Jacek Tomasiewicz Treść zadania | **Jakub Radoszewski** Opracowanie | **Dawid Dąbrowski, Tomasz Syposz** Program

Dostępna pamięć: 128 MB. *OI, etap II, dzień pierwszy, 11.02.2015*

Podział naszyjnika

Mamy naszyjnik złożony z n koralików, z których każdy jest jednego z k rodzajów. Koraliki numerujemy liczbami całkowitymi od 1 do n. Koralik o numerze i sąsiaduje w naszyjniku z koralikami o numerach $i+1$ oraz $i-1$ (o ile koraliki o takich numerach istnieją), a ponadto koraliki o numerach 1 oraz n również sąsiadują ze sobą. Chcemy podzielić naszyjnik dwoma cięciami na dwie niepuste części tak, aby każdy rodzaj koralika występował dokładnie w jednej z części (tzn. jeśli jedna z części zawiera koralik rodzaju j, to druga część nie może zawierać żadnego koralika rodzaju j). Na ile sposobów możemy to zrobić oraz jaka jest minimalna różnica długości otrzymanych części?

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n i k ($2 \leqslant k \leqslant n \leqslant 1\,000\,000$) oddzielone pojedynczym odstępem, oznaczające długość naszyjnika i liczbę rodzajów koralików. Rodzaje koralików numerujemy liczbami od 1 do k. Drugi wiersz wejścia zawiera ciąg n liczb całkowitych $r_1, r_2, \ldots, r_n$ ($1 \leqslant r_i \leqslant k$) pooddzielanych pojedynczymi odstępami; liczba r_i oznacza rodzaj koralika o numerze i. Możesz założyć, że każdy rodzaj koralika wystąpi w naszyjniku co najmniej raz.

W testach wartych łącznie 30% punktów zachodzi dodatkowy warunek $n \leqslant 1000$.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać dwie liczby całkowite oddzielone pojedynczym odstępem. Pierwsza z nich ma oznaczać liczbę sposobów, na jakie można podzielić naszyjnik (możesz założyć, że dla danych wejściowych co najmniej jeden podział jest możliwy). Druga liczba ma oznaczać minimalną różnicę długości otrzymanych części.

Przykład

Dla danych wejściowych:

```
9 5
2 5 3 2 2 4 1 1 3
```

poprawnym wynikiem jest:

```
4 3
```

Wyjaśnienie do przykładu: *Są cztery możliwe podziały; krótsza część może zawierać koraliki (5), (4), $(1,1)$ lub $(4,1,1)$. W ostatnim przypadku uzyskujemy optymalną różnicę długości $6-3=3$.*

Testy „ocen”:

`1ocen`: *krótki naszyjnik z dwoma możliwymi podziałami ($n = 10$);*

`2ocen`: *naszyjnik długości 1000, wszystkie koraliki mają różne rodzaje, wszystkie podziały są poprawne;*

`3ocen`: *$n = 1\,000\,000$, $k = n/2$, naszyjnik postaci $1, \ldots, k, k, \ldots, 1$.*

Rozwiązanie

Dany jest cykliczny naszyjnik złożony z n koralików, które możemy ponumerować od 1 do n. Dla każdego koralika znamy jego rodzaj r_i określony liczbą z zakresu od 1 do k. W zadaniu interesują nas podziały naszyjnika za pomocą dwóch cięć na dwie części, takie że każdy rodzaj koralika występuje w dokładniej jednej z części. Każdy taki podział nazwiemy *prawidłowym*. Naszym zadaniem jest wyznaczenie liczby prawidłowych podziałów naszyjnika oraz prawidłowego podziału na dwie części o jak najbardziej zbliżonych długościach. Wiemy przy tym, że w naszyjniku występuje każdy rodzaj koralika od 1 do k oraz że istnieje przynajmniej jeden prawidłowy podział naszyjnika.

Od cyklu do ciągu

Jeśli nasze dwa cięcia wykonujemy przed koralikami o numerach a oraz b, dla $1 \leqslant a < b \leqslant n$, to naszyjnik rozpada się na części $a, \ldots, b-1$ oraz $b, \ldots, n, 1, \ldots, a-1$. To oznacza, że zawsze jedna z części będzie spójnym fragmentem oryginalnego ciągu (r_i). Tak więc każdy podział naszyjnika odpowiada jakiemuś fragmentowi ciągu (r_i), i odwrotnie: każdy fragment ciągu (r_i) odpowiada jakiemuś podziałowi naszyjnika. Może się jednak zdarzyć, że dwa fragmenty ciągu (r_i) będą odpowiadać temu samemu podziałowi naszyjnika. Przy powyższych oznaczeniach zachodzi to tylko w przypadku, gdy $a = 1$: wówczas otrzymane części naszyjnika to $1, \ldots, b-1$ oraz $b, \ldots, n$. Aby uniknąć dwukrotnego rozpatrywania tego samego podziału, możemy na przykład rozważać tylko fragmenty ciągu kończące się przed pozycją n (odpowiadają one wtedy zawsze fragmentom $a, \ldots, b-1$ z powyższego opisu). Równoważnie moglibyśmy też rozważać tylko fragmenty niezaczynające się na pierwszej pozycji.

Fragment ciągu nazwiemy *prawidłowym*, jeśli odpowiadający mu podział naszyjnika jest prawidłowy. Fragment jest więc prawidłowy, jeśli każdy rodzaj koralika występuje tylko w nim albo nie występuje w nim wcale. Używając tej definicji, możemy teraz sformułować zadanie w wygodniejszy sposób, unikając cykliczności naszyjnika: zamiast zliczać prawidłowe podziały naszyjnika, możemy równoważnie zliczać prawidłowe fragmenty ciągu kończące się przed pozycją n, a zamiast szukać prawidłowego podziału naszyjnika na części o możliwie zbliżonych długościach, wystarczy szukać prawidłowego fragmentu ciągu o długości możliwie zbliżonej do $\frac{n}{2}$ (fragment ten może być dłuższy lub krótszy od $\frac{n}{2}$, jako że nie wiemy, czy odpowiada on dłuższej, czy też krótszej części naszyjnika po podziale).

Jakie fragmenty ciągu są prawidłowe?

Zastanówmy się, na jakiej pozycji j może kończyć się prawidłowy fragment ciągu zaczynający się na danej pozycji i. Wiemy już, że dany fragment ciągu jest prawidłowy, jeśli dla każdego rodzaju koralika albo (1) wszystkie koraliki tego rodzaju występują w tym fragmencie, albo (2) żaden koralik tego rodzaju nie występuje w tym fragmencie. W ten sposób każdy rodzaj koralika nakłada pewne warunki na pozycję j. Spróbujmy określić dokładniej jakie to warunki.

Załóżmy, że koraliki ustalonego rodzaju $c \in \{1, \ldots, k\}$ występują w ciągu (r_i) na pozycjach $p_{c,1} < p_{c,2} < \ldots < p_{c,n_c}$. Zauważmy, że jeśli $n_c = 1$, czyli koralik tego rodzaju występuje w ciągu dokładnie raz, to albo ten koralik wystąpi w rozważanym fragmencie i wtedy dla fragmentu będzie spełniony warunek (1), albo też koralik ten nie wystąpi w tym fragmencie i wtedy dla fragmentu będzie spełniony warunek (2). To oznacza, że taki rodzaj koralika nie nakłada żadnych ograniczeń na koniec fragmentu.

Przyjmijmy teraz, że $n_c \geqslant 2$. Wówczas wszystko zależy od położenia i w ciągu pozycji $p_{c,1}, \ldots, p_{c,n_c}$. Jeśli $i \leqslant p_{c,1}$, to warunek (1) dla tego rodzaju koralików będzie spełniony dla pozycji $j \geqslant p_{c,n_c}$, a warunek (2) dla pozycji $j < p_{c,1}$. Tak więc przedział pozycji *zabronionych*, na których nie może kończyć się rozważany fragment, to $[p_{c,1}, p_{c,n_c} - 1]$. Jeśli $p_{c,l} < i \leqslant p_{c,l+1}$ dla pewnego l, to warunek (2) będzie spełniony dla pozycji $j < p_{c,l+1}$, zaś warunek (1) nigdy nie będzie spełniony, gdyż koralik znajdujący się na pozycji $p_{c,1}$ na pewno znajdzie się poza fragmentem. W ten sposób zabronione są pozycje $[p_{c,l+1}, n]$. Wreszcie jeśli $i > p_{c,n_c}$, to dowolne $j \leqslant n$ spełnia warunek (2), więc żadna pozycja nie jest zabroniona.

Ostatecznie dla każdego rodzaju koralika otrzymaliśmy co najwyżej jeden przedział pozycji zabronionych, czyli takich, na których nie może kończyć się prawidłowy fragment zaczynający się na pozycji i.

Wyznaczanie przedziałów zabronionych

Aby pójść naprzód, musimy jednak umieć efektywnie wyznaczać przedziały pozycji zabronionych (w skrócie: przedziały zabronione) dla każdej kolejnej pozycji początkowej fragmentu i. Dla pozycji $i = 1$ możemy to zrobić siłowo. Natomiast gdy przemieszczamy się o jedną pozycję dalej (z i do $i + 1$), zmianie może ulec jedynie przedział zabroniony odpowiadający koralikowi z opuszczanej właśnie pozycji.

Okazuje się, że do wyznaczenia zmieniających się przedziałów wystarczą dwie tablice:

- *next*$[i]$ – przechowująca dla każdej pozycji $i \in \{1, \ldots, n\}$, pozycję następnego koralika rodzaju r_i lub ∞, jeśli takiego koralika nie ma,
- *interval*$[c]$ – przechowująca dla każdego rodzaju koralika $c \in \{1, \ldots, k\}$, przedział zabroniony wynikający z tego rodzaju koralika dla bieżącej pozycji i.

Faktycznie, załóżmy, że przesuwamy się z pozycji i na pozycję $i + 1$. Mamy wtedy kilka przypadków. Niech $c = r_i$.

- Jeśli *interval*$[c]$ był pusty, to pozostaje pusty; odpowiada to przypadkowi $n_c = 1$.

- W przeciwnym razie, jeśli $next[i] < \infty$, to $interval[c]$ staje się $[next[i], n]$; odpowiada to przypadkowi $i = p_{c,l}$ dla $l < n_c$, czyli $p_{c,l} < i + 1 \leqslant p_{c,l+1}$.
- Jeśli zaś $next[i] = \infty$, to $interval[c]$ staje się pusty; odpowiada to przypadkowi $i = p_{c,n_c}$, czyli $i + 1 > p_{c,n_c}$.

Zauważmy, że na powyższej liście (oczywiście) nie występuje przypadek $i + 1 \leqslant p_{c,1}$.

Elementy tablicy *next* można wyznaczyć w czasie $O(n + k) = O(n)$. W tym celu wystarczy przejść po wszystkich elementach ciągu od końca do początku, przechowując w tablicy rozmiaru k pozycje ostatnio napotkanych koralików każdego rodzaju. Podobnie w czasie $O(n)$ można wyznaczyć początkowe przedziały zabronione dla pozycji 1. Wreszcie wyznaczenie zmieniających się przedziałów zabronionych dla poszczególnych pozycji i zgodnie z opisaną procedurą wykonujemy w czasie $O(n)$. Aby nigdy nie rozważać fragmentów kończących się na pozycji n, na samym początku dodajemy dodatkowy przedział zabroniony $[n, n]$.

Drzewo przedziałowe

Umiemy już efektywnie wyznaczać przedziały zabronione dla poszczególnych pozycji początkowych. Musimy teraz wymyślić, jak na tej podstawie uzyskać odpowiedzi na interesujące nas pytania. Nie będzie niespodzianką, że możemy w tym celu zastosować (statyczne) *drzewo przedziałowe*. Więcej o tej strukturze danych można przeczytać np. w opracowaniu zadania *Logistyka* w tej książeczce i w zawartych tam odnośnikach.

Skoncentrujmy się na razie na pierwszym pytaniu z zadania. Aby na nie odpowiedzieć, wystarczy wyznaczyć liczbę prawidłowych fragmentów zaczynających się na poszczególnych pozycjach ciągu. Liczba ta zależy wprost od liczby pozycji końcowych zawartych w przedziałach zabronionych, czyli od łącznej długości sumy tych przedziałów.

W drzewie przedziałowym będziemy przechowywać aktualne przedziały zabronione. Standardowo, gdy dodajemy nowy przedział zabroniony, wstawiamy go do drzewa z wagą $+1$, a gdy zmieniamy przedział zabroniony, to stary przedział wstawiamy z wagą -1, a następnie nowy z wagą $+1$. Wstawienie przedziału do drzewa polega na rozbiciu go na *przedziały bazowe*, odpowiadające węzłom drzewa. W każdym węźle drzewa v przechowujemy, jako $w[v]$, sumę wag ze wstawionych przedziałów, dla których v znalazł się w rozkładzie na przedziały bazowe. Dzięki temu, że z wagą -1 wstawiamy przedziały, które wcześniej wstawiliśmy z wagą $+1$, wartości w będą zawsze nieujemne.

W każdym węźle drzewa v będziemy także przechowywać liczbę $f[v]$ zabronionych pozycji w jego poddrzewie. Dokładniej, będzie to liczba pozycji zabronionych, biorąc pod uwagę jedynie wartości w z poddrzewa v – jeśli jakiś przodek węzła v ma dodatnią wartość w, to niezależnie od faktycznej wartości $f[v]$ każda pozycja w poddrzewie jest zabroniona, jednak węzeł v nie będzie o tym wiedział. Ostatecznie szukaną liczbę prawidłowych fragmentów wyznaczymy na podstawie wartości f w korzeniu.

Trzeba jeszcze pokazać, że umiemy aktualizować wartości f przy wstawianiu przedziałów. Standardowo, po wykonaniu każdego wstawienia, aktualizujemy te wartości we wszystkich węzłach znajdujących się w przodkach węzłów z rozkładu na przedziały

bazowe. Wystarczy teraz zauważyć, że do wyznaczenia wartości f dla węzła v wystarczą nam dane przechowywane w nim i w jego dzieciach: jeśli $w[v] = 0$, to $f[v]$ jest sumą wartości f dla jego dzieci $left[v]$ i $right[v]$, a w przeciwnym razie jest ona równa długości odpowiadającego mu przedziału bazowego $[a[v], b[v]]$.

Zbudowanie drzewa przedziałowego zajmuje czas $O(n)$. Łącznie wykonujemy co najwyżej $2n$ wstawień przedziałów. Każde takie wstawienie na drzewie przedziałowym działa w czasie $O(\log n)$. Ostatecznie ta część rozwiązania działa w czasie $O(n \log n)$.

Najbardziej zrównoważony podział

Nieco więcej uwagi wymaga zastosowanie drzewa przedziałowego do rozwiązania drugiej części zadania, polegającej na znalezieniu prawidłowego fragmentu o długości najbliższej $\frac{n}{2}$. W chwili rozpatrywania danej pozycji początkowej fragmentu i weźmiemy pod uwagę pozycję $j = i + \lfloor \frac{n}{2} \rfloor - 1$, a jeśli wypada ona za pozycją n, to pozycję $j = n$. Naszym celem będzie teraz znalezienie końcowej pozycji prawidłowego fragmentu jak najbliższej pozycji j. Innymi słowy, będziemy chcieli znaleźć najbliższą pozycji j pozycję *dozwoloną*, czyli niezawierającą się w żadnym z przedziałów zabronionych. Będziemy jej szukać zarówno wśród wcześniejszych pozycji (będzie to tzw. *dozwolony poprzednik* pozycji j), jak i późniejszych pozycji (tzw. *dozwolony następnik* pozycji j). Wyszukiwanie tak zdefiniowanego poprzednika i następnika pozycji będzie przypominać nieco analogiczne procedury używane w drzewie wyszukiwań binarnych (np. jako podprocedury w implementacji usuwania węzła z drzewa BST).

Przede wszystkim na początku warto sprawdzić, czy sama pozycja j nie jest dozwolona. Jest tak dokładnie wtedy, gdy suma wartości w na ścieżce od liścia odpowiadającego pozycji j ($leaf[j]$) do korzenia drzewa ($root$) jest równa zeru. Do przechodzenia w górę drzewa użyjemy tablicy $parent$, wskazującej na ojca węzła.

```
1:  v := leaf[j];
2:  sumw := w[v];
3:  while v ≠ root do begin
4:    v := parent[v];
5:    sumw := sumw + w[v];
6:  end
7:  if sumw = 0 then pozycja j jest dozwolona;
```

Jeśli pozycja j okazała się dozwolona, kandydatem na wynik pochodzącym od pozycji i jest właśnie pozycja j. W przeciwnym razie musimy wyznaczyć dozwolonego poprzednika i następnika pozycji j. Skoncentrujemy się tylko na pierwszym z nich; drugiego szuka się symetrycznie (jeśli w ogóle $j < n$).

Aby znaleźć dozwolonego poprzednika pozycji j, przechodzimy ścieżką od liścia $leaf[j]$ do korzenia drzewa w poszukiwaniu pierwszego węzła, do którego ścieżka ta wchodzi od strony prawego syna, natomiast poddrzewo jego lewego syna zawiera jakąś dozwoloną pozycję. Wiemy wtedy, że szukany poprzednik znajduje się w owym lewym poddrzewie. To, czy poddrzewo węzła v zawiera jakąś dozwoloną pozycję, sprawdzamy, porównując wartość $f[v]$ z długością przedziału bazowego $[a[v], b[v]]$ *oraz* biorąc pod uwagę sumę wartości w na ścieżce od węzła v do korzenia drzewa. Sumę tę

uzyskujemy z obliczonej przed chwilą zmiennej *sumw*. Znaleziony węzeł zapisujemy w zmiennej *node*.

```
1:  v := leaf[j];
2:  { Używamy zmiennej sumw z poprzedniego pseudokodu! }
3:  node := nil;
4:  while v ≠ root do begin
5:    sumw := sumw − w[v];
6:    if v = right[parent[v]] then begin
7:      v' := left[parent[v]];
8:      if (sumw = 0) and (f[v'] < b[v'] − a[v'] + 1) then begin
9:        node := v';
10:       break;
11:     end
12:   end
13:   v := parent[v];
14: end
15: return node;
```

Jeśli na koniec *node* = **nil**, to pozycja j nie ma dozwolonego poprzednika. W przeciwnym razie będzie nim skrajnie prawy dozwolony węzeł w poddrzewie węzła *node*. W poniższym pseudokodzie pokazujemy, jak znaleźć taki węzeł. Zauważmy, że nie musimy już używać zmiennej *sumw*, gdyż wiemy, że dla węzła *node* jest ona równa 0. Wynikiem funkcji jest dozwolony poprzednik pozycji j.

```
1:  v := node;
2:  while a[v] ≠ b[v] do begin { dopóki v nie jest liściem }
3:    v' := right[v];
4:    if f[v'] < b[v'] − a[v'] + 1 then v := v';
5:    else v := left[v];
6:  end
7:  return a[v];
```

Chociaż w powyższym opisie do oznaczania parametrów węzłów używaliśmy notacji tablicowej, doświadczeni w używaniu drzewa przedziałowego zawodnicy wiedzą, że wartości typu *parent*$[v]$, *left*$[v]$ oraz *right*$[v]$ można łatwo wyznaczyć na podstawie numeru węzła, a przedziały $[a[v], b[v]]$ na podstawie numeru węzła i jego głębokości – przy założeniu, że drzewo przedziałowe pamiętane jest w statycznej tablicy. Podobnie rzecz ma się z wyznaczaniem węzła *leaf*$[j]$ na podstawie pozycji j.

Wyznaczenie dozwolonego poprzednika i następnika pozycji j wymaga tylko stałej liczby przejść w górę i w dół drzewa przedziałowego, co wykonujemy w czasie $O(\log n)$. Druga część rozwiązania działa zatem także w czasie $O(n \log n)$. Taka jest więc ostateczna złożoność czasowa całego rozwiązania.

Implementacja rozwiązania wzorcowego zgodna z powyższym opisem znajduje się w pliku `pod4.cpp`, natomiast inne implementacje można znaleźć w plikach `pod.cpp` oraz `pod[1-3].cpp`. Przy pisaniu programu trzeba było pamiętać o tym, że do przechowywania pierwszego z obliczanych wyników potrzebna była zmienna całkowita 64-bitowa.