MIM, jesteś członkiem samorządu studenckiego. Występujesz na uczelni w określonych **rolach**, a każda taka rola daje Ci konkretne uprawnienia. Jako student możesz zabierać głos na forum dyskusyjnym w USOSowni i wypożyczać książki w systemie bibliotecznym, jako student MIM masz dostęp do portali z licencjonowanym oprogramowaniem, jako członek samorządu studenckiego możesz wysyłać maile do studentów MIM. Widać, że rola decyduje o Twoich **uprawnieniach**. Jak powstają role? Na szczęście w pełni automatycznie (to eliminuje ludzką pomyłkę) – jeśli tylko w USOS pojawia się informacja, że zostałeś przyjęty na studia, USOS buduje odpowiednią rolę i wysyła ją do repozytorium, z którego korzysta CAS.

Rola to nic innego jak **widok** (ang. *view*) w bazie danych, zbudowany automatycznie na podstawie danych przechowywanych w jednej lub kilku tabelach.

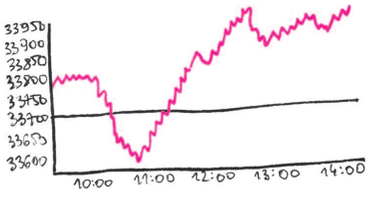
CAS dowiaduje się, że zacząłeś występować w roli studenta. Zadziała to także całkowicie automatycznie w odwrotnym przypadku – gdy skończysz studia i przestaniesz być studentem. Nagle – niestety – uniwersyteckie serwisy przestaną Ciebie rozpoznawać i serdecznie witać. Wtedy jednak będziesz już absolwentem i inne problemy będą zaprzątały Twoją głowę.

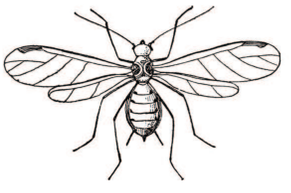
Jak widzisz, proces zarządzania tożsamością decyduje o Twoim istnieniu w wirtualnym świecie macierzystej uczelni. I – potencjalnie – wielu innych. Duża grupa uczelni europejskich, opierając się na wzajemnym zaufaniu, zbudowała federację na bazie swoich lokalnych systemów uwierzytelniania. Takie **federacyjne zarządzanie tożsamością** (ang. *federated identity management*) polega na tym, że gdy logujesz się w systemie informatycznym innej uczelni, jesteś przekierowywany do strony logowania CAS macierzystej uczelni (znowu znajomy orzełek) i jeśli Twój CAS Ciebie rozpozna, to logowanie się powiedzie. Na podobnej zasadzie działa **eduroam** (ang. ***edu**cation **roam**ing*), czyli sieć umożliwiająca bezpieczny roaming użytkowników jednostek naukowych oraz szkolnictwa wyższego. W praktyce oznacza to, że jeśli odpowiednio skonfigurujesz dostęp do eduroam z prywatnego laptopa czy komórki w swojej uczelni, to bez żadnych dodatkowych zmian w ustawieniach logowania będziesz się mógł podłączyć do sieci w pozostałych uczelniach z federacji. I surfować za darmo.

Dasz się poznać USOS-owi, a otworzą się przed Tobą drzwi nie tylko wirtualnego świata Twojej uczelni, ale także wiele innych. Oj, warto żyć z USOS-em w przyjaźni.

Informatyk gra na giełdzie

Tomasz IDZIASZEK





Nasz znajomy informatyk zdecydował się zainwestować część swoich oszczędności na giełdzie papierów wartościowych. Jak na informatyka przystało, do grania na giełdzie postanowił zaprząc komputer. W tym celu, korzystając z najnowszych trendów sztucznej inteligencji, napisał program, który na podstawie przeszłych notowań giełdowych przewiduje, jak kurs akcji będzie się zmieniał w przyszłości, i podejmuje decyzje o kupnie bądź sprzedaży. Nasz znajomy przetestował program, uruchamiając go na dużym zbiorze archiwalnych notowań. Zastanawia się teraz, jak dobrze jego program sobie poradził – stanął zatem przed problemem wyznaczenia najlepszej możliwej gry na giełdzie, jeśli znamy wszystkie notowania.

Model gry na giełdzie będzie następujący. Mamy daną tablicę $A[0..n]$ z notowaniami giełdowymi w kolejnych dniach: $A[i]$ oznacza cenę jednej akcji w i-tym dniu (dla uproszczenia przyjmiemy, że mamy tylko jeden rodzaj akcji). Na początku dysponujemy kwotą P i możemy wykonać nie więcej niż m operacji kupna-sprzedaży akcji (zakładamy, że możemy kupować ułamkową liczbę akcji). Zauważmy, że nie potrzebujemy wykonywać operacji równolegle (tzn. przed każdym kupnem opłaca się nam najpierw sprzedać wszystkie posiadane akcje). Ponadto warto też kupować akcje za całą dostępną kwotę. Z tego wynika, że jeśli pierwszą operację kupna przeprowadzimy w dniu k_1, a odpowiadającą jej sprzedaż w dniu s_1, to po tej operacji będziemy mieli kwotę $P \cdot A[s_1]/A[k_1]$. Naszym celem jest zmaksymalizowanie kwoty po wszystkich m operacjach, czyli iloczynu

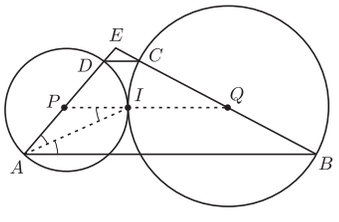
$$P \cdot \prod_{1 \leqslant i \leqslant m} \frac{A[s_i]}{A[k_i]}.$$

Możemy pozbyć się mnożeń i dzieleń, logarytmując powyższy wzór. Innymi słowy, równoważnie możemy zmaksymalizować sumę

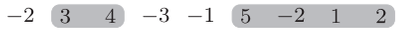
$$\log P + \sum_{1 \leqslant i \leqslant m} (\log A[s_i] - \log A[k_i]).$$

W końcu jeśli wprowadzimy pomocniczą tablicę $a[0..n-1]$, która zawierać będzie zmiany zlogarytmowanych notowań, tzn. $a[i] = \log A[i+1] - \log A[i]$, to

Rozwiązanie zadania M 1399.
Niech P i Q będą odpowiednio środkami ramion AD i BC.



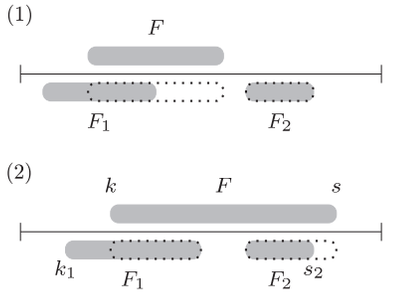
Połączmy je odcinkiem. Jest on równoległy do podstawy AB i zawiera punkt I. Dlatego $\sphericalangle BAI = \sphericalangle AIP = \sphericalangle PAI$, więc I leży na dwusiecznej kąta EAB. Analogicznie I leży na dwusiecznej kąta ABE, więc przechodzi przez niego także trzecia dwusieczna trójkąta ABE.



Rys. 1. Przykładowa tablica a dla $n = 9$. Optymalne rozwiązanie dla $m = 2$ to dwa zaznaczone fragmenty o sumach 7 i 6.

Przedział i fragment oznaczają w tym artykule to samo, używamy jednak dwóch nazw, by łatwo było wyrazić operację znajdowania przedziału o maksymalnej sumie, który zawiera się w pewnym ustalonym przedziale. Mówimy wtedy, że szukamy „maksymalnego fragmentu w zadanym przedziale".

Piszemy również, że znajdujemy maksymalny fragment, choć opisane algorytmy wyznaczają jedynie sumę liczb w tym fragmencie. Modyfikację algorytmów, by wyznaczały również końce fragmentu, pozostawiamy jako ćwiczenie dla Czytelników.



Rys. 2. Rozwiązania optymalne zawierające fragment F oraz fragmenty F_1 i F_2 (szare obszary), jak również nowe rozwiązanie optymalne dla dwóch fragmentów (przerywana linia).

nasze zadanie sprowadzi się do wybrania co najwyżej m rozłącznych fragmentów $[k_i, s_i)$, które maksymalizują sumę liczb do nich należących (rys. 1):

$$\sum_{1 \leqslant i \leqslant m} (a[k_i] + a[k_i + 1] + \ldots + a[s_i - 1]).$$

Dla $m = 1$ jest to klasyczne zadanie znajdowania fragmentu tablicy o największej sumie, które może być znane Czytelnikom. Z kolei jego uogólnienie dla $m \geqslant 2$ było treścią zadania pt. *Tanie linie*, które pojawiło się podczas weekendowej rundy Potyczek Algorytmicznych 2012. Oba problemy mają liczne ciekawe rozwiązania. W dalszej części artykułu przedstawimy, z pożytkiem dla znajomego informatyka, aż osiem z nich.

* * *

Na początek przypomnijmy, jak rozwiązać zadanie dla $m = 1$. Algorytm A1 jest bardzo prosty: w czasie $O(n^2)$ możemy przebadać wszystkie możliwe fragmenty, ustalając lewy koniec fragmentu i iterując po kolejnych możliwych prawych końcach.

Algorytm A2 wykona to samo w optymalnym czasie $O(n)$. Przeglądamy elementy tablicy od lewej do prawej i trzymamy naj – maksymalny fragment w przedziale $[0, i)$ oraz naj_P – maksymalny fragment, który dotyka prawego końca tego przedziału (tzn. kończy się elementem $a[i-1]$). Na początek naj i naj_P inicjujemy zerami, a następnie wykonujemy pętlę:

for $i := 1$ **to** n **do**
  $naj_P := \max(0, naj_P) + a[i-1]$;
  $naj := \max(naj, naj_P)$;

Zadanie dla $m \geqslant 2$ można rozwiązać, korzystając z metody programowania dynamicznego i uogólniając algorytm A2. Przez $naj[i, j]$ oznaczmy największą sumę liczb z przedziału $[0, i)$ zawartych w co najwyżej j rozłącznych fragmentach, a przez $naj_P[i, j]$ to samo, ale z zastrzeżeniem, że ostatni fragment zawiera element $a[i-1]$. Rekurencja (bez uwzględniania warunków brzegowych) jest następująca:

$$naj_P[i, j] = \max(naj[i-1, j-1], naj_P[i-1, j]) + a[i-1],$$
$$naj[i, j] = \max(naj[i-1, j], naj_P[i, j]).$$

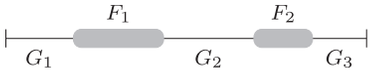
Rozwiązaniem jest $naj[n, m]$; algorytm A3 działa w czasie $O(nm)$.

Poszukując szybszego algorytmu dla $m \geqslant 2$, spróbujmy odpowiedzieć na pytanie: Czy, mając optymalne rozwiązanie dla $m-1$ fragmentów, da się je łatwo rozszerzyć do m fragmentów? Zatrzymajmy się nad przypadkiem $m = 2$. Powiedzmy, że fragment F ma największą sumę. Jak może wyglądać optymalne rozwiązanie dla dwóch fragmentów F_1, F_2? Rozważmy dwa przypadki (rys. 2).

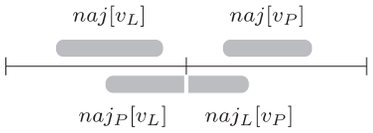
(1) Jeden z fragmentów (powiedzmy F_2) jest rozłączny z F. Wtedy $F \cup F_2$ jest poprawnym rozwiązaniem, zatem z optymalności $F_1 \cup F_2$ dostajemy, że $F \leqslant F_1$. Ale z optymalności F mamy $F \geqslant F_1$, zatem fragmenty F i F_1 mają tę samą sumę. To pokazuje, że $F \cup F_2$ jest również optymalne.
(2) Oba fragmenty $F_1 = [k_1, s_1)$, $F_2 = [k_2, s_2)$ przecinają $F = [k, s)$. Jeśli $k_1 < k$, to fragment $[k_1, k)$ musi mieć sumę zero (inaczej moglibyśmy poprawić F, dodając do niego ten fragment, lub poprawić F_1, usuwając ten fragment). Zatem usunięcie tego fragmentu z F_1 nie zmieni wyniku. Analogicznie dla $k_1 > k$ fragment $[k, k_1)$ musi mieć sumę zero i można go dodać do F_1. Stosując takie samo rozumowanie do prawego końca fragmentu F_2, dostajemy, że istnieje optymalne rozwiązanie $F_1 \cup F_2$, w którym $k = k_1$ i $s = s_2$.

Udowodniliśmy zatem, że możemy znaleźć optymalne rozwiązanie dla $m = 2$, rozszerzając fragment F o największej sumie. W tym celu: albo (1) dodajemy nowy fragment o największej sumie, który jest rozłączny z F, albo (2) znajdujemy fragment o *najmniejszej* sumie zawarty całkowicie w F i usuwamy go, dzieląc F na dwa fragmenty. Wybieramy ten wariant, który lepiej poprawia wynik.

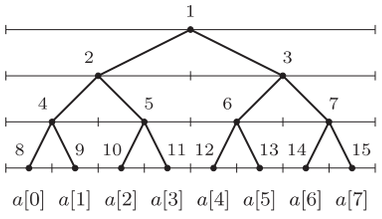
Zachęceni tym sukcesem moglibyśmy wykonać trochę eksperymentów praktycznych i przekonać się, że pomysł ten działa dla dowolnego m. Niech $F_1, F_2, \ldots, F_{m-1}$ to zbiór fragmentów o maksymalnej łącznej sumie,



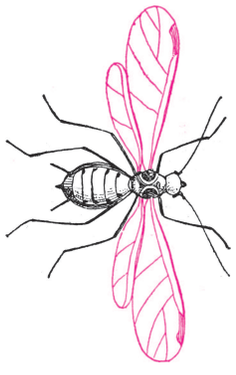
Rys. 3. Podział tablicy po wykonaniu dwóch faz algorytmu.



Rys. 4. Możliwe pozycje maksymalnego fragmentu $naj[v]$ w zależności od fragmentów w lewej i prawej połowie przedziału.



Rys. 5. Drzewo przedziałowe dla $n = 8$ ma 15 węzłów. Węzeł 5 odpowiada przedziałowi bazowemu $[2, 4)$. Przedział $[1, 6)$ można podzielić na przedziały bazowe, którym odpowiadają węzły 9, 5 i 6.



Na marginesie dodajmy, że istnieje też inne rozwiązanie $O(nm)$. W algorytmie A7 wykonujemy m faz. Każda faza polega na znalezieniu w tablicy jednego fragmentu o największej sumie (jak w algorytmie A2), dodaniu sumy do wyniku oraz na zamianie znaków wszystkich elementów w tym fragmencie na przeciwne. Kolejna faza odbywa się na uaktualnionej tablicy.

Czytelnicy zechcą udowodnić poprawność tego rozwiązania.
Wskazówka: po i fazach tablica jest podzielona na $2i + 1$ przedziałów, z których co drugi ma zamienione znaki. Wykazać, że istnieje maksymalny fragment całkowicie zawarty w jednym z tych przedziałów.

a $G_1, G_2, \ldots, G_m$ to pozostałe części tablicy (rys. 3). Aby uzyskać optymalne m fragmentów, albo dołączamy maksymalny fragment, który jest zawarty w pewnym G_i, albo usuwamy minimalny fragment z pewnego F_i. Zachęcamy do próby dowodu poprawności tego rozwiązania. To może nie być proste, ale dla Czytelników Wytrwałych na końcu artykułu podamy wskazówkę, jak można się do tego zabrać.

Pozostaje kwestia efektywnej implementacji tego pomysłu. Jedną fazę możemy wykonać w czasie $O(n)$, uruchamiając algorytm A2 na każdym przedziale osobno. Tak zapisany algorytm A4 będzie działał w czasie $O(nm)$, co nie daje nam jeszcze zysku w porównaniu z algorytmem A3. Widać, że kluczową operacją jest odpowiadanie na pytania „jaki jest maksymalny fragment w danym przedziale?" dla różnych przedziałów. Pokażemy teraz, jak to robić efektywnie.

W tym celu może nam pomóc jeszcze jeden algorytm dla $m = 1$. Algorytm A5 będzie oparty o metodę „dziel i zwyciężaj". Mając dany przedział o długości n, możemy go podzielić na dwie części o długości $n/2$. Maksymalny fragment w tym przedziale może znajdować się w całości w lewej części, w całości w prawej części lub może składać się z maksymalnego fragmentu, który dotyka prawej krawędzi lewej części, oraz maksymalnego fragmentu, który dotyka lewej krawędzi prawej części (por. też rys. 4).

Załóżmy, że $n = 2^N$, i zbudujmy drzewo przedziałowe (rys. 5). Drzewo będzie miało węzły o numerach od 1 do $2^{N+1} - 1$. W węźle $v = 2^l + i$, dla $0 \leqslant i < 2^l$, tego drzewa będą znajdować się informacje o przedziale $[i \cdot 2^{N-l}, (i + 1) \cdot 2^{N-l})$, a konkretnie: $sum[v]$ – suma liczb w przedziale, $naj[v]$ – maksymalny fragment w przedziale oraz $naj_L[v]$ i $naj_P[v]$ – maksymalne fragmenty dotykające odpowiednio lewego i prawego końca przedziału. Wartości w węzłach najniższego poziomu (tzn. dla $v \geqslant 2^N$) inicjujemy, przyjmując $sum[v] = a[v - 2^N]$ oraz $naj[v] = naj_L[v] = naj_P[v] = \max(0, a[v - 2^N])$. Wartości w pozostałych węzłach v wyznaczamy na podstawie wartości w węzłach $v_L = 2v$ i $v_P = 2v + 1$, które odpowiadają lewej i prawej połowie przedziału:

$$\begin{aligned} sum[v] &= sum[v_L] + sum[v_P], \\ naj[v] &= \max(naj[v_L], naj[v_P], naj_P[v_L] + naj_L[v_P]), \\ naj_L[v] &= \max(naj_L[v_L], sum[v_L] + naj_L[v_P]), \\ naj_P[v] &= \max(naj_P[v_L] + sum[v_P], naj_P[v_P]). \end{aligned}$$

Wyznaczenie wartości we wszystkich węzłach zabiera czas $O(n)$ i w takim czasie działa algorytm A5. Odpowiedzią jest, oczywiście, wartość $naj[1]$.

Skonstruowane drzewo przedziałowe wyróżnia się tym, że umożliwia ono znalezienie największego fragmentu dla dowolnego przedziału tablicy w czasie $O(\log n)$. W tym celu przypomnijmy, że każdy przedział można podzielić na $O(\log n)$ przedziałów bazowych, tzn. przedziałów, które odpowiadają węzłom drzewa przedziałowego (rys. 5). Niech $v_1, v_2, \ldots, v_B$ będą kolejnymi węzłami odpowiadającymi takiemu podziałowi. Wtedy maksymalny fragment to będzie albo $naj[v_i]$ dla pewnego i, albo

$$naj_P[v_i] + sum[v_{i+1}] + sum[v_{i+2}] + \ldots + sum[v_{j-1}] + naj_L[v_j]$$

dla pewnych $i < j$. Poniższa pętla wyznacza maksymalny fragment Naj w czasie $O(\log n)$:

for $i := 1$ **to** B **do**
  $Naj := \max(Naj, Naj_P + naj_L[v_i], naj[v_i])$;
  $Naj_P := \max(Naj_P + sum[v_i], naj_P[v_i])$;

Algorytm A6 jest następujący: najpierw budujemy drzewo przedziałowe (jak w algorytmie A5) oraz drugie drzewo przedziałowe, które będzie liczyło minimalne fragmenty. Oprócz wartości $naj[v]$ będziemy potrzebować również końców fragmentów – odpowiednie wzbogacenie drzewa przedziałowego zostawiamy jako ćwiczenie dla Czytelników. Wszystkie przedziały trzymamy w kolejce priorytetowej: przedziały $F_1, \ldots, F_{m-1}$ z priorytetami równymi minimalnym fragmentom w tych przedziałach, zaś przedziały $G_1, \ldots, G_m$ z priorytetami równymi maksymalnym fragmentom.

Każdy krok algorytmu to wyciągnięcie z kolejki przedziału o priorytecie o największej wartości bezwzględnej, uaktualnienie wyniku o wartość bezwzględną tego priorytetu, a następnie dodanie trzech nowych przedziałów do kolejki. Cały algorytm działa w czasie $O(n + m \log n)$.

* * *

Algorytm A6 jest efektywny, ale dość skomplikowany w implementacji. Przedstawimy teraz prostszy (choć nieco zaskakujący) algorytm, na który autor artykułu wpadł, próbując udowodnić poprawność algorytmów A4 i A6. Algorytm ten korzysta z metody „spróbujmy to zrobić od końca".

Wypiszmy liczby z tablicy w ciągu i podzielmy go na maksymalne bloki liczb o tym samym znaku (na potrzeby definicji bloku traktujemy 0 jako liczbę dodatnią). Zauważmy, że w optymalnym rozwiązaniu każdy fragment musi zaczynać się i kończyć pełnym blokiem, który zawiera liczby dodatnie (jeśli kończyłby się niepełnym blokiem dodatnim, to moglibyśmy ten fragment rozszerzyć, uzyskując nie gorsze rozwiązanie, a gdyby kończył się blokiem ujemnym – moglibyśmy go skrócić). Możemy więc zastąpić każdy blok przez jedną liczbę będącą sumą elementów tego bloku. Dodatkowo dodajmy na obu końcach ciągu liczbę-strażnika $-\infty$ (rys. 6).

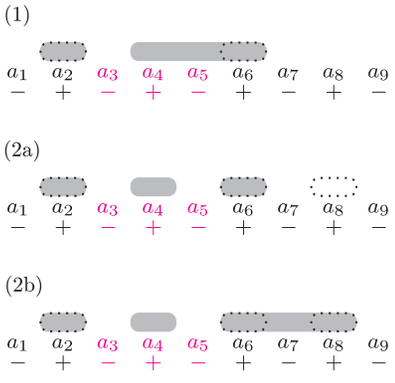
$-\infty \quad 7 \quad -4 \quad 5 \quad -2 \quad 3 \quad -\infty$

$-\infty \quad 7 \quad -4 \quad 6 \quad -\infty$

Rys. 6. Ciąg po podziale na bloki, powstały z tablicy liczb z rysunku 1, oraz ten sam ciąg po pierwszej fazie skracania.

Jeśli w nowym ciągu mamy co najwyżej m liczb dodatnich, to ich suma jest rozwiązaniem zadania. W przeciwnym przypadku będziemy iteracyjnie skracać ciąg, nie zmieniając optymalnego rozwiązania.

Faza skracania jest następująca: wybieramy liczbę w ciągu o najmniejszej wartości bezwzględnej, a_i, a następnie zastępujemy ją i dwie liczby z nią sąsiadujące – ich sumą. Zauważmy, że nowa liczba $a_{i-1} + a_i + a_{i+1}$ będzie miała przeciwny znak do a_i, więc krótszy ciąg nadal będzie zawierał naprzemiennie liczby dodatnie i ujemne. Fazę skracania można wykonać w czasie $O(\log n)$ – wystarczy trzymać elementy ciągu na liście, a poza tym mieć kolejkę priorytetową z liczbami uporządkowanymi względem ich wartości bezwzględnych. Zatem algorytm A8 będzie działał w czasie $O(n \log n)$.



Rys. 7. Konstrukcja optymalnego rozwiązania niezawierającego wyróżnionych trzech liczb (linia przerywana) na podstawie dowolnego rozwiązania optymalnego niezawierającego którejś z tych liczb (szare obszary) dla przypadku dodatniej środkowej liczby.

Pozostaje udowodnić poprawność operacji skracania. Powiedzmy, że ciąg składa się z 9 liczb oraz że liczbą o najmniejszej wartości bezwzględnej jest a_4 (dodatnia), więc chcemy skrócić ciąg, zastępując a_3, a_4, a_5 przez ich sumę (rys. 7). Chcemy wykazać, że istnieje optymalne rozwiązanie, w którym każdy fragment albo zawiera wszystkie liczby a_3, a_4, a_5, albo nie zawiera żadnej z nich. W tym celu pokażemy, jak z rozwiązania optymalnego niespełniającego tego warunku skonstruować rozwiązanie optymalne, które go spełnia. Rozważymy dwa przypadki:

(1) Rozwiązanie zawiera fragment, do którego należą dwie spośród liczb a_3, a_4, a_5. Wyrzucając z tego fragmentu obie te liczby, dostaniemy nie gorsze rozwiązanie (bo $a_4 \leqslant -a_3, -a_5$).
(2) Rozwiązanie zawiera jedną liczbę (czyli fragment a_4). Fragmentów jest mniej niż liczb dodatnich. Jeśli zatem jakaś z liczb dodatnich (powiedzmy a_8) nie należy do żadnego fragmentu, to zamieniając ją z a_4, dostaniemy nie gorsze rozwiązanie (bo $a_4 \leqslant a_8$). W przeciwnym przypadku co najmniej jeden fragment zawiera więcej niż jedną liczbę (powiedzmy a_6, a_7, a_8). Możemy zatem wyrzucić z niego jedną liczbę ujemną (a_7), rozbijając go na dwa fragmenty. Wyrzucając również fragment a_4, znowu dostaniemy nie gorsze rozwiązanie (bo $a_4 \leqslant -a_7$).

Dowód, gdy liczba o najmniejszej wartości bezwzględnej jest ujemna, jest symetryczny i zostawiamy go jako ćwiczenie dla Czytelników.

I wreszcie nadszedł czas na obiecaną wskazówkę do dowodu poprawności algorytmów A4 i A6. Wykonujmy kolejne fazy tych algorytmów, do momentu, aż wszystkie fragmenty będą zawierać liczby o tych samych znakach. Następnie wykonajmy na tak uzyskanym ciągu algorytm A8. Porównując podział tablicy po i-tej fazie algorytmu A4 i przed i-tą od końca fazą algorytmu A8, możemy dojść do wniosku, że w zasadzie te algorytmy działają tak samo, tylko w odwrotnej kolejności. Szkoda, że na giełdzie nie można najpierw sprzedać, a potem kupić...