Jacek Tomasiewicz
Treść zadania, Opracowanie

Bartłomiej Wołowiec
Program

Dostępna pamięć: 32 MB.

OI, Etap II, dzień drugi, 10.02.2011

Temperatura

W Bajtockim Instytucie Meteorologicznym (BIM) codziennie mierzy się temperaturę powietrza. Pomiaru dokonuje się automatycznie, po czym jest on drukowany. Niestety, tusz w drukarce dawno wysechł... Pracownicy BIM przekonali się o tym jednak dopiero, gdy Bajtocka Organizacja Meteorologiczna (BOM) zwróciła się z zapytaniem dotyczącym temperatury.

Na szczęście stażysta Bajtazar systematycznie notował temperatury wskazywane przez dwa zwykłe termometry okienne, znajdujące się na północnej i południowej ścianie budynku BIM. Wiadomo, że temperatura wskazywana przez termometr na południowej ścianie budynku nigdy nie jest niższa niż faktyczna, a wskazywana przez termometr na północnej ścianie budynku nigdy nie jest wyższa niż faktyczna. Nie wiadomo więc dokładnie, jaka danego dnia była temperatura, ale wiadomo, w jakim mieściła się przedziale.

Na szczęście, zapytanie BOM nie dotyczy dokładnych temperatur, a jedynie najdłuższego przedziału czasu, w którym temperatura była niemalejąca (tj. każdego kolejnego dnia była taka sama jak poprzedniego lub wyższa). Szef BIM wie, że BOM zależy na tym, aby znaleźć możliwie najdłuższy taki okres czasu. Postanowił więc zatuszować swoją wpadkę i polecił Bajtazarowi znaleźć, na podstawie jego notatek, najdłuższy taki przedział czasu, w którym temperatura **mogła być** *niemalejąca. Bajtazar nie bardzo wie, jak sobie poradzić z tym zadaniem, więc poprosił Cię o napisanie programu, który znajdzie taki najdłuższy przedział czasu.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 1\,000\,000$) oznaczająca liczbę dni, przez które Bajtazar notował temperatury. Wyniki pomiarów z dnia i są zapisane w wierszu o numerze $i + 1$. Każdy z takich wierszy zawiera dwie liczby całkowite x i y ($-10^9 \leqslant x \leqslant y \leqslant 10^9$). Oznaczają one, odpowiednio, minimalną i maksymalną temperaturę, jaka mogła być danego dnia.

W pewnej liczbie testów, wartych łącznie 50 punktów, temperatury nigdy nie spadają poniżej -50 stopni i nigdy nie wzrastają powyżej 50 stopni ($-50 \leqslant x \leqslant y \leqslant 50$).

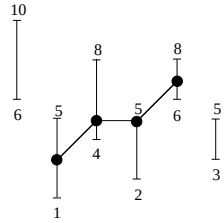
Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien wypisać jedną liczbę całkowitą, oznaczającą maksymalną liczbę dni, przez które temperatura w Bajtocji mogła być niemalejąca.

Przykład

Dla danych wejściowych:

```
6
6 10
1 5
4 8
2 5
6 8
3 5
```



poprawnym wynikiem jest:

```
4
```

Rozwiązanie

Kiedy przedział jest dobry?

Zanim przejdziemy do rozwiązania zadania, spróbujmy zastanowić się nad nieco prostszym problemem. Jak sprawdzić, czy w całym okresie czasu, od pierwszego do ostatniego dnia, temperatura mogła być niemalejąca?

Do odpowiedzi na to pytanie posłuży nam łatwa obserwacja. Załóżmy, że mamy niemalejący ciąg temperatur $(t_i)_{i=1}^n$, spełniających zadane ograniczenia $x_i \leqslant t_i \leqslant y_i$. Wówczas możemy zmniejszyć t_i, tak aby zachodziła równość $t_i = \max(x_i, t_{i-1})$. Po wykonaniu tej operacji dla wszystkich wyrazów otrzymamy ciąg, który wciąż spełnia wszystkie żądane warunki.

Teraz wiemy, że aby sprawdzić, czy istnieje jakikolwiek dobry ciąg temperatur, wystarczy sprawdzić, czy dobry jest ciąg dany wzorem

$$t_1 = x_1, \qquad t_i = \max(x_i, t_{i-1}) \text{ dla } i > 1.$$

Można to więc bardzo łatwo stwierdzić w czasie $O(n)$.

W ten sposób uzyskujemy pierwsze rozwiązanie właściwego zadania. Wystarczy dla każdego przedziału sprawdzić, czy jest dobry, i zapamiętać najdłuższy przedział, dla którego odpowiedź okazała się pozytywna. Oto pseudokod ilustrujący taki algorytm:

```
1:  wynik := 0;
2:  for j := 1 to n do
3:    for k := j to n do begin
4:      dobry := true;
5:      temp := x[j];
6:      for i := j + 1 to k do
7:        if y[i] ⩾ temp then
8:          temp := max(temp, x[i])
9:        else
10:         dobry := false;
```

```
11:       if dobry then
12:         wynik := max(wynik, k − j + 1);
13:     end
14:   return wynik;
```

Złożoność czasowa tego rozwiązania to $O(n^3)$, ponieważ wszystkich przedziałów czasu jest $O(n^2)$, a ich łączna długość to $O(n^3)$. Można było za nie uzyskać około 16 punktów.

Nieco szybsze rozwiązanie

Złożoność sześcienna jest bardzo daleka od naszych oczekiwań. Zauważmy jednak, że przedstawiony algorytm dla kolejnych fragmentów o tym samym początku wykonuje dokładnie te same obliczenia. Jeśli rozpoczynamy konstrukcję ciągu w j-tym dniu i w pewnym momencie natrafiamy na dzień, dla którego nie możemy wybrać temperatury, to znamy najdłuższy dobry przedział czasu o początku w j-tym dniu. Możemy więc, dla każdego dnia z osobna (uznając, że jest początkiem przedziału), zachłannie sprawdzać, jak długo temperatura mogła być niemalejąca.

```
1:  wynik := 1;
2:  for j := 1 to n do begin
3:    temp := x[j];
4:    i := j + 1;
5:    while i ⩽ n and y[i] ⩾ temp do begin
6:      temp := max(temp, x[i]);
7:      i := i + 1;
8:    end
9:    wynik := max(wynik, i − j);
10: end
11: return wynik;
```

Powyższy algorytm działa w czasie $O(n^2)$, gdyż dla każdego początku przedziału sprawdzamy $O(n)$ kolejnych dni. Za takie rozwiązanie można było uzyskać około 25 punktów. Przykładowa implementacja znajduje się w plikach `tems1.cpp` oraz `tems5.pas`.

Ciekawe rozwiązanie nieoptymalne

Okazuje się, że poprzedni algorytm można jeszcze przyspieszyć. To przyspieszenie samo w sobie jest interesujące, natomiast nie jest ono potrzebne do uzyskania rozwiązania wzorcowego.

Otóż w algorytmie kwadratowym dla każdego j wyznaczamy największe takie i, że przedział $[j, i]$ jest dobry. Zauważmy, że gdy obliczamy tę wartość dla różnych j, otrzymany wynik zależy tylko od bieżących wartości zmiennych i oraz *temp*. W szczególności, gdy w linii 6 zachodzi $temp \leqslant x[i]$ i *de facto* dokonujemy przypisania $temp := x[i]$,

możemy z góry stwierdzić, że najdłuższy przedział zaczynający się w j kończy się dokładnie tam, gdzie najdłuższy przedział zaczynający się w i. Kolejne obroty pętli będą bowiem w obu przypadkach dokładnie takie same.

Możemy wobec tego wyznaczać wyniki dla j w kolejności malejącej, spamiętując je i wykorzystując do dalszych obliczeń. Przy okazji pozbywamy się zmiennej *temp*, bo i tak nie zmienialibyśmy jej wartości. Innymi słowy, kiedy w poprzednim rozwiązaniu wartość *temp* wzrasta do wartości $x[i]$ (lub $x[i] = temp$), w poniższym rozwiązaniu wykorzystujemy spamiętaną wartość $najdl[i]$.

```
1:  wynik := 1;
2:  for j := n downto 1 do begin
3:    i := j + 1;
4:    while i ⩽ n and y[i] ⩾ x[j] do begin
5:      if x[j] ⩽ x[i] then begin
6:        i := najdl[i] + 1;
7:        break;
8:      end else i := i + 1;
9:    end
10:   najdl[j] := i − 1;
11:   wynik := max(wynik, i − j);
12: end
13: return wynik;
```

Zastanówmy się, jaka jest złożoność takiego algorytmu. Na pewno daje się oszacować przez $O(n^2)$. Spróbujmy jednak uzależnić ją również od Δ — liczby różnych wartości w tablicy x. Oczywiście cały program poza pętlą w liniach 4–9 wykonuje się w czasie liniowym. Wystarczy więc oszacować łączną liczbę obrotów tej pętli.

Ustalmy pewną konkretną wartość M. Niech $j_1 < j_2 < \ldots < j_m$ będą wszystkimi indeksami, dla których wartość tablicy x wynosi M. Zauważmy, że gdy dla $j = j_k$ w pętli dojdziemy do $i = j_{k+1}$, natychmiast wyjdziemy z pętli. Stąd obliczenie wyników dla $j_1, \ldots, j_m$ zajmuje czas $O(n)$, a złożoność czasowa całego algorytmu to $O(n\Delta)$.

Oczywiście $\Delta \leqslant \max_i x[i] - \min_i x[i] + 1$. Dzięki temu taki algorytm przechodzi testy z ograniczonymi wartościami temperatur, co sprawia, że dostaje 50 punktów. Przykładowy implementujący go program można znaleźć w pliku `tems11.cpp`.

Podobną złożoność można uzyskać na kilka innych sposobów, przykładowe implementacje są w plikach `tems[2,3,6,7,8,9,10].[cpp|pas]`. Można, na przykład, zastosować programowanie dynamiczne, w którym dla danego j i wartości *temp* obliczamy (także w kolejności malejących j) najdłuższy przedział zaczynający się w j, dla którego istnieje niemalejący ciąg temperatur spełniających odpowiednie warunki, a przy tym rozpoczynający się temperaturą równą *temp*. To rozwiązanie, w podstawowej wersji, zużywa aż $\Theta(n\Delta)$ pamięci. Jednakże zawsze przy obliczaniu wartości dla j korzystamy tylko z wartości dla $j + 1$, więc można używać na zmianę tylko dwóch tablic rozmiaru $\Theta(\Delta)$.

Rozwiązanie wzorcowe

Poprzednie rozwiązanie ciągle nie jest satysfakcjonujące, ponieważ dla $\Delta = \Theta(n)$ zdarza mu się działać w czasie $\Theta(n^2)$. Aby otrzymać istotnie szybszy algorytm, zastosujemy technikę nazywaną czasami *metodą gąsienicy*. Służy ona właśnie do znajdowania dla każdego j najdłuższego w jakimś sensie dobrego przedziału zaczynającego się w j. Wymagana jest przy tym monotoniczność, tzn. aby każdy podprzedział dobrego przedziału był dobry (co ma miejsce w naszym przypadku).

Przedstawmy zarys tej metody. Zaczynamy od jednoelementowego przedziału zawierającego pierwszy element. Następnie przesuwamy prawy koniec przedziału, dopóki przedział pozostaje dobry, w ten sposób uzyskujemy maksymalny przedział zaczynający się na pierwszej pozycji. Teraz zwiększamy lewy koniec przedziału o 1 i znów zaczynamy przesuwać prawy koniec przedziału, począwszy od miejsca, w którym skończyliśmy wcześniej, itd.

Na cały proces składa się liniowa liczba przesunięć końców przedziałów. Po każdym przesunięciu prawego końca musimy sprawdzić, czy ta operacja nie spowoduje, że przedział przestanie być dobry. Podobną operację wykonywaliśmy już w poprzednich algorytmach. Potrzebowaliśmy do tego zmiennej *temp*, którą aktualizowaliśmy przy przesuwaniu prawego końca przedziału. W tym rozwiązaniu musimy jeszcze umieć ją aktualizować przy przesuwaniu lewego końca. Wystarczy nam do tego proste spostrzeżenie, że jeśli aktualnie rozważany przedział to $[j, i]$, to bieżąca wartość zmiennej *temp* powinna być równa $\max(x[j], x[j+1], \ldots, x[i])$. Możemy wobec tego zapisać następujący pseudokod, w którym już nie używamy *explicite* zmiennej *temp*:

```
1: wynik := 1;
2: i := 1;
3: for j := 1 to n do begin
4:    while i ⩽ n and max(x[j], x[j + 1], ..., x[i − 1]) ⩽ y[i] do
5:       i := i + 1;
6:    wynik := max(wynik, i − j);
7: end
8: return wynik;
```

Obliczanie maksimów

Teraz pozostaje nam tylko szybkie obliczanie wartości postaci

$$\max(x_i, x_{i+1}, \ldots, x_{j-1}).$$

Znajdowanie maksymalnego (lub minimalnego) elementu w zadanym fragmencie ciągu jest dosyć standardowym problemem, znanym też pod nazwą RMQ (ang. *Range Minimum Query*). Jest kilka klasycznych struktur danych pozwalających na rozwiązanie tego problemu. Są to drzewa przedziałowe, zwane też licznikowymi (rozmiar struktury $O(n)$, koszt zapytania $O(\log n)$), struktura danych podobna do słownika podsłów bazowych (rozmiar $O(n \log n)$, koszt zapytania $O(1)$), jest wreszcie znane rozwiązanie optymalne (niestety trudne w implementacji i dlatego raczej niestosowane w praktyce), w którym po wstępnych obliczeniach w czasie $O(n)$ możemy odpowiadać na

zapytania w czasie stałym[1]. Ze względu na małe ograniczenie pamięciowe w tym zadaniu, najlepsza z wymienionych metod to zastosowanie drzew przedziałowych — już za takie rozwiązanie można było uzyskać 100 punktów. Przykładowa implementacja znajduje się w plikach `tem2.cpp` oraz `tem3.pas`.

Maksima szczególnej postaci

Aby otrzymać prostsze i efektywniejsze rozwiązanie naszego zadania, należy zauważyć, że występujące w nim zapytania są bardzo szczególnej postaci — przedział, o który pytamy, „pełznie" przez tablicę. Wystarczy nam zatem struktura danych podobna do kolejki, wspierająca operacje: wstawiania na koniec (*wstaw*), zdejmowania z początku (*zdejmij*) oraz odczytywania maksimum (*max*). Pseudokod używający takiej struktury wygląda następująco[2]:

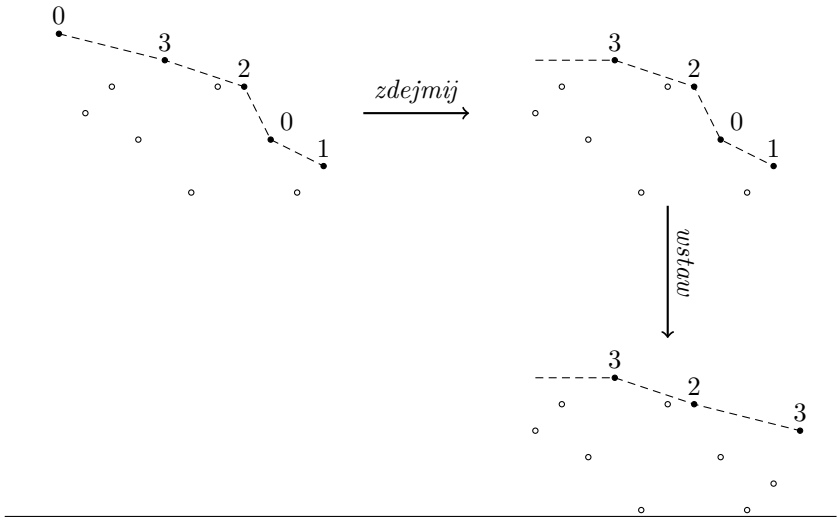
```
1:  i := 1;
2:  for j := 1 to n do begin
3:    while i ⩽ n and Q.max() ⩽ y[i] do begin
4:      Q.wstaw(x[i]);
5:      i := i + 1;
6:    end
7:    wynik := max(wynik, i − j);
8:    Q.zdejmij();
9:  end
10: return wynik;
```

Jeśli udałoby nam się wykonywać operacje dodawania elementu, usuwania najdawniej wstawionego elementu i znajdowania maksimum w zamortyzowanym czasie stałym, uzyskalibyśmy algorytm liniowy. Taka struktura danych wystąpiła już w rozwiązaniach zadań olimpijskich: w zadaniu *Piloci* z III etapu XVII Olimpiady Informatycznej [17] oraz w zadaniu *BBB* z II etapu XV Olimpiady [15]. Aby zachować kompletność opisu rozwiązania wzorcowego, omawiamy ją także poniżej.

Dla ustalenia uwagi przypuśćmy, że wstawiane elementy są parami różne, można to osiągnąć np. przyjmując, że spośród dwóch równych elementów większy jest ten wstawiony później. Zastanówmy się, jakie elementy kolejki mają szansę stać się kiedykolwiek maksimum. Oczywiście, jeśli w kolejce jest jakaś liczba, a potem wstawiona została większa liczba, ta pierwsza nigdy już nie będzie maksimum. Co więcej, nie będziemy potrzebować jej wartości, ponieważ przy operacji *zdejmij* nie interesuje nas wartość, którą usuwamy. Liczbę zawartą w kolejce nazwiemy *ciekawą*, jeśli żadna później wstawiona do kolejki liczba nie jest od niej większa. Liczby ciekawe tworzą ciąg malejący, który łatwo aktualizować przy wstawianiu. Maksimum ($Q.max()$) jest, oczywiście, pierwszy element kolejki. Problematyczne pozostaje usuwanie, czasem bowiem usuwamy liczby ciekawe, a czasem nieciekawe. Wystarczy jednak zapamiętać,

[1] Krótki opis pierwszych dwóch wspomnianych metod wraz z odnośnikami można znaleźć w opracowaniu zadania *Piorunochron* w tej książeczce. Opis trzeciej metody można znaleźć w dwuczęściowym artykule poświęconym problemom RMQ i LCA, w numerach 9/2007 i 11/2007 czasopisma *Delta* (artykuły dostępne także na stronie `http://www.deltami.edu.pl`).

[2] Zakładamy, że dla pustej kolejki mamy $Q.max() = -\infty$.



Rys. 1: Ilustracja działania struktury: pełne kółka reprezentują elementy kolejki D, puste to elementy nieciekawe, współrzędne kółka to odpowiednio indeks w ciągu i wartość, a liczba przy kółku to liczba nieciekawych elementów w ciągu po poprzednim ciekawym (albo od początku ciągu, jeśli rozważamy pierwszy ciekawy element).

ile jest nieciekawych liczb przed maksimum i między każdymi dwiema sąsiednimi ciekawymi.

Taką strukturę łatwo zapisać w terminologii kolejki dwustronnej par liczb całkowitych, np. kontenera `deque` z biblioteki STL w C++. Załóżymy, że mamy do dyspozycji taką kolejkę dwustronną D z operacjami *pusta*(), *pierwszy*(), *ostatni*(), *zdejmij_pierwszy*(), *zdejmij_ostatni*(), *wstaw_ostatni*(x, y). Do elementów pary będziemy odwoływać się przez *.wartość* oraz *.ile*.

Wstawienie elementu. Operacja wstawiania działa według schematu: dopóki ostatni element kolejki nie przekracza aktualnie dodawanego, usuwaj go. Następnie wstaw aktualny element na początek kolejki.

```
1: procedure wstaw(wartość)
2: begin
3:   ile := 0;
4:   while (not D.pusta()) and D.ostatni().wartość ⩽ wartość do begin
5:     ile := ile + D.ostatni().ile + 1;
6:     D.zdejmij_ostatni();
7:   end
8:   D.wstaw_ostatni(wartość, ile);
9: end
```

Koszt zamortyzowany tej operacji jest stały. Każdy element wyjściowego ciągu zostaje w algorytmie wstawiony dokładnie raz, a łączna liczba usunięć jest nie większa niż liczba wstawień.

Usunięcie elementu. Sprawdzamy pierwszy element kolejki — jeśli jest ciekawy, usuwamy go, jeśli nie, zmniejszamy odpowiedni licznik.

```
1: procedure zdejmij()
2: begin
3:   if D.pierwszy().ile = 0 then
4:     D.zdejmij_pierwszy()
5:   else
6:     D.pierwszy().ile := D.pierwszy().ile − 1;
7: end
```

Usuwanie odbywa się w czasie stałym.

Sprawdzenie maksimum. Zwracamy wartość pierwszego elementu.

```
1: function max()
2: begin
3:   if D.pusta() then
4:     return −∞;
5:   return D.pierwszy().wartość;
6: end
```

Znów ewidentnie mamy czas stały operacji.

To kończy opis rozwiązania wzorcowego. Implementacje znajdują się w plikach `tem.cpp`, `tem1.pas` oraz `tem5.cpp`.

Utrudnione zadanie

Podczas zawodów część zawodników błędnie zrozumiała treść zadania — zamiast szukać spójnego przedziału, szukali niespójnego podciągu. Zawodnicy mogli zasugerować się znanym problemem znajdywania najdłuższego podciągu rosnącego i tym samym utrudnili sobie zadanie. Rozwiązanie takiej wersji pozostawiamy Czytelnikowi jako ćwiczenie.

Testy

Testy zostały podzielone na trzy główne grupy:

- *mały losowy* — małe losowe testy poprawnościowe,
- *duży wynik* — wynikiem jest długi przedział czasu,
- *duży stos* — w strukturze z rozwiązania wzorcowego znajduje się równocześnie wiele elementów.

W testach oznaczonych jako *małe temperatury* temperatury mieszczą się w przedziale od -50 do 50.

Nazwa	n	Opis
tem1a.in	92	mały test losowy (małe temperatury)
tem1b.in	93	mały test losowy (małe temperatury)
tem1c.in	94	mały test losowy (małe temperatury)
tem2a.in	100	mały test losowy (małe temperatury)
tem2b.in	98	duży wynik (małe temperatury)
tem2c.in	82	duży wynik (małe temperatury)
tem2d.in	102	duży stos (małe temperatury)
tem3a.in	10 000	duży wynik (małe temperatury)
tem3b.in	12 101	duży wynik (małe temperatury)
tem3c.in	13 221	duży stos (małe temperatury)
tem4a.in	458 325	duży wynik (małe temperatury)
tem4b.in	522 312	duży wynik (małe temperatury)
tem4c.in	593 721	duży stos (małe temperatury)
tem5a.in	724 855	duży wynik
tem5b.in	785 775	duży wynik
tem5c.in	793 722	duży stos
tem6a.in	763 538	duży wynik
tem6b.in	812 289	duży wynik
tem6c.in	813 783	duży stos
tem7a.in	891 379	duży wynik
tem7b.in	832 280	duży wynik
tem7c.in	999 999	duży stos
tem8a.in	813 780	duży stos
tem8b.in	828 183	duży stos
tem8c.in	812 289	duży wynik
tem9a.in	813 780	duży stos (małe temperatury)
tem9b.in	828 183	duży stos (małe temperatury)
tem9c.in	812 289	duży wynik (małe temperatury)
tem10a.in	765 324	duży wynik (małe temperatury)
tem10b.in	850 626	duży stos (małe temperatury)
tem11a.in	713 964	duży wynik
tem11b.in	956 352	duży stos

Nazwa	n	Opis
tem12a.in	687 555	duży wynik
tem12b.in	854 694	duży stos

Zawody III stopnia

opracowania zadań