Jakub Radoszewski
Treść zadania, Opracowanie

Bartosz Górski
Program

Dostępna pamięć: 64 MB.

OI, Etap I, 19.10–16.11.2009

Test na inteligencję

Jedno z zadań w bajtockim teście na inteligencję polega na wykreślaniu liczb z zadanego początkowego ciągu tak, aby otrzymywać w ten sposób różne inne zadane sekwencje. Bajtazar chciałby zostać bajtockim mistrzem IQ, ale wyjątkowo kiepsko radzi sobie z zadaniami tego typu. Zamierza dużo ćwiczyć i poprosił Cię o napisanie programu, który pomoże mu szybko sprawdzać odpowiedzi.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą m ($1 \leqslant m \leqslant 1\,000\,000$). Drugi wiersz zawiera ciąg m liczb całkowitych $a_1, a_2, \ldots, a_m$ ($1 \leqslant a_i \leqslant 1\,000\,000$ dla $1 \leqslant i \leqslant m$) pooddzielanych pojedynczymi odstępami, tworzących początkowy ciąg w zadaniu z testu. Trzeci wiersz wejścia zawiera jedną dodatnią liczbę całkowitą n. Kolejne $2n$ wierszy zawiera opisy ciągów, które mają powstać w wyniku wykreślania różnych liczb z początkowego ciągu. Opis każdego z tych ciągów zajmuje po dwa kolejne wiersze. W pierwszym wierszu każdego opisu znajduje się liczba całkowita m_i ($1 \leqslant m_i \leqslant 1\,000\,000$). Drugi wiersz zawiera m_i-elementowy ciąg liczb całkowitych $b_{i,1}, b_{i,2}, \ldots, b_{i,m_i}$ ($1 \leqslant b_{i,j} \leqslant 1\,000\,000$ dla $1 \leqslant j \leqslant m_i$) pooddzielanych pojedynczymi odstępami. Możesz założyć, że suma długości podanych n sekwencji nie przekroczy 1 000 000.

Wyjście

Twój program powinien wypisać na standardowe wyjście n wierszy. Wiersz o numerze i (dla $1 \leqslant i \leqslant n$) powinien zawierać jedno słowo „`TAK`" lub „`NIE`" (bez cudzysłowów), w zależności od tego, czy i-ta sekwencja z wejścia może powstać w wyniku wykreślenia (tj. usunięcia) pewnych (niekoniecznie kolejnych) liczb z początkowego ciągu. Oczywiście, kolejność liczb w powstałym po wykreśleniach ciągu ma znaczenie (patrz przykład).

Przykład

Dla danych wejściowych:

```
7
1 5 4 5 7 8 6
4
5
1 5 5 8 6
3
2 2 2
3
5 7 8
4
```

```
1 5 7 4
```
poprawnym wynikiem jest:
```
TAK
NIE
TAK
NIE
```

Rozwiązanie

Analiza problemu

Zacznijmy od sformułowania właściwego problemu, który mamy do rozwiązania. Dany jest ciąg a o długości m oraz n ciągów b_i o długościach m_i, wszystkie indeksowane od jedynki. Ograniczenia z zadania sugerują wprowadzenie dodatkowych parametrów:

- $len = m_1 + m_2 + \ldots + m_n$, czyli suma długości wszystkich sekwencji b_i,
- s — maksimum wartości elementów wszystkich ciągów,
- $S = \{1, 2, \ldots, s\}$.

Treść zadania gwarantuje, że $m, n, len, s \leqslant 10^6$ oraz że elementy wszystkich ciągów należą do zbioru S.

Zdefiniowana w zadaniu operacja wykreślania elementów z ciągu a sprowadza się do tego, że chcemy dla każdego ciągu b_i sprawdzić, czy jest on *podciągiem* ciągu a. Przypomnijmy, że ciąg $(c_i)_{i=1}^{p}$ nazywamy podciągiem ciągu $(d_j)_{j=1}^{q}$, jeśli można wybrać takie indeksy $j_1, j_2, \ldots, j_p$, że:

$$1 \leqslant j_1 < j_2 < \ldots < j_p \leqslant q$$

oraz:

$$c_1 = d_{j_1}, \quad c_2 = d_{j_2}, \quad \ldots, \quad c_p = d_{j_p}.$$

Zanim zajmiemy się konstruowaniem wymyślnych algorytmów, warto zacząć od najprostszego możliwego rozwiązania, w którym dla każdego kolejnego ciągu b_i w bezpośredni sposób sprawdzamy, czy jest on podciągiem ciągu a. Każde takie sprawdzenie zaczynamy od znalezienia w ciągu a pierwszego wystąpienia elementu $b_{i,1}$, następnie w dalszej części ciągu a poszukujemy pierwszego wystąpienia elementu $b_{i,2}$ itd. Poniższy pseudokod zawiera implementację takiego podejścia.

```
1: function podciąg(a, m, b_i, m_i)
2: begin
3:   k := 1; { pozycja w ciągu b_i }
4:   for j := 1 to m do
5:     if (k ⩽ m_i) and (a_j = b_{i,k}) then
6:       k := k + 1;
7:   if k > m_i then return TAK else return NIE;
8: end
```

Łączna liczba operacji wykonywanych w funkcji *podciąg* jest rzędu $O(m)$. Wywołując tę funkcję dla każdego kolejnego ciągu b_i, otrzymujemy zatem rozwiązanie o złożoności czasowej $O(n \cdot m)$, co wobec ograniczeń z zadania, jeszcze nas nie satysfakcjonuje.

Powyższa funkcja jest zapisana niezbyt efektywnie; można ją próbować usprawniać, korzystając z rozmaitych spostrzeżeń:

- jeśli $m_i > m$, to można od razu zwrócić `NIE`;
- jeśli w pewnym momencie wykonywania pętli **for** zachodzi $k > m_i$, to możemy od razu przerwać wykonywanie pętli i zwrócić `TAK`;
- (bardziej pomysłowe:) możemy na wstępie sprawdzać, czy w ogóle multizbiór (czyli zbiór z powtórzeniami) elementów ciągu b_i jest podzbiorem multizbioru elementów ciągu a, a jeśli nie, to od razu zwracać odpowiedź `NIE` (implementację tego usprawnienia w dodatkowym, łącznym koszcie czasowym $O(m + s + len)$ pozostawiamy Czytelnikowi).

Niestety, żadne z powyższych ulepszeń nie poprawia złożoności czasowej naszego rozwiązania, o czym można przekonać się, biorąc pod uwagę np. ciągi $a = (1, 1, \ldots, 1, 2)$, $b_i = (2)$. Tego typu rozwiązania zdobywały na zawodach 20-30% punktów. Stosowne implementacje można znaleźć w plikach `tess4.cpp` oraz `tess5.pas`.

Pierwsze rozwiązanie wzorcowe

Aby efektywniej sprawdzać, czy dany ciąg b_i jest podciągiem ciągu a, posłużymy się pomocniczą strukturą danych, którą zbudujemy raz na samym początku rozwiązania. Naszym celem jest zredukowanie kosztu czasowego pojedynczego sprawdzenia z $O(m)$ do kosztu zależnego od m_i — wówczas suma kosztów wszystkich sprawdzeń będzie zależała już nie od $n \cdot m$, ale od parametru $len = m_1 + m_2 + \ldots + m_n$.

Zauważmy, że na funkcję *podciąg* możemy spojrzeć jak na m_i-krotne wykonanie operacji: „znajdź pierwszy element w ciągu a położony za a_j i równy $b_{i,k}$" — jeśli któraś z tych m_i operacji nie powiedzie się, możemy od razu zwrócić odpowiedź `NIE`. Skorzystajmy z tego, że rozmiar zbioru S jest nieduży, i dla każdego elementu $c \in S$ zapiszmy indeksy jego kolejnych wystąpień w ciągu a. Oznaczmy taki (posortowany) ciąg przez ℓ_c. Wówczas żądaną operację możemy wykonać, znajdując w $\ell_{b_{i,k}}$ pierwszy element większy niż j. Jeśli dodatkowo struktury ℓ_c będą zorganizowane jak tablice, tzn. będą dopuszczały swobodny dostęp do poszczególnych elementów (czyli dostęp w czasie stałym), to ów indeks można będzie wyznaczyć efektywnie za pomocą wyszukiwania binarnego.

Przykład 1. W przypadku ciągu $a = (2, 1, 4, 2, 1, 5, 4, 1, 2)$ i zbioru $S = \{1, 2, 3, 4, 5\}$ ciągi ℓ_c mają postać: $\ell_1 = (2, 5, 8)$, $\ell_2 = (1, 4, 9)$, $\ell_3 = ()$, $\ell_4 = (3, 7)$, $\ell_5 = (6)$.

Spróbujmy zapisać pseudokod powyższego rozwiązania, pomijając na razie to, jak konkretnie reprezentujemy strukturę danych ℓ_c oraz jak ją tworzymy — załóżmy tylko, że kolejne elementy ciągu ℓ_c to $\ell_c[1], \ell_c[2], \ldots, \ell_c[r]$, przy czym $r = size(\ell_c)$ to długość tego ciągu.

```
1:  { Wyszukiwanie binarne — funkcja zwraca pierwszy element ℓc[i] ciągu }
2:  { rosnącego ℓc większy niż j lub BRAK, jeśli takiego elementu nie ma. }
3:  function pierwszy_większy(ℓc, j)
4:  begin
5:    lewy := 1; prawy := size(ℓc);
6:    while lewy < prawy do begin
7:      śr := (lewy + prawy) div 2;
8:      if ℓc[śr] ⩽ j then lewy := śr + 1
9:      else prawy := śr;
10:   end
11:   if ℓc[lewy] > j then return ℓc[lewy]
12:   else return BRAK;
13: end
14:
15: function podciąg2(bi, mi)
16: begin
17:   j := 0;
18:   for k := 1 to mi do begin
19:     j := pierwszy_większy(ℓb_{i,k}, j);
20:     if j = BRAK then return NIE;
21:   end
22:   return TAK;
23: end
```

Koszt czasowy funkcji *pierwszy_większy* to $O\big(\log(size(\ell_c))\big) = O(\log m)$, oczywiście o ile parametr ℓ_c nie jest przekazywany bezpośrednio, lecz przez wskaźnik bądź referencję. Stąd złożoność czasowa funkcji *podciąg2* to $O(m_i \log m)$. Pozostaje pytanie, w jaki sposób reprezentować ciągi ℓ_c.

Programujący w języku C++ mogą do przechowywania struktury ℓ użyć tablicy vectorów; `vector` to kontener z biblioteki STL, który zachowuje się dokładnie jak tablica, tyle że dodatkowo umożliwia dodawanie nowych elementów na koniec[1]. Zakładając jednak, że nie mamy takich udogodnień do dyspozycji, możemy upakować całą strukturę ℓ do jednej, dużej tablicy $t[1..m]$: na początku umieścimy w niej ℓ_1, potem ℓ_2, itd. aż do ℓ_s. Konstrukcję tablicy t wykonujemy za pomocą poniższego pseudokodu.

```
1: Algorytm wypełniania tablicy t[1..m]:
2:   for i := 1 to s do count[i] := 0;
3:   for j := 1 to m do count[aj] := count[aj] + 1;
4:   pocz[1] := 1;
5:   for i := 2 to s do pocz[i] := pocz[i − 1] + count[i − 1];
6:   for i := 1 to s do kon[i] := pocz[i] − 1;
7:   for j := 1 to m do begin
8:     kon[aj] := kon[aj] + 1;
```

[1] W STL-u można także znaleźć funkcję działającą podobnie jak nasza *pierwszy_większy*, a mianowicie `upper_bound`.

```
 9:      t[kon[a_j]] := j;
10:  end
```

W powyższym algorytmie, w wierszach 2-3 zliczamy wystąpienia poszczególnych elementów w ciągu a i zapamiętujemy wyniki w tablicy $count[1..s]$. W dwóch kolejnych wierszach na podstawie tych wartości wypełniamy tablicę *pocz* indeksów początków tablic $\ell_1, \ell_2, \ldots, \ell_s$ w ramach t. Wreszcie w wierszach 6-10 wypełniamy tablicę t, przy okazji obliczając końce wystąpień poszczególnych tablic ℓ_i w ramach t (tablica *kon*) — po zakończeniu tej procedury mamy $\ell_i = t[pocz[i]..kon[i]]$. (Przy czym przedział postaci $[x..x-1]$ reprezentuje ciąg pusty). Warto jeszcze wspomnieć, że powyższy algorytm działa bardzo podobnie do sortowania kubełkowego (patrz np. książka [21]).

Nie da się ukryć, że koszt czasowy wypełnienia tablicy t to $O(m+s)$. Po tych wstępnych obliczeniach wyszukiwanie binarne w ciągu ℓ_c można wykonywać na fragmencie tablicy t od $pocz[c]$ do $kon[c]$. W ten sposób złożoność czasowa funkcji *podciąg2* pozostaje zgodna z wcześniejszymi przewidywaniami, tzn. $O(m_i \log m)$ w jednym wywołaniu, a łącznie $O(len \cdot \log m)$. Ostateczna złożoność czasowa tego rozwiązania to zatem $O(len \cdot \log m + m + s)$. Łatwo sprawdzić, że jego złożoność pamięciowa szacuje się przez $O(len + m + s)$.

Implementacje rozwiązań opartych na tym podejściu można znaleźć w plikach `tess1.c`, `tess2.cpp` oraz `tess3.pas`.

Drugie rozwiązanie wzorcowe

W tym rozwiązaniu również będziemy przyspieszać podany na początku algorytm bezpośredni, tym razem jednak w zupełnie inny sposób, a mianowicie rozpatrując wszystkie ciągi b_i naraz. Będzie to wyglądało mniej więcej tak, jakbyśmy równolegle uruchomili funkcję *podciąg* dla wszystkich ciągów b_i i śledzili tylko, dla każdego z tych ciągów, w którym jego miejscu znajduje się jego aktualnie „oczekiwany" (tj. poszukiwany w ciągu a) element, wskazywany w oryginalnej funkcji przez zmienną k. Skorzystamy z faktu, że każdy z tych oczekiwanych elementów znajduje się w zbiorze S, dzięki czemu ciągi b_i będziemy mogli w każdym momencie podzielić na grupy oczekujących na poszczególne elementy $c \in S$.

W rozwiązaniu wykorzystamy trochę nietypową strukturę danych, a mianowicie *worek*. Jak sama nazwa wskazuje, jej zadaniem jest przechowywanie elementów z pewnego zbioru. Worek udostępnia dwie operacje: można włożyć do niego podany element oraz wyciągnąć z niego jakiś (bliżej nieokreślony) element. Będziemy tu dodatkowo zakładać, iż nigdy nie będziemy chcieli wstawić do worka czegoś, co już w nim się znajduje. Dla każdego elementu $c \in S$ będziemy utrzymywać właśnie taki worek numerów ciągów b_i, które w danej chwili oczekują na pojawienie się w ciągu a elementu c. I teraz w miarę przeglądania kolejnych elementów ciągu a będziemy przerzucać elementy pomiędzy workami, symulując przesunięcie wszystkim ciągom z worka odpowiadającego danemu a_i wskaźnika aktualnie oczekiwanego elementu na następny.

Poniższy pseudokod realizuje opisane podejście. Tablica *worki* przechowuje aktualne zawartości worków związanych z poszczególnymi elementami $c \in S$, tablica

zadowolony — informacje o tym, które spośród ciągów b_i okazały się już podciągami ciągu a, wreszcie w tablicy k pamiętamy, dla każdego ciągu b_i, indeks aktualnie oczekiwanego elementu tego ciągu (zbieżność nazwy tej tablicy ze zmienną k w funkcji *podciąg* jest nieprzypadkowa).

```
1:  Rozwiązanie wzorcowe z użyciem worków:
2:    worki[1..s] := (∅, ∅, ..., ∅);
3:    zadowolony[1..n] := (false, false, ..., false);
4:    k[1..n] := (1, 1, ..., 1);
5:    for i := 1 to n do worki[b_{i,1}].insert(i);
6:    for j := 1 to m do begin
7:      W := worki[a_j]; { kopiujemy cały worek, a nie referencję do niego }
8:      worki[a_j] := ∅;
9:      foreach i ∈ W do begin
10:       k[i] := k[i] + 1;
11:       if k[i] > m_i then zadowolony[i] := true
12:       else worki[b_{i,k[i]}].insert(i);
13:     end
14:   end
15:   for i := 1 to n do
16:     if zadowolony[i] then write(TAK) else write(NIE);
```

Do zakończenia opisu powyższego algorytmu pozostał jeszcze jeden drobiazg — kwestia reprezentacji naszych worków. Odpowiednia struktura danych powinna umożliwiać wstawianie (metoda *insert* powyżej) oraz usuwanie elementów, w jakiejkolwiek kolejności. Zauważmy, że za pomocą tych operacji można bez problemów symulować przeniesienie zawartości jednego worka do innego (wiersze 7-8), a także jednokrotne przejrzenie wszystkich elementów worka (pętla **foreach** w wierszu 9). Do tego celu świetnie nadaje się większość klasycznych dynamicznych struktur danych: stos (kolejka LIFO), kolejka (FIFO), lista itp. — o tych i innych strukturach dynamicznych można poczytać w większości książek poświęconych algorytmom, np. [19] czy [21]. Każda z nich umożliwia wstawianie i usuwanie elementów w czasie stałym. Dodajmy tylko, że programujący w C++ mogą korzystać z gotowych implementacji dynamicznych struktur danych w bibliotece STL, takich jak `stack`, `queue`, `deque`, `list`, czy nawet wspomniany już wcześniej `vector`.

Spróbujmy wreszcie przeanalizować złożoność czasową opisanego rozwiązania. Widać wyraźnie, że główny koszt czasowy stanowią pętle w wierszach 6-14. Można by szacować ten koszt zgrubnie — pętla **for** wykonuje m obrotów, a w każdym z nich przeglądamy jakiś worek, którego rozmiar nie przekracza n — i w ten sposób otrzymać oszacowanie $O(n \cdot m)$. Pokażemy jednak, że złożoność czasowa opisanego rozwiązania jest istotnie mniejsza.

W tym celu zastosujemy analizę kosztu zamortyzowanego (patrz np. książka [21]), tylko w bardzo, bardzo prostej postaci. Pokażemy, że jakkolwiek pojedyncza sekwencja obrotów pętli **foreach** (wiersz 9) może być długa, to *łączna* liczba jej obrotów jest ograniczona. Zauważmy, że w wyniku każdego takiego obrotu wskaźnik oczekiwanego elementu w jakimś ciągu (tj. $k[i]$) przesuwa się o 1. Łączna liczba zwiększeń

danego $k[i]$ nie może przekroczyć m_i, co pokazuje, że sumaryczna liczba obrotów pętli **foreach** jest ograniczona przez $m_1 + m_2 + \ldots + m_n = len$. Dodając do tego koszt czasowy inicjacji struktur danych (wiersze 2-5), wypisywania wyniku (wiersze 15-16) i „jałowych" obrotów pętli **for** (pusty worek), otrzymujemy łączną złożoność czasową tego rozwiązania: $O(len + m + s)$. Jest ona lepsza niż w przypadku poprzedniego algorytmu, ale w praktyce różnica jest na tyle nieznaczna, że oba zostały uznane za wzorcowe. Dodajmy, że złożoność pamięciowa jest tu taka sama jak czasowa.

Implementacje tego rozwiązania można znaleźć w plikach `tes.cpp`, `tes1.c`, `tes2.pas` i `tes3.cpp`.

Testy

Rozwiązania niniejszego zadania były sprawdzane na 10 zestawach danych testowych. Testy *a* to testy losowe, które w tym zadaniu pełnią też rolę testów poprawnościowych. Testy *b* zawierają dużo zapytań o w miarę krótkie podciągi i pełnią rolę testów wydajnościowych. Test *10c* to maksymalny i zarazem skrajny przypadek testowy.

Nazwa	m	n	len	odpowiedzi tak/nie
tes1.in	20	10	100	3 / 7
tes2.in	1 000	100	100 000	10 / 90
tes3a.in	10 000	10 000	100 000	1 460 / 8 540
tes3b.in	100 000	2 500	100 000	1 900 / 600
tes4a.in	100 000	5 000	800 000	422 / 4 578
tes4b.in	200 000	5 000	200 000	3 785 / 1 215
tes5a.in	300 000	9 000	1 000 000	625 / 8 375
tes5b.in	300 000	7 500	300 000	5 560 / 1 940
tes6a.in	500 000	1 000	500 000	999 / 1
tes6b.in	500 000	12 500	500 000	9 358 / 3 142
tes7a.in	700 000	2 000	800 000	887 / 1 113
tes7b.in	700 000	17 500	700 000	13 141 / 4 359
tes8a.in	800 000	4 000	900 000	1 868 / 2 132
tes8b.in	800 000	20 000	800 000	14 944 / 5 056
tes9a.in	900 000	10 000	1 000 000	9 802 / 198
tes9b.in	900 000	22 500	900 000	16 924 / 5 576
tes10a.in	1 000 000	100 000	1 000 000	95 740 / 4 260
tes10b.in	1 000 000	25 000	1 000 000	18 700 / 6 300
tes10c.in	1 000 000	1	1 000 000	1 / 0

Zawody II stopnia

opracowania zadań