Jacek Tomasiewicz
Treść zadania, Opracowanie

Maciej Matraszek
Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień pierwszy, 13.03.2013*

Bajtokomputer

Dany jest ciąg n liczb całkowitych $x_1, x_2, \ldots, x_n$ o wartościach ze zbioru $\{-1, 0, 1\}$. **Bajtokomputer** *to urządzenie, które umożliwia wykonywanie tylko jednego rodzaju operacji na tym ciągu: zwiększenia wartości x_{i+1} o wartość x_i, dla dowolnego $1 \leqslant i < n$. Liczby całkowite, jakie może pamiętać bajtokomputer, nie są ograniczone. W szczególności elementy przetwarzanego ciągu mogą przybierać dowolnie duże wartości.*

Zaprogramuj bajtokomputer, aby za pomocą minimalnej liczby operacji przekształcił dany ciąg w ciąg niemalejący, czyli taki, że $x_1 \leqslant x_2 \leqslant \ldots \leqslant x_n$.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 1\,000\,000$), oznaczającą liczbę elementów w danym ciągu. Drugi wiersz zawiera n liczb całkowitych $x_1, x_2, \ldots, x_n$ ($x_i \in \{-1, 0, 1\}$) stanowiących kolejne elementy danego ciągu, pooddzielane pojedynczymi odstępami.

W testach wartych łącznie 24% punktów zachodzi dodatkowy warunek $n \leqslant 500$, a w testach wartych łącznie 48% punktów zachodzi $n \leqslant 10\,000$.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, równą minimalnej liczbie operacji, które musi wykonać bajtokomputer, aby przekształcić dany ciąg w ciąg niemalejący, lub jedno słowo `BRAK`*, gdy otrzymanie takiego ciągu nie jest możliwe.*

Przykład

Dla danych wejściowych:
```
6
-1 1 0 -1 0 1
```

poprawnym wynikiem jest:
```
3
```

Wyjaśnienie do przykładu: *Za pomocą trzech operacji bajtokomputer może uzyskać ciąg $-1, -1, -1, -1,\ 0,\ 1$.*

Testy „ocen”:

`0ocen`: *$n = 6$, mały test z odpowiedzią* `BRAK`*;*

`1ocen`: *$n = 6$, mały test z odpowiedzią 4;*

`2ocen`: *$n = 500$, wszystkie elementy ciągu równe 1;*

`3ocen:` $n = 10\,000$, $x_1 = x_2 = \ldots = x_{9\,000} = -1$, $x_{9\,001} = \ldots = x_{9\,900} = 1$, $x_{9\,901} = \ldots = x_{10\,000} = 0$;

`4ocen:` $n = 1\,000\,000$, $x_1 = x_2 = \ldots = x_{999\,997} = -1$, $x_{999\,998} = 1$, $x_{999\,999} = 1$ i $x_{1\,000\,000} = -1$.

Rozwiązanie

Treść zadania można wyrazić całkiem zwięźle: mając dany ciąg $x_1, x_2, \ldots, x_n$ o wartościach w zbiorze $\{-1, 0, 1\}$, chcemy przekształcić go w ciąg niemalejący za pomocą minimalnej liczby operacji $x_{i+1} := x_{i+1} + x_i$. Dla jasności, wartości w początkowym ciągu będziemy oznaczać przez $\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n$.

Za chwilę udowodnimy, że w trakcie przekształcania ciągu opłaca się tworzyć tylko wartości $x_i \in \{-1, 0, 1\}$. Ponadto wszystkie operacje mogą być wykonywane od lewej do prawej.

Ścisły dowód nie jest prosty. Czytelnik niepotrzebujący dowodu może pominąć poniższy rozdział i od razu przejść do opisu implementacji.

Dowód

Oznaczmy przez o_i operację $x_i := x_i + x_{i-1}$. Niech O będzie pewną optymalną (tj. najkrótszą) sekwencją operacji, która przekształca początkowy ciąg wartości w ciąg niemalejący. Po wykonaniu operacji z O, końcowy ciąg można podzielić na trzy (być może puste) bloki:

- blok ujemny – pierwszy blok z lewej, zawierający liczby ujemne,
- blok zerowy – środkowy blok, zawierający same zera,
- blok dodatni – ostatni blok, zawierający liczby dodatnie.

Blok dodatni

Na początek przeanalizujemy, jak może wyglądać sekwencja operacji, która doprowadziła do powstania bloku dodatniego (o ile jest on niepusty). Załóżmy, że w sekwencji O operacja o_n jest wykonywana k razy. Możemy przekształcić sekwencję O tak, żeby te k operacji było wykonywanych w momencie, gdy x_{n-1} jest największe. Wtedy na końcu ostatni wyraz ciągu będzie nie mniejszy niż poprzednio.

Załóżmy teraz, że blok dodatni ma co najmniej dwa wyrazy i że któraś operacja o_{n-1} w sekwencji O zmniejsza wartość x_{n-1}. Zobaczmy, co się stanie, jeśli zamiast tej operacji dołożymy jedną operację typu o_n (tak jak poprzednio, gdy x_{n-1} jest największe).

Oznaczmy przez m_{n-1} maksymalną wartość x_{n-1} w tym zmodyfikowanym ciągu. Mamy więc teraz $k+1$ operacji typu o_n.

- Jeśli $k \geqslant 1$, to $m_{n-1} \geqslant 1$ i na końcu dostajemy:

$$x_n = (k+1) \cdot m_{n-1} + \hat{x}_n \geqslant 2 \cdot m_{n-1} - 1 \geqslant m_{n-1} \geqslant x_{n-1}.$$

- Jeśli $k = 0$, to wiemy, że $\hat{x}_n = 1$. Skoro operacja zmniejszająca wartość x_{n-1} była w którymś momencie potrzebna, to znaczy, że bez niej byłoby na końcu $x_{n-1} > 1$ (inaczej sekwencja O nie byłaby najkrótsza), czyli także $m_{n-1} > 1$. Zamieniając ją na operację o_n w najdogodniejszym momencie, dostajemy

$$x_n = 1 + m_{n-1} > x_{n-1}.$$

Wobec tego możemy przekształcić sekwencję O także tak, żeby nigdy nie zmniejszała x_{n-1} i żeby wykonywała operacje o_n tylko jak x_{n-1} jest maksymalne. Ale to z kolei oznacza, że można wykonać wszystkie operacje o_n po wszystkich operacjach o_{n-1}. Rozumując indukcyjnie, można dzięki temu pokazać, że na całym bloku dodatnim możemy wykonywać operacje od lewej do prawej, nie tracąc na optymalności.

Załóżmy, że na pierwszej pozycji j w bloku dodatnim mamy $\hat{x}_j < 1$. Wobec tego w którymś momencie x_{j-1} musiało być dodatnie (żeby x_j stało się dodatnie), a potem musiało stać się niedodatnie. Możemy przeprowadzić podobne rozumowanie jak wyżej i zamienić operacje tak, żeby po tym, jak x_{j-1} było dodatnie, już go nie zmniejszać, a w zamian za to zwiększać x_j i w konsekwencji rozszerzyć nasz blok. Stąd wynika, że jeśli blok dodatni zaczyna się wartością mniejszą niż 1, to można go zawsze rozszerzyć w lewo, nie powiększając długości ciągu operacji.

Zastanówmy się teraz, ile co najmniej operacji trzeba wykonać, żeby blok dodatni stał się niemalejący, przy założeniu, że operacje wykonujemy od lewej do prawej i tylko zwiększamy wartości wyrazów ciągu. Każdy wyraz $\hat{x}_i = 0$ musi być zwiększony co najmniej raz, bo zwiększamy go tylko o ostateczną wartość wyrazu x_{i-1}. Podobnie, każdy wyraz $\hat{x}_i = -1$ musi być zwiększony co najmniej dwa razy. Zauważmy, że możemy, idąc od lewej do prawej, każdy wyraz ciągu początkowo równy 0 zwiększyć raz, a każdy wyraz początkowo równy -1 dokładnie dwa razy. Dzięki temu zamienimy wszystkie wyrazy bloku na jedynki.

Ostatecznie oznacza to tyle, że jako dodatnie bloki możemy rozważać sufiksy ciągu początkowego rozpoczynające się jedynką i optymalnie jest zamieniać kolejno wszystko w takim bloku na 1 (od lewej do prawej).

Blok ujemny

Ponieważ żadna operacja nie zmienia wartości pierwszego wyrazu ciągu, zatem blok ujemny (jeśli jest niepusty) musi początkowo zaczynać się od $\hat{x}_1 = -1$, a na końcu mieć wszystkie wyrazy równe -1.

Rozważmy pewną optymalną sekwencję operacji, która do tego prowadzi. Zauważmy, że aby ostatecznie wyraz $\hat{x}_i = 0$ zawierał -1, trzeba co najmniej raz wykonać operację o_i. Jeśli natomiast $\hat{x}_i = 1$, to albo potrzebujemy co najmniej dwóch operacji o_i (gdy $x_{i-1} = -1$ w momencie wykonania tej operacji), albo co najmniej jednej operacji zmniejszającej x_i (o co najmniej 2) i co najmniej jednej operacji zwiększającej x_{i-1}. W obu przypadkach potrzebujemy zatem co najmniej dwóch operacji na jedynkę.

Zupełnie podobnie jak wcześniej w przypadku bloku dodatniego, możemy osiągnąć to dolne oszacowanie na liczbę operacji, idąc od lewej do prawej i zamieniając wszystko na -1, wykonując po jednej operacji dla każdego 0 i po dwie dla każdej 1.

Blok zerowy

Przyjmijmy, że po optymalnym ciągu operacji dostajemy blok $x_p, x_{p+1}, \ldots, x_{p+l}$ z zerami, który początkowo nie był tej postaci. Zauważmy, że jeśli $\hat{x}_p = -1$, to równie dobrze moglibyśmy (z zerowym kosztem) rozszerzyć w prawo blok ujemny. Załóżmy zatem, że $\hat{x}_p \geqslant 0$, po którym następowało dokładnie k zer. Jeśli $k \neq l$, to mamy $\hat{x}_{p+1} = \ldots = \hat{x}_{p+k} = 0$ i $\hat{x}_{p+k+1} \neq 0$. Aby wyzerować wyraz $\hat{x}_{p+k+1}$, w którymś momencie każdy z wyrazów $\hat{x}_{p+k-i}$ musiał stać się tego samego znaku co $\hat{x}_{p+k+1}$ (dla i nieparzystego) lub przeciwnego znaku do $\hat{x}_{p+k+1}$ (dla i parzystego), po czym z powrotem stać się zerem. Zatem potrzebujemy co najmniej $2k + 1 + \hat{x}_p$ operacji na wyzerowanie wyrazów $x_p, \ldots, x_{p+k+1}$. Co więcej, blok ujemny nie może być w takim wypadku pusty, zatem zakładamy, że $x_{p-1} = -1$. Zauważmy jednak, że wystarczy nam $k + 1 + \hat{x}_p$ operacji na rozszerzenie bloku ujemnego w prawo o $k + 1$ wyrazów oraz dodatkowa operacja na wyzerowanie x_{p+k+1}, jeśli $\hat{x}_{p+k+1} = 1$. Jeśli więc $k \geqslant 1$, to opłaca nam się rozszerzyć blok ujemny (dla $k = 0$ łatwo sprawdzić, że to również jest prawdą).

Ostatecznie zatem, istnieje optymalne rozwiązanie, w którym blok zerowy zajmuje przestrzeń, na której pierwotnie były same zera, ewentualnie z jedną jedynką na początku (przy czym ten drugi przypadek może mieć miejsce jedynie, gdy blok ujemny jest niepusty).

Implementacja

W poprzednim rozdziale udowodniliśmy, że wystarczy rozważać końcowe ciągi, które składają się z trzech bloków zawierających wartości -1, 0 i 1. Oznaczmy te bloki literami A, B i C. Każdy z tych bloków może być pusty. Udowodniliśmy ponadto, że blok B musi początkowo składać się z samych zer, ewentualnie poprzedzonych jedną jedynką (ale w tym drugim przypadku blok A musi być niepusty). Natomiast blok C (o ile jest niepusty) musi początkowo rozpoczynać się jedynką, a blok A (o ile jest niepusty) musi rozpoczynać się wartością -1 (rys. 1).

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe będzie przebiegać następująco. Rozważamy wszystkie możliwe długości pierwszego bloku. Dla tak ustalonego bloku A znajdujemy najdłuższy blok B, rozpatrując dwa przypadki (gdy B zaczyna się od 0 i od 1). Dzięki temu znamy też

-1	?	?	?	$\frac{0}{1}$	0	0	0	1	?	?	?
-1	-1	-1	-1	0	0	0	0	1	1	1	1
		A				B				C	

Rys. 1: Początkowy i końcowy wygląd ciągu w podziale na bloki.

długość bloku C. Jeśli bloki A, B i C są poprawne, wyznaczamy liczbę potrzebnych operacji do utworzenia tych bloków i porównujemy z dotychczas znalezionym minimum.

Znalezienie najdłuższego bloku B możemy wykonać w czasie stałym. Niech $mZero[i]$ będzie maksymalną liczbą kolejnych zer w ciągu, poczynając od pozycji i. W tym celu wystarczy przeglądać ciąg od końca, pamiętając maksymalną liczbę zer:

```
ileZer := 0;
for k := n downto 1 do begin
  if x̂[k] = 0 then
    ileZer := ileZer + 1
  else
    ileZer := 0;
  mZero[k] := ileZer;
end
```

Koszt utworzenia bloku B jest łatwy do wyznaczenia. W pierwszym przypadku (gdy blok ten składa się z samych zer) koszt jest zerowy. W drugim przypadku (gdy blok ten zaczyna się jedynką i blok A jest niepusty) wystarczy jedna operacja, która wyzeruje pierwszy element.

Dla bloków A i C możemy równie szybko wyznaczyć koszt ich utworzenia. Skupmy się na bloku A (blok C jest analogiczny). Jeśli $\hat{x}_1 = -1$, to blok A o długości a można zamienić na same -1, używając po dwie operacje na każdą jedynkę i jedną operację na każde zero, czyli:

$$zamiany = 2 \cdot jed(1, a) + zer(1, a).$$

Funkcja $jed(l, p)$ zwraca liczbę jedynek w ciągu w przedziale od pozycji l do pozycji p. Analogicznie funkcja $zer(l, p)$ zwraca liczbę zer w tym przedziale. Aby je efektywnie zapisać, możemy uprzednio przygotować tablice, będące sumami prefiksowymi. Przykładowo do wyliczania liczby jedynek:

```
jedynki[0] := 0;
for k := 1 to n do begin
  jedynki[k] := jedynki[k − 1];
  if x̂[k] = 1 then
    jedynki[k] := jedynki[k] + 1;
end
```

Dzięki temu wyznaczenie liczby jedynek w dowolnym przedziale jest proste:

$$jed(l, p) = jedynki[p] - jedynki[l - 1].$$

W ten sposób możemy wyznaczyć liczbę $1, 0$ i -1 w dowolnym przedziale w czasie $O(1)$. Wobec tego każdy blok A analizujemy w czasie stałym, więc złożoność całego rozwiązania wynosi $O(n)$. Zostało ono zaimplementowane w plikach `baj.cpp`, `baj1.cpp` i `baj2.pas`.

Rozwiązanie siłowe $O(n^3)$

Można wprost rozpatrywać wszystkie możliwe podziały ciągu na trzy bloki i dla każdego podziału symulować wykonywanie operacji. Wszystkich podziałów możemy mieć $O(n^2)$, a zliczenie potrzebnych operacji trwa $O(n)$. Stąd dostajemy algorytm w złożoności $O(n^3)$. Takie rozwiązanie zostało zaimplementowane w plikach `bajs1.cpp` i `bajs2.pas` i otrzymywało 24% punktów.

Rozwiązanie wolne $O(n^2)$

Usprawnieniem poprzedniego rozwiązania jest wyliczenie sum prefiksowych, przez co zliczanie potrzebnych operacji możemy wykonać w czasie $O(1)$. Stąd całkowita złożoność to $O(n^2)$. Takie rozwiązanie zostało zaimplementowane w plikach `bajs3.cpp` i `bajs4.pas`. Otrzymywało ono 48% punktów.

Rozwiązanie alternatywne

Wiedząc, że istnieje optymalne rozwiązanie, takie że wszystkie operacje wykonujemy od lewej do prawej i mamy tylko wartości $x_i \in \{-1, 0, 1\}$, możemy napisać rozwiązanie oparte o programowanie dynamiczne.

Wyznaczamy minimalną liczbę operacji do uzyskania ciągu o prefiksie długości i, kończącego się wartościami $0, 1$ i -1. Poruszamy się od najmniejszych i – wyliczamy wynik na podstawie uprzednio wyliczonych wartości. Takie rozwiązanie samo troszczy się o wszystkie przypadki i jest prostsze w implementacji.

Rozwiązanie znajduje się w pliku `baj3.cpp`. Za takie rozwiązanie otrzymywało się maksymalną liczbę punktów.

Testy

Przygotowano 8 grup testów:

- grupa *1* – małe ręczne testy poprawnościowe,
- grupa *2* i *3* – większe testy poprawnościowe,
- testy *4a–8a* – losowe testy z krótkimi przedziałami takich samych liczb,
- testy *4b–8b* – losowe testy z długimi przedziałami (około $\sqrt{n}$) takich samych liczb.