Piotr Chrząstowski — Treść zadania, Opracowanie
Wojciech Śmietanka — Program

Dostępna pamięć: 64 MB. *OI, Etap III, dzień drugi, 15.04.2010*

Piloci

Piloci w Bajtockim Ośrodku Treningowym przygotowują się do pełnienia misji wymagających niezwykłej precyzji i opanowania. Miernikiem sukcesu jest uzyskanie możliwie długiego lotu wzdłuż zadanej trasy bez zbytnich odchyleń — chodzi o utrzymanie w miarę równego kursu. Jest to zadanie trudne, gdyż symulator jest tak czuły, że rejestruje najmniejsze nawet ruchy wolanta[1]*. Zapisuje przy tym jeden parametr zwany wychyleniem. Przed każdym eksperymentem treningowym piloci mają ustawiony poziom tolerancji t; ich celem jest wykonanie możliwie długiego fragmentu lotu o tej własności, że wszystkie pomiary wychyleń w tym czasie będą różniły się nie więcej niż o t. Innymi słowy, fragment lotu od momentu i do momentu j uznamy za mieszczący się w ramach tolerancji t, jeśli wychylenia, począwszy od i-tego pomiaru a skończywszy na j-tym, tworzą taki ciąg $a_i, \ldots, a_j$, że dla każdych a_k, a_l z tego ciągu zachodzi $|a_k - a_l| \leqslant t$.*

Twoim zadaniem jest napisanie programu, który na podstawie tolerancji t i ciągu pomiarów wychyleń wyznaczy długość najdłuższego fragmentu tego ciągu mieszczącego się w ramach zadanej tolerancji.

Wejście

W pierwszym wierszu standardowego wejścia zapisane są dwie liczby całkowite t oraz n ($0 \leqslant t \leqslant 2\,000\,000\,000$, $1 \leqslant n \leqslant 3\,000\,000$), oddzielone pojedynczym odstępem i określające poziom tolerancji oraz liczbę dokonanych pomiarów wychyleń. W drugim wierszu znajdują się pooddzielane pojedynczymi odstępami wartości zmierzonych wychyleń, będące liczbami całkowitymi z zakresu od 1 do 2 000 000 000.

Wyjście

Twój program powinien wypisać w pierwszym i jedynym wierszu standardowego wyjścia jedną liczbę całkowitą: maksymalną długość fragmentu lotu mieszczącego się w ramach zadanej tolerancji.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`3 9` `5 1 3 5 8 6 6 9 10`	`4`

Wyjaśnienie do przykładu: *Mamy dwa najdłuższe fragmenty, oba długości 4:* `5, 8, 6, 6` *oraz* `8, 6, 6, 9`.

[1] urządzenie sterujące w lotnictwie

Rozwiązanie

Rozwiązanie naiwne o koszcie $O(n^4)$

Oznaczmy przez $a_1, a_2, \ldots, a_n$ ciąg kolejnych wartości zmierzonych wychyleń. Powiemy, że przedział $[i, j]$ jest *t-stabilny*, jeśli dla zadanego parametru t spełnia warunki zadania, czyli dla każdych indeksów i', j' takich, że $i \leqslant i' \leqslant j' \leqslant j$, zachodzi $|a_{i'} - a_{j'}| \leqslant t$. Naszym zadaniem jest wyznaczenie długości najdłuższego t-stabilnego przedziału dla danej stałej t.

Na pierwszy rzut oka narzuca się naiwne rozwiązanie polegające na zbadaniu wszystkich par indeksów i, j i wybraniu takiej, która definiuje najdłuższy stabilny segment.

```
1: rekord := 1; { długość najdłuższego stabilnego segmentu dotąd znalezionego }
2: for i := 1 to n do
3:   for j := i + 1 to n do
4:     if t-stabilny(i, j) then
5:       if j − i + 1 > rekord then rekord := j − i + 1;
```

Jest to wyjątkowo paskudne rozwiązanie. Jego koszt wynosi $O(n^2)$ razy koszt wykonania funkcji *t-stabilny*. Mamy bowiem $\frac{n(n-1)}{2}$ par indeksów (i, j) takich, że $i < j$. Jeśli teraz funkcja *t-stabilny* będzie realizowana dla każdego segmentu od nowa, to przy najprymitywniejszym rozwiązaniu każde jej wywołanie zajmie $O(n^2)$ sprawdzeń, czy któraś z par indeksów badanego segmentu nie zaburza warunku stabilności. To dałoby algorytm o łącznym koszcie $O(n^4)$. Wierząc, że każdy czytelnik tego opracowania, być może z obrzydzeniem, ale poradziłby sobie z tym rozwiązaniem, porzućmy ten przykry temat.

Rozwiązanie trochę mniej naiwne, o koszcie $O(n^3)$

Rzecz jasna, natychmiast można o jeden rząd wielkości ten wynik poprawić. Wystarczy zauważyć, że do stwierdzenia t-stabilności nie jest wymagane przebadanie wszystkich par z danego przedziału, a jedynie stwierdzenie, jaka jest w nim minimalna i maksymalna wartość, i sprawdzenie, czy różnica między nimi nie przekracza t. To da się zrobić w koszcie $2n - 2$, a nawet, jak się trochę postaramy, w koszcie nieprzekraczającym $\frac{3}{2}n - 2$ (o szczegółach optymalnego algorytmu wyznaczającego jednocześnie minimum i maksimum można poczytać w książce [18]). Stosując ten algorytm, obniżymy koszt do $O(n^3)$, ale to nas wcale jeszcze nie zadowala — rozwiązanie tej złożoności otrzymywało na zawodach 20-30 pkt. Jego implementacja znajduje się w plikach `pils[6-9].[cpp|pas]`.

Rozwiązanie przyzwoite $O(n^2)$

Zauważmy, że jeśli różnica $|a_i - a_j|$ dla $i < j$ przekracza t, to ani a_i nie może być częścią stabilnego ciągu zawierającego jakikolwiek element za a_j, ani a_j nie może być częścią

jakiegokolwiek stabilnego ciągu zawierającego element występujący przed a_i. Innymi słowy, jeśli przedział $[i, j]$ nie jest stabilny, a jednocześnie $i' \leqslant i, j \leqslant j'$, to przedział $[i', j']$ też nie jest stabilny. Oznacza to, że jeśli dla danego początku przedziału i będziemy posuwali się do przodu końcem przedziału, czyli indeksem j, i stwierdzimy po raz pierwszy, że przedział $[i, j]$ nie jest stabilny, to możemy zaprzestać dalszej analizy dla tego i: wszystkie dalsze indeksy końca przedziału też dadzą negatywny wynik i na pewno rekordu nie pobiją. Zauważmy też, że dla ustalonego i możemy, idąc z j do przodu, na bieżąco aktualizować wartości minimalną i maksymalną z danego przedziału. Po prostu, zwiększając j, sprawdzamy, czy nowa wartość a_j staje się nowym minimum lub maksimum. W ten sposób otrzymujemy dość prosty algorytm kwadratowy:

```
 1: i := 1;
 2: rekord := 1;
 3: while i ⩽ n − rekord do begin
 4: { nie ma co badać większych i, gdyż wtedy takie zbyt duże i }
 5: { nie może być początkiem rekordowego przedziału }
 6:   min := A[i];   { najmniejsza... }
 7:   max := A[i]; { i największa wartość z badanego przedziału }
 8:   j := i + 1;
 9:   stabilny := true;
10:   while (j ⩽ n) and stabilny do begin
11:     if A[j] > max then max := A[j]
12:     else if A[j] < min then min := A[j];
13:     if max − min > t then stabilny := false
14:     else j := j + 1;
15:   end
16:   if j − i > rekord then rekord := j − i;
17:   { nie musieliśmy tego warunku sprawdzać w każdym obrocie pętli; }
18:   { wszak teraz mamy w ręce najlepsze j dla danego i }
19: end
```

Zauważmy, że tu zawsze kończymy z j o 1 za dużym — jest to najmniejszy indeks, dla którego nie ma przedziału stabilnego $[i, j]$. Zatem różnica $j - i$ daje nam poszukiwaną liczbę elementów najdłuższego segmentu zaczynającego się w i.

Ponieważ mamy tu dwie zagnieżdżone pętle, a każda z nich wykonuje co najwyżej liniową liczbę obrotów, więc koszt tego rozwiązania jest kwadratowy. Można było za nie otrzymać ok. 40 pkt. Przykładowa implementacja w pliku `pils3.pas`.

Rozwiązanie jeszcze przyzwoitsze $O(n \log n)$

Dalsze obniżanie kosztu jest możliwe, ale potrzebne są nieco bardziej zaawansowane techniki. Skoro ważne są tylko elementy maksymalny i minimalny z danego przedziału, to wystarczy, jeśli będziemy umieli je szybko wyznaczać. Możemy zatem iść dwoma wskaźnikami: i oraz j w prawo, i jeśli element a_j psuje stabilność, to pchnąć do przodu i, a jeśli nie, to pchnąć j. W tym celu warto użyć sprytnej struktury danych

do reprezentowania multizbioru (czyli zbioru z możliwością powtórzeń elementów) odpowiadającego danemu przedziałowi $[i, j]$, i to takiej, że wyznaczenie minimum i maksimum w multizbiorze jest szybkie.

Jeśli do reprezentacji multizbioru użyjemy drzew zrównoważonych, np. AVL lub czerwono-czarnych[1], to w każdym węźle będziemy poza wartością przechowywali jej krotność, czyli ile razy pojawia się w reprezentowanym multizbiorze. Drzewa AVL mają logarytmiczną wysokość, a elementy minimalny i maksymalny znajdują się na końcu ścieżek odpowiednio samych lewych i samych prawych synów, idąc od korzenia, więc dostęp do nich jest logarytmiczny.

Pseudokod algorytmu jest dość prosty:

```
 1: i := 1;
 2: TwórzMultizbiórAVL(A[1]); { tworzy drzewo AVL z elementem A[1] }
 3: rekord := 1;
 4: min := A[1];
 5: max := A[1];
 6: j := 2;
 7: while j ⩽ n do begin
 8:    WstawAVL(A[j]);
 9:    if A[j] > max then max := A[j]
10:    else if A[j] < min then min := A[j];
11:    while max − min > t do begin
12:    { jeśli A[j] nie zmieniło min lub max lub jeśli zmieniło, ale nie tak, }
13:    { żeby utracić t-stabilność, to pętla ta nie wykona się ani razu }
14:       UsuńAVL(A[i]);
15:       Aktualizuj(min, max);
16:       i := i + 1;
17:    end
18:    j := j + 1;
19:    if j − i > rekord then rekord := j − i;
20: end
```

Aktualizację *min* i *max* związaną z usuwaniem elementów $A[i]$ można zrobić sprytniej, modyfikując odpowiednio procedury *WstawAVL* i *UsuńAVL*, ale nie będziemy się nad tym specjalnie rozwodzili. Tak czy siak, nawet w takiej wersji taka aktualizacja ma koszt co najwyżej logarytmiczny.

W rezultacie dostaliśmy algorytm o liniowej liczbie obrotów pętli. Indeks i lub j wzrasta o 1 przy każdym obrocie każdej pętli; nieistotne tu jest to, że jedna jest zagnieżdżona w drugiej — łącznie mogą one wykonać się co najwyżej $2n-2$ razy. W każdym obrocie pętli, dzięki zastosowaniu drzew AVL, mamy co najwyżej logarytmiczny koszt wykonania wszystkich znajdujących się w niej operacji, stąd końcowy koszt $O(n \log n)$. Implementację takiego rozwiązania można znaleźć w pliku `pils2.cpp`.

[1] O drzewach zrównoważonych można przeczytać m.in. w książkach [19, 21]. Programujący w C++ mogli użyć na zawodach gotowej implementacji multizbioru za pomocą drzewa czerwono-czarnego, a mianowicie klasy `multiset`.

Warto w tym miejscu dodać, że istnieją także inne rozwiązania niniejszego zadania o złożoności czasowej $O(n \log n)$, które różnią się głównie zastosowaną strukturą danych. Szukana struktura danych musi umożliwiać efektywne wyznaczanie minimów i maksimów w poszczególnych segmentach wyjściowego ciągu. Dla przykładu, bardzo łatwo jest do tego celu zaadaptować statyczne drzewo przedziałowe[2], przechowujące wszystkie elementy wyjściowego ciągu. Szczegóły tego rozwiązania pozostawiamy do dopracowania Czytelnikowi. Dodajmy tylko, że drzewa przedziałowe są nieco bardziej efektywne od opisanej wcześniej reprezentacji multizbioru: mają mniejszą stałą w złożoności czasowej i pamięciowej. Przykłady implementacji w plikach `pils4.cpp` i `pils5.pas`.

Rozwiązania o koszcie $O(n \log n)$ otrzymywały na zawodach od ok. 60 do ponad 90 punktów w zależności od zużycia pamięci oraz stałej ukrytej w notacji O.

Rozwiązania liniowe

Zauważmy, że trzymanie całego multizbioru odpowiadającego przedziałowi $[i, j]$ nie jest konieczne. Tak naprawdę potrzebne jest nam przede wszystkim ustalenie, jakie są wartości minimum i maksimum w zadanym przedziale. Jeśli więc będziemy w stanie zrobić to szybko, to cała reszta wartości multizbioru staje się nieistotna. Podamy teraz dwa różne rozwiązania o liniowym koszcie.

Rozwiązanie liniowe specyficzne dla tego zadania

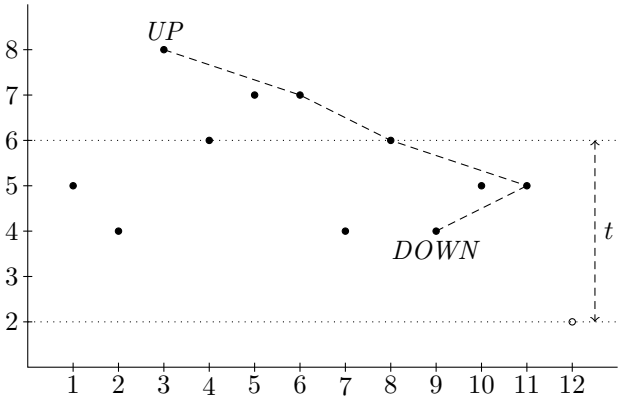
Określmy dla każdego $j \in \{1, \ldots, n\}$ wartość najmniejszego indeksu i, dla którego przedział $[i, j]$ jest t-stabilny. Oznaczmy tę wartość przez $p[j]$. Szukamy najdłuższego przedziału t-stabilnego, więc możliwym rozwiązaniem jest wyznaczenie $p[j]$ dla każdego j i zbadanie, która z par wartości $(p[j], j)$ definiuje najdłuższy przedział, czyli czemu jest równe maksimum z wartości $j - p[j] + 1$. Niech $njm(i, j)$ oraz $njw(i, j)$ oznaczają odpowiednio najmniejszą i największą wartość z przedziału domkniętego $[i, j]$.

Zastanówmy się nad odpowiedzią, myśląc indukcyjnie. To jest sposób, za pomocą którego można dojść do wielu efektywnych rozwiązań. Metoda ta polega w naszym przypadku na przeglądaniu kolejnych elementów ciągu a_j dla $j = 2, \ldots, n$ i opracowaniu schematu przejścia od $j - 1$ do j. Załóżmy indukcyjnie, że znamy wartość $p[j-1]$ i zastanawiamy się, jak wykorzystać tę wiedzę przy wyznaczaniu $p[j]$. Konieczne jest tu dostrzeżenie, dlaczego wartość $p[j]$ miałaby być inna niż $p[j-1]$. Może się tak zdarzyć tylko wtedy, gdy element a_j psuje stabilność, czyli gdy

$$A[j] - njm(p[j-1], j-1) > t \quad \text{albo} \quad njw(p[j-1], j-1) - A[j] > t.$$

Gdy żadna z tych sytuacji nie zachodzi, to $p[j]$ po prostu równa się $p[j-1]$. Natomiast gdy tak dobrze nie jest, powinniśmy znaleźć najwcześniejszy indeks wartości, która nie destabilizuje ciągu kończącego się w j. Rozważmy, wobec tego, trzy przypadki:

[2] Jest to struktura danych zbudowana nad ciągiem, pozwalająca (w podstawowej wersji) w czasie $O(\log n)$ obliczać wyniki dowolnego działania łącznego (na przykład sumy, minimum, maksimum) na zadanym przedziale tego ciągu oraz równie szybko aktualizować jego wyrazy. Więcej na temat drzew przedziałowych można znaleźć np. w opracowaniu zadania *Latarnia* w tej książeczce oraz w podanych tam odnośnikach.



Rys. 1: Ilustracja przykładu (1).

1. Jeżeli $A[j-1] > A[j]$, to zagrożenie jest tylko z góry. Element $A[j]$ może się okazać zbyt mały dla pewnych elementów z segmentu $[p[j-1], j-1]$. Zatem powinniśmy w takim przypadku znaleźć najmniejszy w przedziale $[p[j-1], j-1]$ indeks k taki, że żaden z elementów $A[k], \ldots, A[j-1]$ nie przekracza $A[j]$ o więcej niż t. Dobrze byłoby więc znać indeksy malejących lewostronnych maksimów w tym przedziale, poczynając od wartości największej w całym przedziale (lewostronne maksimum w przedziale to taka wartość, że każda dalsza jest od niej mniejsza).

2. Jeżeli $A[j-1] < A[j]$, to zagrożenie jest tylko z dołu. Tym razem powinniśmy, analogicznie do poprzedniego przypadku, mieć pod ręką indeksy rosnących lewostronnych minimów w tym przedziale, poczynając od najmniejszej wartości w całym przedziale (lewostronne minimum w przedziale to taka wartość, że każda dalsza jest od niej większa).

3. Jeżeli $A[j] = A[j-1]$, to nie musimy nic robić poza sprawdzeniem, czy nie pobiliśmy rekordu — w tym przypadku bowiem zachodzi $p[j] = p[j-1]$.

Widać więc, że kluczem do sukcesu jest tu posiadanie informacji o dwóch specyficznych ciągach: lewostronnych maksimów i lewostronnych minimów w przedziale $[p[j-1], j-1]$. W obu przypadkach, poza samymi wartościami, trzeba też będzie pamiętać ich indeksy, bo wartości występujące w ciągach wcale nie muszą być sąsiadami w oryginalnym ciągu wychyleń.

Przykładowo, dla $t = 4$ i stabilnego dla tej wartości ciągu

$$5, 4, 8, 6, 7, 7, 4, 6, 4, 5, 5 \tag{1}$$

interesujące nas malejące wartości lewostronnych maksimów to $8, 7, 6, 5$ z odpowiadającymi im indeksami $3, 6, 8, 11$, a rosnące lewostronne minima to $4, 5$ z odpowiadającymi im indeksami $9, 11$. Tutaj, rzecz jasna, $p[11] = 1$.

Teraz wydłużenie ciągu o liczbę o indeksie 12 — być może psującą stabilność ciągu — może spowodować modyfikację każdego z tych ciągów. Gdybyśmy, na przykład, do

naszego ciągu wejściowego dodali na końcu 2 na pozycji dwunastej, to w tym konkretnym przypadku pierwszy z ciągów dla nowego, krótszego przedziału kończącego na tej dwójce stanie się równy $6, 5, 2$, o indeksach $8, 11, 12$, zaś drugi stanie się jednoelementowy: 2, o indeksie 12. Dla wydłużonego o dwójkę ciągu wartość $p[12]$ będzie równa 7 — ciąg 4-stabilny kończący się nowo wstawioną dwójką zaczyna się od przedostatniej czwórki. Zauważmy tu ciekawą rzecz. Tak naprawdę nie jest dla nas istotne to, ile wynosi aktualnie $p[j]$. Powinniśmy je wyznaczyć tylko wtedy, gdy przy wzroście j pozostawiamy t-stabilny przedział z tym samym początkiem, co dla $j-1$. Jedynie wtedy możemy pobić rekord.

Dokładnie rzecz ujmując, definiujemy dla każdego j ciąg malejący lewostronnych maksimów UP_j następująco: pierwszym jego elementem jest największa wartość z przedziału $[p[j], j]$ występująca możliwie najpóźniej (czyli, spośród potencjalnie wielu równych największych elementów, ten o największym indeksie). Kolejne elementy tego ciągu tworzą ostatnie wystąpienia wartości mniejszych od poprzednich elementów, o ile spełnią warunek, że aż do końca (czyli do j) wszystkie elementy są od nich mniejsze. Ciąg rosnący lewostronnych minimów $DOWN_j$ definiujemy analogicznie, poczynając od ostatniego wystąpienia elementu najmniejszego z przedziału $[p[j], j]$.

Zauważmy, że oba ciągi, o które dbamy, pozwalają w dość prosty sposób wyznaczyć wartość $p[j]$. W przypadku pierwszego z nich — malejącego, wszystkie jego wartości przekraczające o więcej niż t wartość $A[j]$ odpadają wraz z tymi elementami, które znajdują się przed nimi. Zatem kandydatem na $p[j]$ jest indeks o jeden większy od ostatniego elementu w ciągu UP_j, który psuje stabilność. Jeśli zatem pierwszy element tego ciągu nie jest zbyt duży, to na tym kończymy (tzn. wykorzystamy indeks obliczony w ten sposób dla $j-1$). Jeśli natomiast pierwszy element przerasta $A[j]$ o więcej niż t, to usuwamy z początku ciągu UP_j wszystkie elementy zbyt duże i zatrzymujemy się na pierwszym, którego wartość jest większa od $A[j]$ o co najwyżej t, pamiętając powiększony o jeden indeks ostatnio usuniętego elementu. Może się zdarzyć, że wszystkie elementy ciągu zostaną z niego usunięte.

Analogicznie postępujemy w przypadku drugiego ciągu, wyznaczając drugiego kandydata na $p[j]$. Obaj kandydaci są pierwszymi elementami, które nie protestują z powodu pojawienia się $A[j]$. Są to elementy o indeksach o 1 większych od ostatnio usuniętych wartości. Pierwszy z nich gwarantuje, że jest najmniejszym indeksem takim, że wszystkie wartości pomiędzy nim a j nie przekraczają $A[j]$ o więcej niż t. Natomiast drugi, że jest najmniejszym indeksem takim, że wszystkie wartości między nim a j są mniejsze od $A[j]$ o co najwyżej t. Wartość $p[j]$, a więc nowe i, jest więc równa większemu z tych dwóch kandydatów.

Teraz pora na aktualizację ciągów UP i $DOWN$ z drugiej strony. Nowa wartość $A[j]$ musi stać się ostatnim elementem każdego z tych ciągów. W przypadku ciągu malejącego powinniśmy zatem usunąć z końca wszystkie mniejsze wartości i — jeśli ta wartość jeszcze nie występowała — dodać na końcu wartość $A[j]$, a jeśli występowała, to nic nie dodawać, tylko zaktualizować indeks ostatniego wystąpienia tej wartości. W przypadku drugiego ciągu — analogicznie powinniśmy usunąć wszystkie większe wartości i dodać ją na końcu, jeśli jej jeszcze nie było, a jeśli była, to też zaktualizować indeks ostatniego wystąpienia. Wykonanie tej czynności zagwarantuje nam zachowanie niezmiennika pętli: ciągi UP i $DOWN$ będą spełniały warunki definicji,

a jednocześnie będziemy mieli informację o ostatnim wystąpieniu tej wartości.

Z punktu widzenia jednorodności kodu warto dopiero teraz sprawdzić, czy nowy element $A[j]$ wydłużył najdłuższy do tej pory zauważony ciąg stabilny. Wystarczy w tym celu sprawdzić, czy $j - i$ nie pobiło rekordu.

Nasz algorytm rozbija się zatem na trzy fazy:

1. Aktualizacja ciągów *UP* i *DOWN* z początku.
2. Aktualizacja ciągów *UP* i *DOWN* z końca.
3. Aktualizacja rekordu.

Oba nasze ciągi wygodnie jest zrealizować za pomocą tzw. kolejki dwustronnej, czyli struktury, która pozwala dodawać i usuwać elementy z obu stron. Konkretnie chodzi nam o następujące operacje, przy założeniu, że elementami kolejki będą rekordy o dwóch polach: *val* i *ind*, reprezentujących odpowiednio wartość i indeks danego elementu ciągu:

- Inicjuj(q) — procedura inicjująca kolejkę q na pustą,
- Pusta(q) — funkcja sprawdzająca pustość kolejki q,
- Pierwszy(q) — funkcja przekazująca jako wynik pierwszy element kolejki q,
- UsuńPierwszy(q) — funkcja usuwająca pierwszy element kolejki q i przekazująca go jako wynik,
- Ostatni(q) — funkcja przekazująca jako wynik ostatni element kolejki q,
- UsuńOstatni(q) — funkcja usuwająca ostatni element kolejki q i przekazująca go jako wynik,
- WstawNaKoniec(q, x) — procedura wstawiająca na koniec kolejki q element x.

Można taką strukturę danych zrealizować np. za pomocą listy dwukierunkowej albo w tablicy za pomocą bufora cyklicznego[3]. Każda z podanych operacji dokonuje się w tych implementacjach w czasie stałym.

Pseudokod rozwiązania liniowego

W pseudokodzie zmienna *ost_up* wskazuje na indeks ostatniego elementu ciągu *UP*, który zaburzał stabilność od góry, zaś *ost_down* wskazuje na indeks ostatniego elementu ciągu *DOWN*, który zaburzał stabilność od dołu. Wszystkie wyliczenia złożonych warunków logicznych są leniwe, czyli przerywane, gdy tylko wynik stanie się wiadomy. Tak właśnie dzieje się w programach napisanych m.in. w C++ i Pascalu.

```
1: Inicjuj(UP); { inicjujemy ciąg malejący na pusty }
2: Inicjuj(DOWN); { inicjujemy ciąg rosnący na pusty }
```

[3]Szczegóły implementacji bufora cyklicznego można znaleźć pod adresem `http://wazniak.mimuw.edu.pl/index.php?title=Wst%C4%99p_do_programowania` w rozdziale *Stosy i kolejki*.

```
3:  ost_up := 0; ost_down := 0;
4:  rekord := 1;
5:  for j := 1 to n do begin
6:    x.ind := j;
7:    x.val := A[j];
8:    while (not Pusta(UP)) and (Pierwszy(UP).val − t > A[j]) do
9:      ost_up := UsuńPierwszy(UP).ind;
10:   while (not Pusta(DOWN)) and (Pierwszy(DOWN).val + t < A[j]) do
11:     ost_down := UsuńPierwszy(DOWN).ind;
12:   i := max(ost_up + 1, ost_down + 1);
13:   { koniec fazy pierwszej }
14:   while (not Pusta(UP)) and (Ostatni(UP).val < A[j]) do
15:     UsuńOstatni(UP);
16:   if Pusta(UP) or (Ostatni(UP).val > A[j]) then
17:     WstawNaKoniec(UP, x)
18:   else Ostatni(UP).ind := j;
19:   while (not Pusta(DOWN)) and (Ostatni(DOWN).val > A[j]) do
20:     UsuńOstatni(DOWN);
21:   if Pusta(DOWN) or (Ostatni(DOWN).val < A[j]) then
22:     WstawNaKoniec(DOWN, x)
23:   else Ostatni(DOWN).ind := j;
24:   { koniec fazy drugiej }
25:   if j − i + 1 > rekord then rekord := j − i + 1;
26: end
```

Pozostaje pytanie o złożoność tego rozwiązania. Na pierwszy rzut oka nie widać, żeby to miało być rozwiązanie o liniowym koszcie. Wszak wewnątrz pętli po wszystkich j mamy pętle przetwarzające ciągi *UP* i *DOWN* — oba o potencjalnie liniowej długości. Jednak to przetwarzanie łącznie we wszystkich obrotach pętli po j nie może zająć więcej niż liniowo wiele: przecież wewnętrzne pętle usuwają elementy nadmiarowe, a każdy z nich był wstawiony tylko raz do każdego z tych ciągów. Nie można wielokrotnie usuwać czegoś, co tylko raz się pojawiło. Zatem łączna liczba wszystkich usunięć też jest liniowa, bo liniowo dużo elementów do tych ciągów wstawiliśmy.

Implementacje rozwiązań podobnych do opisanego tutaj można znaleźć w plikach `pil.cpp`, `pil2.pas` i `pil3.cpp`.

Odnośniki

W tym miejscu warto dodać, że zadanie o *Pilotach* sprowadziliśmy w tym opracowaniu tak naprawdę do następującego problemu. Należy na początkowo pustym ciągu liczb wykonać n operacji postaci: wstaw zadany element na początku ciągu lub usuń jeden element z końca ciągu. Po każdej operacji musimy umieć podać minimum (równoważnie: maksimum) z elementów ciągu.

Rozwiązania tak sformułowanego problemu o koszcie czasowym $O(n)$, działające na takiej samej zasadzie jak wyżej opisane, pojawiały się już na zawodach olimpijskich,

patrz opracowanie zadania *BBB* z XV Olimpiady Informatycznej [15] lub opracowanie zadania *Sound* z Bałtyckiej Olimpiady Informatycznej 2007 [4].

Range Minimum Query

Dużo bardziej wyrafinowane rozwiązanie wyjściowego problemu otrzymujemy, stosując tzw. algorytm RMQ. Pozwala on szybko odpowiadać na pytania, jaka jest minimalna/maksymalna wartość z jakiegoś segmentu ciągu. Po wstępnym przetworzeniu w czasie liniowym udziela odpowiedzi w czasie stałym. To rozwiązanie jest dość trudne do zaimplementowania. Jego opis można znaleźć w dwuczęściowym artykule poświęconym problemom RMQ i LCA, w numerach 9/2007 i 11/2007 czasopisma *Delta*[5].

Testy

Ze względu na fakt, że do tego zadania można było wymyślić wiele różnych heurystyk, zestaw testów jest stosunkowo liczny.

1. Testy *a* są losowe.
2. Testy *b* to na zmianę losowo: albo $+1$, albo -1 w stosunku do poprzedniej wartości.
3. Testy *c* generują sinusoidę. Długość okresu różni się w zależności od testu.
4. W testach *d* różnice między kolejnymi wyrazami ciągu są losowane z pewnego przedziału.
5. W testach *e* ciąg generalnie powoli rośnie, lokalnie zdarzają się zaburzenia tego trendu.
6. Testy *f* zawierają przypadki brzegowe.

Poniżej znajduje się tabela z podstawowymi statystykami poszczególnych testów.

Nazwa	n	t
pil1a.in	50	5 000
pil1b.in	50	6
pil1c.in	50	2 500
pil1d.in	50	16
pil1e.in	50	35
pil1f.in	1	1
pil2a.in	200	20 000
pil2b.in	200	8

Nazwa	n	t
pil2c.in	200	10 000
pil2d.in	200	66
pil2e.in	200	280
pil2f.in	5	0
pil3a.in	500	1 000 000 000
pil3b.in	500	9
pil3c.in	500	500 000 000
pil3d.in	500	166

[4] Opracowania zadań z zawodów BOI 2007 można znaleźć — tylko w wersji anglojęzycznej — na stronie internetowej `http://www.boi2007.de/tasks/book.pdf`

[5] Artykuły dostępne także na stronie internetowej czasopisma: `http://www.mimuw.edu.pl/delta/`

Nazwa	n	t
pil3e.in	500	1 100
pil4a.in	10 000	30 000
pil4b.in	10 000	14
pil4c.in	10 000	15 000
pil4d.in	10 000	3 333
pil4e.in	10 000	100 000
pil5a.in	50 000	1 000 000 000
pil5b.in	50 000	16
pil5c.in	50 000	500 000 000
pil5d.in	50 000	16 666
pil5e.in	50 000	1 115 000
pil6a.in	100 000	10 000 000
pil6b.in	100 000	17
pil6c.in	100 000	5 000 000
pil6d.in	100 000	33 333
pil6e.in	100 000	3 000 000
pil7a.in	300 000	300 000
pil7b.in	300 000	19
pil7c.in	300 000	150 000
pil7d.in	300 000	100 000
pil7e.in	300 000	9 000 000

Nazwa	n	t
pil8a.in	1 000 000	1 000 000 000
pil8b.in	1 000 000	20
pil8c.in	1 000 000	500 000 000
pil8d.in	1 000 000	333 333
pil8e.in	1 000 000	30 000 000
pil9a.in	1 500 000	1 000 000 000
pil9b.in	1 500 000	21
pil9c.in	1 500 000	500 000 000
pil9d.in	1 500 000	500 000
pil9e.in	1 500 000	45 000 000
pil10a.in	3 000 000	1 000 000 000
pil10b.in	3 000 000	22
pil10c.in	3 000 000	500 000 000
pil10d.in	3 000 000	1 000 000
pil10e.in	3 000 000	90 000 000
pil10f.in	3 000 000	2 000 000 000

XXII Międzynarodowa Olimpiada Informatyczna,

Waterloo, Kanada 2010