zadać co najmniej $\lceil \log_2 n \rceil$ pytań dotyczących min.

Podamy teraz strategię wygrywającą dla drugiego gracza. Z każdym elementem tablicy $A[i]$ związemy licznik $l[i]$; początkowo wszystkie liczniki są ustawione na 1. Będziemy dbali o to, aby wszystkie liczniki były zawsze nieujemne i całkowite oraz aby ich suma była równa n. Będziemy też utrzymywać niezmiennik, że wszystkie dodatnie liczniki odpowiadają kandydatom na minimum, czyli elementom, które w żadnym porównaniu nie okazały się większe.

Metoda odpowiadania na pytania jest następująca: jeśli pierwszy gracz chce porównać elementy $A[i]$ i $A[j]$, to odpowiadamy, że mniejszy jest ten element, który ma nie mniejszy licznik. Wówczas licznik mniejszego elementu zwiększamy o wartość licznika większego elementu, a drugi z tych liczników zerujemy (patrz przykład na rysunku 3). Jeśli tylko przed wykonaniem zapytania co najmniej jeden z liczników $l[i]$, $l[j]$ był niezerowy, to postępując zgodnie z tą metodą, na pewno nigdy nie udzielimy odpowiedzi sprzecznej z poprzednimi, gdyż porównanie wygra kandydat na minimum. A jeżeli oba liczniki były zerowe, to musimy po prostu odpowiedzieć tak, żeby nie skłamać: jeśli na podstawie dotychczasowych odpowiedzi można wywnioskować, który z elementów $A[i]$, $A[j]$ jest mniejszy, to odpowiadamy zgodnie z prawdą, a jeśli nie, to odpowiadamy jakkolwiek.

Łatwo sprawdzić, że podana metoda spełnia wszystkie żądane niezmienniki. Na końcu gry element minimalny musi mieć licznik równy n, a pozostałe liczniki muszą być zerowe. A ponieważ po każdym pytaniu licznik elementu minimalnego może się co najwyżej podwoić, więc łącznie wykonaliśmy na tym liczniku co najmniej $\lceil \log_2 n \rceil$ operacji. To kończy dowód!

1 1 1 1 1 1 1	$A[3] < A[5]$
1 1 2 1 0 1 1	$A[6] < A[7]$
1 1 2 1 0 2 0	$A[2] < A[4]$
1 2 2 0 0 2 0	$A[2] < A[5]$
1 2 2 0 0 2 0	$A[6] < A[1]$
0 2 2 0 0 3 0	$A[3] < A[2]$
0 0 4 0 0 3 0	$A[5] < A[4]$
0 0 4 0 0 3 0	$A[3] < A[6]$
0 0 7 0 0 0 0	

Rys. 3. Przykładowa rozgrywka w 7-elementowej „grze w minimum”: w lewej kolumnie są liczniki $l[1..7]$, a w prawej porównywane elementy (pytania pierwszego gracza) i wyniki porównań (odpowiedzi drugiego gracza). Minimum okazał się element $A[3]$. Wziął on udział w trzech porównaniach.

Intuicyjne znaczenie liczników $l[i]$ jest takie: gdybyśmy za każdym razem porównywali tylko liczby będące kandydatami na minimum, to wartość $l[i]$ oznaczałaby liczbę elementów, o których wiemy, że są większe od danego kandydata.

Polecamy także artykuł Marcina Peczarskiego „Sortowanie z minimalną liczbą porównań” z *Delty* 11/2004.