Jakub Wojtaszczyk
Treść zadania, Opracowanie

Bartosz Górski
Program

Dostępna pamięć: 32 MB.

OI, Etap II, dzień drugi, 12.02.2009

Architekci

Król Bajtazar postanowił wybudować sobie nowy pałac. Ogłosił więc konkurs na najlepszy projekt architektoniczny pałacu. Chcąc zmotywować architektów do spiesznej pracy, ogłosił też, że projekty będzie rozpatrywał w takiej kolejności, w jakiej będą nadsyłane.

Z tym zleceniem wiąże się ogromny prestiż, dlatego też architekci z całego świata nadsyłają swoje propozycje do królewskiej kancelarii. Projektów nadchodzi bardzo dużo, a Bajtazar nie ma czasu ich wszystkich przeglądać. Zrezygnował zatem z samodzielnego wykonywania tej czynności i poprosił swojego kanclerza o to, by wstępnie przejrzał nadchodzące projekty zgodnie z następującymi zasadami:

- *kanclerz ma wybrać k projektów, odrzucając resztę od razu — Bajtazar wie, że więcej niż k projektów i tak nie będzie w stanie przejrzeć;*
- *projekty mają zostać przedstawione Bajtazarowi w takiej kolejności, w jakiej zostały nadesłane — w takiej też kolejności Bajtazar będzie je przeglądał, zgodnie z tym co ogłosił;*
- *spośród wszystkich ciągów k projektów spełniających powyższe warunki, kanclerz ma wybrać ciąg* **najlepszy**, *zgodnie z poniższą definicją.*

  Powiemy, że ciąg projektów $(p_1, p_2, \ldots, p_k)$ jest **lepszy** *od ciągu $(r_1, r_2, \ldots, r_k)$, jeśli dla pewnego $l \geqslant 1$ pierwszych $l-1$ projektów w obu ciągach jest równie dobrych, zaś l-ty projekt w ciągu p jest lepszy od l-tego projektu w ciągu r (czyli $p_i = r_i$ dla $i < l$ i $p_l > r_l$).*

Projekty cały czas nadchodzą i nie wiadomo, do kiedy Bajtazar rozkaże je przyjmować. Kanclerz nie chce zostawiać wyboru k projektów na ostatni moment, jednak bardzo boi się popełnić błąd i narazić na gniew króla. Dlatego poprosił Cię o pomoc.
Napisz program, który:

- *pobierze za pomocą dostarczonej biblioteki liczbę k oraz ciąg liczb reprezentujących jakość kolejnych projektów,*
- *wyznaczy najlepszy ciąg k projektów, zgodnie z podanymi zasadami,*
- *zwróci jakości wybranych projektów za pomocą biblioteki.*

Opis użycia biblioteki

Aby użyć biblioteki, należy wpisać w swoim programie:

- *C/C++:* `#include "carclib.h"`
- *Pascal:* `uses parclib;`
- *Java: nic nie trzeba robić, lecz by uruchomić rozwiązanie, należy mieć skompilowaną bibliotekę jarclib (plik* `jarclib.class`*) w tym samym katalogu co program.*

Biblioteka udostępnia trzy procedury, funkcje lub metody statyczne:

- `inicjuj` — *zwraca liczbę całkowitą k ($1 \leqslant k \leqslant 1\,000\,000$), określającą, jak wiele projektów ma zawierać ciąg wynikowy. Powinna być użyta dokładnie raz, na samym początku działania programu.*
  - *C/C++:* `int inicjuj();`
  - *Pascal:* `function inicjuj(): longint;`
  - *Java:* `public static int inicjuj();`, *będąca metodą statyczną klasy* `jarclib`.
- `wczytaj` — *i-te wywołanie zwraca liczbę całkowitą p_i ($1 \leqslant p_i \leqslant 1\,000\,000\,000$) oznaczającą jakość i-tego projektu (im większa liczba, tym lepszy projekt), albo 0, co oznacza, że nie ma już więcej projektów. Liczba projektów nie jest znana przed wczytaniem danych, jednak możesz założyć, że wszystkich projektów jest przynajmniej k, a co najwyżej $15\,000\,000$. Funkcja ta powinna być wywoływana do momentu, aż skończą się projekty, i* **ani razu więcej**.
  - *C/C++:* `int wczytaj();`
  - *Pascal:* `function wczytaj(): longint;`
  - *Java:* `public static int wczytaj();`, *będąca metodą statyczną klasy* `jarclib`.
- `wypisz` — *za pomocą tej procedury/funkcji wypisujesz jakości kolejnych projektów, które kanclerz przedstawi królowi. Powinna być ona użyta dokładnie k razy; w i-tym wywołaniu należy podać jakość i-tego w kolejności projektu. k-te wywołanie tej procedury/funkcji zakończy działanie Twojego programu.*
  - *C/C++:* `void wypisz(int jakoscProjektu);`
  - *Pascal:* `procedure wypisz(jakoscProjektu: longint);`
  - *Java:* `public static void wypisz(int jakoscProjektu);`, *będąca metodą statyczną klasy* `jarclib`.

Twój program nie może otwierać żadnych plików ani używać standardowego wejścia i wyjścia. Rozwiązanie będzie kompilowane wraz z biblioteką za pomocą następujących poleceń:

- *C:* `gcc -O2 -static carclib.c arc.c -lm`
- *C++:* `g++ -O2 -static carclib.c arc.cpp -lm`
- *Java:* `javac arc.java`, *a skompilowany plik biblioteki* `jarclib` — `jarclib.class` — *będzie się znajdował w tym samym katalogu.*
- *Pascal:* `ppc386 -O2 -XS -Xt arc.pas`, *a plik biblioteki* `parclib` *będzie znajdował się w tym samym katalogu.*

W katalogu `/home/zawodnik/rozw/lib` *możesz znaleźć przykładowe pliki bibliotek i przykładowe rozwiązania ilustrujące sposób ich użycia. Przykładowa biblioteka wczytuje scenariusz testowy ze standardowego wejścia w następującym formacie:*

- *Pierwszy wiersz wejścia zawiera jedną dodatnią liczbę całkowitą k.*
- *Kolejne wiersze wejścia zawierają po jednej dodatniej liczbie całkowitej; $(i+1)$-szy wiersz zawiera liczbę p_i, oznaczającą jakość i-tego zgłoszonego projektu.*
- *Ostatni wiersz wejścia zawiera liczbę 0, oznaczającą koniec listy projektów.*

Przykładowa biblioteka wypisuje na standardowe wyjście k wierszy — jakości projektów zgłoszone przez program.

Przykładowy przebieg programu

C/C++	Pascal	Java	Zwracane wartości i wyjaśnienia
`k = inicjuj();`	`k := inicjuj();`	`k = jarclib.inicjuj();`	Od tego momentu $k = 3$.
			Wczytywanie jakości kolejnych projektów.
`wczytaj();`	`wczytaj();`	`jarclib.wczytaj();`	12
`wczytaj();`	`wczytaj();`	`jarclib.wczytaj();`	5
`wczytaj();`	`wczytaj();`	`jarclib.wczytaj();`	8
`wczytaj();`	`wczytaj();`	`jarclib.wczytaj();`	3
`wczytaj();`	`wczytaj();`	`jarclib.wczytaj();`	15
`wczytaj();`	`wczytaj();`	`jarclib.wczytaj();`	8
`wczytaj();`	`wczytaj();`	`jarclib.wczytaj();`	0 — oznacza koniec ciągu projektów
			Wypisujemy rozwiązanie, czyli 3-elementowy ciąg:
`wypisz(12);`	`wypisz(12);`	`jarclib.wypisz(12);`	12
`wypisz(15);`	`wypisz(15);`	`jarclib.wypisz(15);`	15
`wypisz(8);`	`wypisz(8);`	`jarclib.wypisz(8);`	8

Rozwiązanie

Wprowadzenie

Ciąg nazywany w treści zadania „lepszym” najczęściej określa się mianem „leksykograficznie większego”; w niniejszym opracowaniu będziemy używali drugiego z tych terminów.

Wpierw zastanówmy się chwilę, co dokładnie oznacza określenie największy leksykograficznie podciąg długości k. Otóż jeżeli mamy trzy różne podciągi A, B i C i A jest większy leksykograficznie od B, zaś B — większy leksykograficznie od C, to oczekiwalibyśmy, że A jest większy leksykograficznie od C. I faktycznie — A różni się od B na pewnej pozycji l_{AB}, zaś B od C na pewnej pozycji l_{BC}. Jeśli $l_{AB} = l_{BC}$, to ciągi A i C są takie same aż do pozycji l_{AB}, a potem $A_{l_{AB}} > B_{l_{AB}} > C_{l_{AB}}$. Jeśli $l_{AB} > l_{BC}$, to ciągi A i C są takie same aż do l_{BC}, a potem $A_{l_{BC}} = B_{l_{BC}} > C_{l_{BC}}$; analogicznie, jeśli $l_{AB} < l_{BC}$, to $A_{l_{AB}} > B_{l_{AB}} = C_{l_{AB}}$. Czyli, w szczególności, jeśli zaczniemy od dowolnego ciągu k–elementowego, to — tak długo, jak istnieje jakikolwiek leksykograficznie większy — możemy przechodzić do tego większego, i nigdy

nie wrócimy do raz przejrzanego ciągu. Jako że wszystkich podciągów jest skończenie wiele, to w końcu zatrzymamy się i trafimy na ciąg, od którego żaden inny nie jest leksykograficznie większy. Taki ciąg nazwiemy najlepszym albo leksykograficznie największym.

Zauważmy teraz jeszcze, że jeśli dwa ciągi różnią się, to istnieje pierwsza pozycja l, na której się różnią, i porównując elementy na tej pozycji jesteśmy w stanie stwierdzić, który z nich jest leksykograficznie większy. Nie oznacza to, że dany ciąg ma tylko jeden leksykograficznie największy podciąg (bo np. w ciągu samych jedynek każdy podciąg jest leksykograficznie największy), ale oznacza, że każde dwa podciągi leksykograficznie największe są równe (a więc w szczególności to, co mamy wypisać, jest wyznaczone jednoznacznie).

To, że może istnieć więcej niż jeden największy ciąg, nastręcza pewnych trudności przy analizowaniu rozwiązania. Poradzimy sobie z tym problemem, pokazując jeden konkretny największy ciąg i starając się właśnie ten ciąg skonstruować. Załóżmy, że największy ciąg będzie miał wartości $(a_1, a_2, \ldots, a_k)$. Wówczas wybierzemy ten ciąg, który zawiera pierwsze wystąpienie a_1, potem pierwsze wystąpienie a_2 występujące po wybranym już a_1, następnie pierwsze wystąpienie a_3 po wybranym a_2 itd. Taki ciąg otrzymamy, jeżeli ustalimy, że z dwóch równie dobrych projektów za lepszy uznamy ten, który wpłynął wcześniej. Przy takim założeniu największy leksykograficznie ciąg jest już wyznaczony jednoznacznie.

Naszym zadaniem jest znalezienie największego leksykograficznie podciągu długości k w ciągu o nieznanej długości $n \leqslant 15\,000\,000$. Cały ciąg n liczb nie mieści się w pamięci — faktycznie, pojedyncza liczba z przedziału $[1,\ 1\,000\,000\,000]$ musi zająć przynajmniej $\log_2 1\,000\,000\,000 > 29$ bitów, zatem 15 000 000 takich liczb musi zająć ponad 43 MB, a więc istotnie więcej niż dostępne 32 MB. Będziemy zatem musieli w trakcie działania programu w jakiś sposób odrzucać z pamięci wartości, o których wiemy, że już się nam nie przydadzą.

Rozwiązanie wzorcowe

W tym rozwiązaniu będziemy w każdym kroku przechowywać w pamięci wyłącznie największy leksykograficznie ciąg długości k wybrany z dotychczas wczytanych elementów. Żeby to podejście mogło zadziałać, musimy zrozumieć, jak — mając największy leksykograficznie podciąg długości k w ciągu $(x_1, x_2, \ldots, x_m)$ — wybrać największy leksykograficznie podciąg długości k w ciągu $(x_1, x_2, \ldots, x_m, x_{m+1})$. Pomogą nam w tym następujące spostrzeżenia:

Fakt 1. *Niech $(x_1, x_2, \ldots, x_m)$ będzie dowolnym ciągiem liczb. Niech dalej $(x_{i_1}, x_{i_2}, \ldots, x_{i_k})$ będzie największym leksykograficznie podciągiem długości k tego ciągu (oczywiście $k \leqslant m$ oraz $i_1 < i_2 < \ldots < i_k$). Wtedy $x_{i_1} = \max\{x_1, \ldots, x_{m-k+1}\}$ (przypomnijmy, że w przypadku remisu jako i_1 wybieramy pierwsze wystąpienie maksimum), zaś $(x_{i_2}, x_{i_3}, \ldots, x_{i_k})$ jest największym $(k-1)$–elementowym podciągiem ciągu $(x_{i_1+1}, x_{i_1+2}, \ldots, x_m)$.*

Dowód: Skoro $(x_{i_1}, x_{i_2}, \ldots, x_{i_k})$ jest podciągiem $(x_1, x_2, \ldots, x_m)$, to

$$\begin{aligned} i_k &\leqslant m \\ i_{k-1} &\leqslant m-1 \\ &\ldots \\ i_1 &\leqslant m-k+1 \end{aligned}$$

Zatem x_{i_1} musi być jedną z liczb $x_1,\ldots,x_{m-k+1}$. Załóżmy, że $x_{i_1} \neq \max\{x_1,\ldots,x_{m-k+1}\}$. Wówczas $x_{i_1} < \max\{x_1,\ldots,x_{m-k+1}\} \stackrel{\text{def}}{=} x_j$. Ciąg $(x_j,x_{j+1},\ldots,x_{j+k-1})$ jest więc podciągiem $(x_1,x_2,\ldots,x_m)$ (bo $j+k-1 \leqslant m$) oraz jest leksykograficznie większy od ciągu $(x_{i_1},x_{i_2},\ldots,x_{i_k})$ (co otrzymujemy, wstawiając w definicji $l=1$), a zatem ciąg $(x_{i_1},x_{i_2},\ldots,x_{i_k})$, wbrew założeniu, nie mógł być największym leksykograficznie podciągiem długości k.

Drugą część faktu otrzymujemy już teraz wprost z definicji leksykograficznej większości. ■

Fakt 2. *Jeśli* $(x_{i_1},x_{i_2},\ldots,x_{i_k})$ *jest największym leksykograficznie podciągiem długości* k *ciągu* $(x_1,x_2,\ldots,x_m)$*, to największy leksykograficznie podciąg* $(x_{j_1},x_{j_2},\ldots,x_{j_k})$ *długości* k *ciągu* $(x_1,x_2,\ldots,x_m,x_{m+1})$ *jest podciągiem ciągu* $(x_{i_1},x_{i_2},\ldots,x_{i_k},x_{m+1})$.

Dowód: Dowód przebiega przez indukcję ze względu na k. Dla $k=1$ największy leksykograficznie podciąg danego ciągu to po prostu jego największy element — i faktycznie, największy element $(x_1,x_2,\ldots,x_{m+1})$ to albo największy element $(x_1,x_2,\ldots,x_m)$, albo też x_{m+1}.

Załóżmy teraz, że teza faktu zachodzi dla wszystkich ciągów długości $k-1$, i przyjrzyjmy się ciągom długości k. Na mocy Faktu 1 wiemy, że x_{j_1} jest równe największej spośród liczb $x_1,x_2,\ldots,x_{m-k+2}$. Wiemy też, że

$$x_{i_1} = \max\{x_1,x_2,\ldots,x_{m-k+1}\} \quad \text{oraz} \quad x_{i_2} = \max\{x_{i_1+1},x_{i_1+2},\ldots,x_{m-k+2}\}.$$

Zatem $\max\{x_{i_1},x_{i_2}\} = x_{j_1}$.

Jeśli $x_{i_1} = x_{j_1}$, to ciąg $(x_{i_2},x_{i_3},\ldots,x_{i_k})$ jest największym podciągiem długości $k-1$ ciągu $(x_{i_1+1},x_{i_1+2},\ldots,x_m)$, zaś $(x_{j_2},x_{j_3},\ldots,x_{j_k})$ jest największym podciągiem długości $k-1$ ciągu $(x_{i_1+1},x_{i_1+2},\ldots,x_m,x_{m+1})$. Zatem na mocy założenia indukcyjnego $(x_{j_2},x_{j_3},\ldots,x_{j_k})$ jest podciągiem $(x_{i_2},x_{i_3},\ldots,x_{i_k},x_{m+1})$, wobec czego $(x_{j_1},x_{j_2},x_{j_3},\ldots,x_{j_k})$ jest podciągiem $(x_{i_1},x_{i_2},x_{i_3},\ldots,x_{i_k},x_{m+1})$. Jeśli natomiast $x_{j_1} = x_{i_2}$, to $j_1 = i_2 = m-k+2$. To oznacza, że

$$i_w = m-k+w \text{ dla } w=2,3,\ldots,k \quad \text{oraz} \quad j_{w-1} = m-k+w \text{ dla } w=2,3,\ldots,k+1,$$

czyli znowu $(x_{j_1},x_{j_2},x_{j_3},\ldots,x_{j_k})$ jest podciągiem $(x_{i_1},x_{i_2},x_{i_3},\ldots,x_{i_k},x_{m+1})$. ■

Umiemy już zatem ominąć problem braku pamięci — wystarczy po wczytaniu pierwszych m elementów przechowywać w pamięci wyłącznie największy leksykograficznie podciąg długości k ciągu $(x_1,x_2,\ldots,x_m)$ i wczytawszy kolejny element, poprawiać ten ciąg tak, by był największym leksykograficznie podciągiem ciągu $(x_1,x_2,\ldots,x_{m+1})$. Dochodzimy teraz do algorytmicznego problemu efektywnego znajdowania tego największego k–elementowego podciągu spośród $k+1$ elementów.

Ponieważ wszystkich takich podciągów jest $k+1$ (wystarczy wybrać element, który nie należy do naszego podciągu), a porównanie leksykograficzne dwóch podciągów można wykonać w czasie $O(k)$, więc koszt czasowy skrajnie siłowego podejścia to $O(nk^2)$. Wobec ograniczeń na n i k potrzebne jest zastosowanie innego podejścia. Zastanówmy się mianowicie, w jakich sytuacjach usunięcie pierwszego spośród $k+1$ elementów $(x_{i_1},x_{i_2},\ldots,x_{i_k},x_{m+1})$ daje największy leksykograficznie ciąg długości k. Oczywiście usunięcie tego elementu prowadzi do ciągu rozpoczynającego się od x_{i_2}. Jeżeli $x_{i_1} > x_{i_2}$, to widzimy, że w ten sposób nie możemy otrzymać największego ciągu, gdyż na pewno lepszym

kandydatem jest ciąg złożony z k początkowych elementów. Przypadek $x_{i_1} = x_{i_2}$ na chwilę pomińmy. Jeżeli zaś $x_{i_1} < x_{i_2}$, to na pewno usunięcie x_{i_1} prowadzi do ciągu większego niż $(x_{i_1}, x_{i_2}, \ldots, x_{i_k})$. Co więcej, ponieważ każdy k–elementowy podciąg wyjściowego ciągu poza tym z usuniętym x_{i_1} zaczyna się od x_{i_1}, widzimy, że usuwając x_{i_1}, w tym przypadku otrzymujemy na pewno największy leksykograficznie podciąg.

Podobnie sytuacja ma się dla kolejnych elementów: jeżeli i-ty element jest większy od następnego, to na pewno nie opłaca się go usuwać, a jeżeli mniejszy, to na pewno jego usunięcie jest najlepszą możliwą decyzją — oczywiście o ile przypadek ten nie wystąpił dla żadnego z wcześniejszych elementów. Zauważmy dalej, że w pomijanym dotychczas przypadku równości kolejnych elementów tak naprawdę nie musimy rozważać możliwości usunięcia pierwszego z nich, gdyż dokładnie taki sam ciąg otrzymamy, jeżeli usuniemy drugi — to pokazuje, że ten przypadek sprowadza się w końcu do jednego z dwóch pozostałych. Podsumowując: aby uzyskać podciąg największy leksykograficznie, musimy znaleźć najdłuższy fragment ciągu, w którym jest on nierosnący, a następnie usunąć jego ostatni element (zauważmy, że to sformułowanie uwzględnia przypadek, w którym cały wyjściowy ciąg jest nierosnący). W ten sposób możemy zredukować złożoność czasową rozwiązania do $O(nk)$:

```
1:  k := inicjuj();
2:  for i := 1 to k+1 do najwiekszy[i] := wczytaj();
3:  while najwiekszy[k+1] > 0 do
4:  begin
5:    opuszczany := 1;
6:    while opuszczany ⩽ k and
7:          najwiekszy[opuszczany] ⩾ najwiekszy[opuszczany+1] do
8:      opuszczany := opuszczany+1;
9:    for i := opuszczany to k do najwiekszy[i] := najwiekszy[i+1];
10:   najwiekszy[k+1] := wczytaj();
11: end
12: for i := 1 to k do wypisz(najwiekszy[i]);
```

Ale teraz możemy jeszcze zauważyć, że w powyższym rozwiązaniu po wielokroć przeglądamy i porównujemy te same elementy — a przecież jeżeli już przejrzeliśmy pewien początkowy odcinek ciągu i stwierdziliśmy, że jest on nierosnący, to możemy zrezygnować z przeglądania go ponownie. Musimy cofnąć się o jeden element ciągu (bo jeśli skasowaliśmy element na pozycji j, to teraz po pozycji $j-1$ następuje pozycja $j+1$, a pary $j-1, j+1$ jeszcze nie porównywaliśmy), natomiast wszystkie poprzednie pary już są sprawdzone.

```
1:  k := inicjuj();
2:  for i := 1 to k+1 do najwiekszy[i] := wczytaj();
3:  opuszczany := 1;
4:  while najwiekszy[k+1] > 0 do
5:  begin
6:    while opuszczany ⩽ k and
7:          najwiekszy[opuszczany] ⩾ najwiekszy[opuszczany+1] do
8:      opuszczany := opuszczany+1;
9:    for i := opuszczany to k do najwiekszy[i] := najwiekszy[i+1];
```

```
10:    if opuszczany ⩾ 1 then opuszczany := opuszczany − 1;
11:    najwiekszy[k+1] := wczytaj();
12: end
13: for i := 1 to k do wypisz(najwiekszy[i]);
```

Teraz już wyszukiwanie odbywa się w amortyzowanej złożoności czasowej $O(1)$. Dla każdego wczytanego elementu ciągu liczba *opuszczany* maleje o co najwyżej jeden, jej początkowa wartość to 1, końcowa to co najwyżej $k+1$, zatem może w trakcie działania całego programu wzrosnąć co najwyżej $n+k$ razy. Niestety, w powyższym programie mamy jeszcze przepisywanie ciągu (pętla **for** w linii 9), które może pesymistycznie wymagać $O(k)$ operacji przy każdym wczytanym elemencie. Ten problem jednak łatwo rozwiązać — wystarczy zamiast trzymać ciąg w tablicy, użyć do tego listy. W poniższym pseudokodzie korzystamy z dwukierunkowej listy *lista*, która umożliwia wykonywanie w stałej złożoności czasowej operacji: efektywnego dostępu do pierwszego i ostatniego elementu, przechodzenia do poprzedniego i następnego elementu listy, wstawiania i usuwania elementów i pobierania wartości elementu.

```
1: k := inicjuj();
2: for i := 1 to k+1 do lista.wstaw_na_koniec(wczytaj());
3: opuszczany := lista.poczatek();
4: while lista.koniec().wartosc > 0 do
5: begin
6:    while opuszczany ≠ lista.koniec() and
7:          opuszczany.wartosc ⩾ opuszczany.nastepny().wartosc do
8:       opuszczany := opuszczany.nastepny();
9:    if opuszczany ≠ lista.poczatek() then begin
10:      opuszczany := opuszczany.poprzedni();
11:      lista.usun(opuszczany.nastepny());
12:   end
13:   else begin
14:      opuszczany := opuszczany.nastepny();
15:      lista.usun(opuszczany.poprzedni());
16:   end
17:   lista.wstaw_na_koniec(wczytaj());
18: end
19: wsk := lista.poczatek();
20: while wsk ≠ lista.koniec() do
21: begin
22:    wypisz(wsk.wartosc);
23:    wsk := wsk.nastepny();
24: end
```

Rozwiązanie to zostało zaimplementowane w plikach `arc10.cpp` (samodzielna implementacja listy na tablicy), `arc.cpp` (lista z biblioteki STL), `arc1.c` i `arc2.pas` (samodzielne implementacje listy dynamicznej).

Rozwiązanie alternatywne

Zastanówmy się wpierw, jak rozwiązywalibyśmy nasze zadanie, gdybyśmy byli w stanie wczytać cały ciąg do pamięci. W tym rozwiązaniu również zakładamy, że spośród dwóch równie dobrych projektów lepszy jest ten, który pojawił się wcześniej.

Ponownie zacznijmy od Faktu 1. Wiemy, że x_{i_1} to największa spośród pierwszych $n-k+1$ liczb. Liczba x_{i_2} to największy element występujący po x_{i_1} spośród pierwszych $n-k+2$ liczb, i analogicznie x_{i_w} to zawsze największy element występujący po $x_{i_{w-1}}$ i nie dalej niż na pozycji $n-k+w$.

Aby wyznaczyć taki ciąg, wykorzystamy stos. Zakładamy, że stos jest początkowo pusty oraz że umożliwia wykonywanie każdej z następujących operacji w czasie $O(1)$: wstawienie elementu na szczyt stosu, zdjęcie elementu ze szczytu, sprawdzenie wartości elementu na szczycie i obliczenie rozmiaru struktury, czyli liczby zawartych w niej elementów.

```
1:  i := 1;
2:  while n+1-i+stos.rozmiar() > k do
3:    if stos.pusty() or stos.wierzch >= X[i] then
4:    begin
5:      if stos.rozmiar() < k then stos.wstaw(X[i]);
6:      i := i+1;
7:    end
8:    else stos.zdejmij_z_gory();
9:  while i <= n do
10: begin
11:   stos.wstaw(X[i]);
12:   i := i+1;
13: end
```

Powyższy pseudokod może wydawać się na pierwszy rzut oka nieco tajemniczy. W dalszej części tekstu pokażemy kolejno, że ma on własność stopu, czyli że w końcu zatrzymuje się, oraz że jest poprawny, czyli że ciąg uzyskany na stosie spełnia wymienione wcześniej warunki.

W uzasadnieniu własności stopu wystarczy skupić się na pętli **while** z linii 2. Zauważmy przede wszystkim, że liczba wykonań instrukcji z linii 8, odpowiadającej sytuacji, w której warunek z linii 3 nie zachodzi, jest nie większa niż liczba wykonań instrukcji z linii 5–6. Faktycznie, każdy element zdejmowany ze stosu musi wcześniej się tam znaleźć. To pokazuje, że w analizie wystarczy skupić się na instrukcjach z linii 5–6. Za każdym razem, gdy są one wykonywane, wartość zmiennej i wzrasta o 1. Wystarczy teraz zauważyć, że wartość i nie może nigdy przekroczyć $n+1$, co wynika z warunku pętli **while**, wobec faktu, że rozmiar stosu nie może przekroczyć k (patrz warunek w linii 5). Pokazaliśmy zatem, że łączna liczba obrotów pierwszej pętli **while** jest rzędu $O(n)$; analogiczne stwierdzenie w przypadku drugiej z pętli jest oczywiste.

Przejdźmy teraz do uzasadnienia poprawności wyniku. Po pierwsze, jak łatwo sprawdzić, w każdym obrocie pierwszej pętli **while** liczba $n+1-i+stos.rozmiar()$ albo nie zmienia się, albo maleje o jeden, zatem w momencie wyjścia z pętli będzie zachodzić

$n+1-i+stos.rozmiar() = k$. Wobec tego po wstawieniu pozostałych $n+1-i$ elementów na stos w drugiej pętli **while** faktycznie otrzymamy ciąg długości k.

Po drugie, kolejność elementów na stosie odpowiada kolejności elementów w pierwotnym ciągu, bo wstawiamy na stos w kolejności przeglądania.

Po trzecie, udowodnijmy, że otrzymany ciąg to faktycznie ten największy leksykograficznie. Zauważmy, że podczas wykonywania pierwszej pętli elementy na stosie tworzą ciąg posortowany (najmniejszy element znajduje się na szczycie stosu). Rozważmy największy element ciągu $x_1, x_2, \ldots, x_{n-k+1}$. Gdy go wczytamy, to zdejmiemy ze stosu wszystkie wcześniejsze elementy, a później już nigdy go nie zdejmiemy (elementy o numerach do $n-k+1$ są nie większe, więc go nie zdejmą, zaś elementy po $n-k+1$ nie zdejmą go, gdyż stos byłby zbyt mały). Wobec tego drugi element stosu będzie elementem występującym po x_{i_1}. Oczywiście będzie występował nie dalej niż na pozycji $n-k+2$ (bo po nim musi wystąpić jeszcze $k-2$ elementów). I będzie największym z elementów na rozpatrywanych pozycjach — znowu, kiedy ten największy element nadejdzie, to zdejmie wszystkie elementy poza pierwszym ze stosu, a następnie sam nie zostanie zdjęty. I dalej analogicznie — w–ty element stosu będzie największym elementem na pozycjach od $i_{w-1}+1$ do $n-k+w$, bo gdy największy element z tych pozycji nadejdzie, to zdejmie wszystkie elementy ze stosu poza pierwszymi $w-1$, a potem sam nie zostanie zdjęty. Zatem faktycznie otrzymany na stosie ciąg będzie największym leksykograficznie.

Niestety — w naszym zadaniu nie możemy wczytać całego ciągu do pamięci, w szczególności zaś nie znamy liczby n. Zauważmy jednak, że liczbę tę wykorzystujemy wyłącznie do sprawdzenia warunku $n+1-i+stos.rozmiar() > k$. Zatem, w szczególności, jeśli $n-i > k$, to nie musimy znać dokładnej wartości n, gdyż wiemy, że ten warunek jest spełniony. Możemy zatem nie wczytywać całego ciągu na raz, a jedynie czytać „o k elementów do przodu", by upewnić się, czy jest jeszcze wystarczająco wiele elementów ciągu przed nami — a jeżeli nie, to możemy już śmiało stosować algorytm z poprzedniego pseudokodu. Do implementacji opisanego podejście możemy zastosować tzw. *bufor* będący np. kolejką cykliczną implementowaną za pomocą tablicy k–elementowej:

```
1:  { Operacje wykonywane na samym początku. }
2:  for i := 1 to k do bufor[i] := wczytaj();
3:  n := k;
4:  pozycja := 1;
5:  koniec := false;
6:
7:  { Funkcja implementująca dostęp do kolejnego elementu ciągu wejściowego. }
8:  function wczytaj_przez_bufor() : int
9:  begin
10:   elem := bufor[pozycja];
11:   if not koniec then
12:   begin
13:     bufor[pozycja] := wczytaj();
14:     if bufor[pozycja] = 0
15:       then koniec := true;
16:       else n := n+1;
17:   end
18:   pozycja := pozycja+1;
```

```
19:   if pozycja = k+1 then pozycja := 1;
20:   return elem;
21: end
```

Rozwiązanie to zostało zaimplementowane w `arc3.cpp`, `arc4.pas`, działa w złożoności czasowej $O(n+k)$ i pamięciowej $O(k)$.

Testy

Zadanie było sprawdzane na 10 zestawach danych testowych, po trzy testy w każdym zestawie. Testy „a" to testy tworzone przy pomocy generatora liczb pseudolosowych, które w tym zadaniu pełnią też rolę testów poprawnościowych. Testy „b" to testy, które zawierają długi podciąg niemalejący, a testy „c" zawierają długi podciąg nierosnący — te grupy pełniły rolę testów wydajnościowych. Wielkości testów zostały tak dobrane, aby rozwiązania, które działają w złożoności pamięciowej $O(n)$ lub większej, nie dostawały więcej niż 50 punktów, niezależnie od złożoności obliczeniowej.

Nazwa	n	k	Opis
arc1a.in	30	23	test losowy
arc1b.in	14	6	test zawierający długi podciąg niemalejący
arc1c.in	16	9	test zawierający długi podciąg nierosnący
arc2a.in	14045	547	test losowy
arc2b.in	16456	769	test zawierający długi podciąg niemalejący
arc2c.in	28683	1984	test zawierający długi podciąg nierosnący
arc3a.in	154632	6428	test losowy
arc3b.in	276543	18632	test zawierający długi podciąg niemalejący
arc3c.in	597843	29567	test zawierający długi podciąg nierosnący
arc4a.in	1000000	12343	test losowy
arc4b.in	1500000	28754	test zawierający długi podciąg niemalejący
arc4c.in	2000000	32462	test zawierający długi podciąg nierosnący
arc5a.in	4000000	53421	test losowy
arc5b.in	5000000	86223	test zawierający długi podciąg niemalejący
arc5c.in	6000000	83456	test zawierający długi podciąg nierosnący
arc6a.in	8000000	76243	test losowy
arc6b.in	9000000	74437	test zawierający długi podciąg niemalejący
arc6c.in	10000000	138643	test zawierający długi podciąg nierosnący
arc7a.in	11000000	1976	test losowy
arc7b.in	11500000	18653	test zawierający długi podciąg niemalejący

Nazwa	n	k	Opis
arc7c.in	12 000 000	23 442	test zawierający długi podciąg nierosnący
arc8a.in	12 000 000	32 124	test losowy
arc8b.in	12 500 000	87 234	test zawierający długi podciąg niemalejący
arc8c.in	13 000 000	123 676	test zawierający długi podciąg nierosnący
arc9a.in	13 000 000	213 534	test losowy
arc9b.in	13 500 000	431 223	test zawierający długi podciąg niemalejący
arc9c.in	14 000 000	744 456	test zawierający długi podciąg nierosnący
arc10a.in	14 500 000	989 974	test losowy
arc10b.in	15 000 000	989 563	test zawierający długi podciąg niemalejący
arc10c.in	15 000 000	1 000 000	test zawierający długi podciąg nierosnący