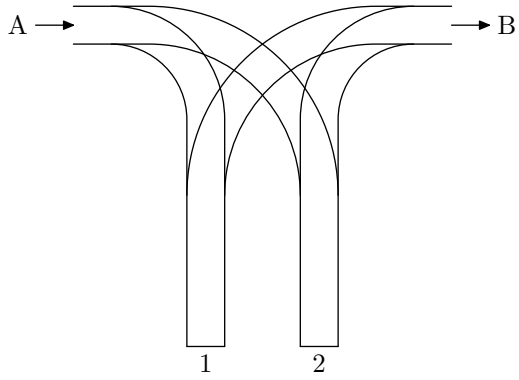
Bartosz Walczak — Treść zadania, Opracowanie
Michał Pilipczuk — Program

Dostępna pamięć: 128 MB. *OI, Etap I, 19.10–16.11.2009*

Kolej

Bocznica kolejowa składa się z dwóch ślepo zakończonych torów 1 i 2. Wjazd na bocznicę odbywa się torem A, a wyjazd torem B (patrz rysunek poniżej).



Na torze A stoi n wagonów ponumerowanych od 1 do n. Są one ustawione w kolejności $a_1, a_2, \ldots, a_n$ *(tzn. w takiej kolejności wjeżdżają na bocznicę). Trzeba je tak przetoczyć przez bocznicę, aby opuściły ją torem B w kolejności* $1, 2, \ldots, n$*. Każdy wagon można dokładnie raz przetoczyć z toru A na jeden z torów 1 lub 2, oraz dokładnie raz z toru 1 lub 2 na tor B. W każdej chwili na każdym z torów 1, 2 może czekać dowolnie wiele wagonów.*

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę naturalną n *(*$1 \leqslant n \leqslant 100\,000$*), oznaczającą liczbę wagonów do przetoczenia. W drugim wierszu znajdują się liczby* $a_1, a_2, \ldots, a_n$ *będące permutacją liczb* $1, 2, \ldots, n$ *(czyli każda z liczb* a_i *należy do zbioru* $\{1, 2, \ldots, n\}$ *oraz wszystkie te liczby są różne), pooddzielane pojedynczymi odstępami.*

Wyjście

Pierwszy wiersz standardowego wyjścia powinien zawierać słowo `TAK`*, jeśli istnieje sposób uporządkowania wagonów w kolejności* $1, 2, \ldots, n$ *poprzez przetaczanie ich przez bocznicę, albo słowo* `NIE`*, jeśli taki sposób nie istnieje. W przypadku, gdy odpowiedzią jest* `TAK`*, drugi wiersz powinien zawierać pooddzielane pojedynczymi odstępami numery torów bocznicy (1 lub 2), na które są wtaczane kolejne wagony* $a_1, a_2, \ldots, a_n$ *w pewnym poprawnym sposobie ich uporządkowania. Jeżeli istnieje wiele możliwych sposobów uporządkowania wagonów, należy wypisać dowolny z nich.*

Przykład

Dla danych wejściowych:	*jednym z poprawnych wyników jest:*
`4`	`TAK`
`1 3 4 2`	`1 1 2 1`
Natomiast dla danych wejściowych:	*poprawnym wynikiem jest:*
`4`	`NIE`
`2 3 4 1`	

Wyjaśnienie do przykładu: *W pierwszym przykładzie zaczynamy od odstawienia wagonu numer 1 na pierwszą bocznicę, po czym zaraz ten sam wagon zjeżdża na tor B. Następnie wagon numer 3 odstawiamy na pierwszą bocznicę, wagon numer 4 — na drugą, wreszcie wagon numer 2 trafia na pierwszą bocznicę. Na koniec z pierwszej bocznicy zjeżdżają na tor B kolejno wagony o numerach 2 i 3, po czym z drugiej bocznicy zjeżdża wagon numer 4.*

Rozwiązanie

Wprowadzenie

Problem z zadania można łatwo przetłumaczyć na język matematyki. Dana jest permutacja $a_1, \ldots, a_n$ liczb od 1 do n. Kolejne jej elementy możemy odkładać na jeden z dwóch stosów. Możemy także zdejmować elementy ze stosów, tworząc drugą permutację. Przy tym z każdego stosu możemy zdjąć tylko element znajdujący się na jego szczycie, czyli element, który został odłożony na ten stos najpóźniej. Pytanie brzmi: czy za pomocą takich dwóch stosów możemy posortować permutację $a_1, \ldots, a_n$? A jeżeli możemy, to jak wygląda jej *algorytm sortowania*, czyli sekwencja ruchów polegających na odkładaniu elementów na stosy i zdejmowaniu elementów ze stosów, prowadzących do posortowania permutacji?

Obserwacja 1. *Każdy algorytm sortowania permutacji utrzymuje na obu stosach malejący porządek elementów (przeglądając stos od spodu do szczytu).*

Dowód: Jeżeli $a_i < a_j$, to element a_i jest zdejmowany ze stosu wcześniej niż a_j. Jeżeli więc a_i i a_j są na tym samym stosie, to a_i musi być bliżej szczytu stosu niż a_j. ■

Działanie każdego algorytmu sortowania permutacji można podzielić na n kroków. W i-tym kroku algorytm odkłada element a_i na jeden ze stosów, po czym zdejmuje ze stosów pewne elementy (być może nie zdejmuje żadnego).

Dla każdego i zdefiniujmy $\ell(i) = \max\{j : a_j \leqslant a_i\}$, jest to indeks w permutacji $a_1, \ldots, a_n$ ostatniego elementu nie większego niż a_i. Żaden element a_i nie może być zdjęty ze stosu wcześniej niż w kroku $\ell(i)$, gdyż dopiero w tym kroku przychodzi element $a_{\ell(i)} \leqslant a_i$. Poniższa obserwacja pokazuje, że można go zdjąć dokładnie w kroku $\ell(i)$.

Obserwacja 2. *Każdy algorytm sortowania permutacji $a_1, \ldots, a_n$ można tak zmodyfikować, aby zdejmował ze stosu każdy element a_i w kroku $\ell(i)$.*

Dowód: Niech A będzie dowolnym algorytmem sortowania permutacji $a_1, \ldots, a_n$. Algorytm A' działa następująco: każdy przychodzący element odkłada na ten sam stos, co algorytm A, oraz każdy element a_i usuwa ze stosu w kroku $\ell(i)$ (bez względu na to, czy element ten jest na szczycie stosu). Przy tym elementy przeznaczone do usunięcia w jednym kroku są usuwane w kolejności rosnącej. Pokażemy, że jest to poprawny algorytm sortowania permutacji $a_1, \ldots, a_n$ za pomocą dwóch stosów.

Ponieważ dla $a_i < a_j$ mamy $\ell(i) \leqslant \ell(j)$, więc elementy są usuwane ze stosów we właściwej kolejności. Musimy jeszcze uzasadnić, że gdy algorytm próbuje usunąć element ze stosu, operacja ta jest dozwolona, czyli usuwany element jest na szczycie stosu. Ponieważ (z definicji) $i \leqslant \ell(i)$, więc usuwany element rzeczywiście znajduje się na stosie. Ponadto, algorytm A' utrzymuje malejący porządek na stosach, ponieważ odkłada elementy na stosy tak samo jak algorytm A i zdejmuje je nie później niż algorytm A, a algorytm A utrzymuje malejący porządek na stosach na podstawie Obserwacji 1. Zatem w chwili, gdy algorytm A' próbuje usunąć element ze stosu, jest to najmniejszy element ze wszystkich pozostałych na stosach, więc znajduje się na szczycie stosu. ■

Na koniec wstępnych rozważań zauważmy jeszcze, że wszystkie indeksy $\ell(i)$ można obliczyć w czasie liniowym. W poniższym pseudokodzie w polu $arrived[j]$ zapisujemy, czy w dotychczas przejrzanym fragmencie ciągu a wystąpiła liczba j, i jeśli tak, na której była pozycji. W zmiennej $next$ jest natomiast najmniejsza wartość, która jeszcze nie wystąpiła przy przeglądaniu permutacji.

```
1:  Algorytm obliczania wartości ℓ(i):
2:    arrived[1..n] := (0, 0, ..., 0);
3:    next := 1;
4:    for i := 1 to n do begin
5:      arrived[a[i]] := i;
6:      while (next ⩽ n) and (arrived[next] ≠ 0) do begin
7:        ℓ[arrived[next]] := i;
8:        next := next + 1;
9:      end
10:   end
```

Redukcja do kolorowania grafu

Wiemy już, kiedy zdejmować elementy ze stosów. Spróbujmy teraz wywnioskować coś o tym, na który stos należy je kłaść. Zacznijmy od prostej obserwacji. Element a_i wkładamy w kroku i-tym, a zdejmujemy w $\ell(i)$-tym, więc, aby zachować malejący porządek stosów, wszystkie a_j, takie że $i < j \leqslant \ell(i)$ oraz $a_i < a_j$, muszą znaleźć się na innym stosie niż a_i.

Otrzymaliśmy zatem zbiór warunków postaci „a_i oraz a_j muszą być odłożone na różne stosy". Przedstawmy je w postaci grafu nieskierowanego $G = (V, E)$, którego wierzchołkami są indeksy $1, \ldots, n$. Krawędzie ilustrują nasze warunki, tzn. dla dowolnych dwóch indeksów $i < j$ istnieje krawędź (i, j) wtedy i tylko wtedy, gdy $a_i < a_j$ oraz $\ell(i) \geqslant j$. Przypomnijmy, że naszym celem jest teraz przyporządkowanie

elementom permutacji stosów. Zamiast tego będziemy mówić o przypisaniu każdemu wierzchołkowi grafu jednego z dwóch kolorów. Graf skonstruowaliśmy w ten sposób, że wierzchołki połączone krawędzią trzeba pokolorować na różne kolory.

W teorii grafów mówimy, że funkcja $k : V \to \{1, 2\}$ jest *poprawnym dwukolorowaniem* grafu $G = (V, E)$, jeżeli dla każdej krawędzi $(i, j) \in E$ zachodzi $k(i) \neq k(j)$. Graf G jest *dwukolorowalny*, gdy istnieje jego poprawne dwukolorowanie.

Stąd, aby permutację a dało się posortować przy pomocy dwóch stosów, odpowiadający jej graf G musi być dwukolorowalny. Naturalne wydaje się teraz pytanie, czy ów warunek jest wystarczający. Okazuje się, że odpowiedź na to pytanie jest twierdząca.

Twierdzenie 1. *Istnieje algorytm sortowania permutacji $a_1, \ldots, a_n$ za pomocą dwóch stosów wtedy i tylko wtedy, gdy odpowiadający jej graf G jest dwukolorowalny. Jeżeli ponadto G jest dwukolorowalny, to istnieje algorytm sortowania permutacji $a_1, \ldots, a_n$, który każdy element a_i odkłada na stos określony przez kolor wierzchołka i.*

Dowód: Konieczność warunku dwukolorowalności grafu G uzasadniliśmy powyżej. Przejdźmy więc od razu do dowodu jego dostateczności. Załóżmy, że graf G można pokolorować dwoma kolorami 1 i 2. Niech $k(i)$ będzie kolorem wierzchołka i. Rozważmy algorytm, który każdy element a_i odkłada na stos o numerze $k(i)$ oraz usuwa ze stosu w kroku $\ell(i)$, przy czym elementy przeznaczone do usunięcia w jednym kroku są usuwane w kolejności rosnącej. Pokażemy (analogicznie jak w dowodzie Obserwacji 2), że jest to poprawny algorytm sortowania permutacji $a_1, \ldots, a_n$ za pomocą dwóch stosów. Ponieważ dla $a_i < a_j$ mamy $\ell(i) \leqslant \ell(j)$, więc elementy są usuwane ze stosów we właściwej kolejności. Musimy jeszcze uzasadnić, że gdy algorytm chce usunąć element ze stosu, operacja ta jest dozwolona. W tym celu wystarczy udowodnić, że algorytm utrzymuje malejący porządek elementów na stosach.

Popatrzmy na dowolne elementy a_i, a_j, takie że $i < j$, $a_i < a_j$ i $k(i) = k(j)$. Wówczas $\ell(i) < j$. W przeciwnym przypadku w grafie G mielibyśmy krawędź (i, j), więc wierzchołki i, j nie mogłyby otrzymać tego samego koloru. Zatem w chwili odłożenia na stos elementu a_j, elementu a_i już na tym stosie nie ma (został zdjęty wcześniej). ■

Twierdzenie to naturalnie uogólnia się na dowolną liczbę stosów: problem sortowania permutacji za pomocą k stosów jest równoważny problemowi k-kolorowania grafu G.

Można by pomyśleć, że zadanie jest już rozwiązane: budujemy graf G i kolorujemy go dwoma kolorami na przykład za pomocą przeszukiwania w głąb[1]. Ale czy na pewno? Niestety, istnieją permutacje (również takie z odpowiedzią pozytywną), dla których graf G ma kwadratową liczbę krawędzi. Oto przykład:

$$m, m-1, \ldots, 2, n, n-1, \ldots, m+1, 1, \quad \text{gdzie} \quad m \approx \tfrac{n}{2}.$$

Zatem złożoność obliczeniowa takiego algorytmu w pesymistycznym przypadku wynosi $\Theta(n^2)$ i jest niewystarczająca do potrzeb naszego zadania.

[1] Mowa o algorytmie DFS, opisanym np. w książce [21].

Redukcja rozmiaru grafu

Okazuje się, że zamiast pełnego grafu G wystarczy skonstruować jego *las rozpinający* — podgraf acykliczny grafu G, który ma dokładnie te same spójne składowe co G. Taki podgraf ma rozmiar liniowy względem liczby wierzchołków. Oczywiście, każde poprawne dwukolorowanie grafu G indukuje poprawne dwukolorowanie lasu rozpinającego. Prawdziwa jest również odwrotna własność.

Obserwacja 3. *Niech L będzie lasem rozpinającym graf $G = (V, E)$ i niech $k : V \to \{1, 2\}$ będzie poprawnym dwukolorowaniem lasu L. Wówczas albo k jest poprawnym dwukolorowaniem grafu G, albo graf G nie jest dwukolorowalny.*

Dowód: Załóżmy, że dla pewnej krawędzi $(i, j) \in E$ zachodzi $k(i) = k(j)$. Wierzchołki i, j są w tej samej spójnej składowej lasu L, więc są w L połączone ścieżką. Wierzchołki na tej ścieżce mają kolory na przemian równe $k(i)$ i różne od $k(i)$, więc skoro $k(i) = k(j)$, ścieżka ta ma parzystą liczbę krawędzi. Zatem wraz z krawędzią (i, j) tworzy w grafie G cykl nieparzystej długości, który nie jest dwukolorowalny. ■

Tak dochodzimy do następującego schematu rozwiązania:

1. Dla podanej na wejściu permutacji $a_1, \ldots, a_n$ skonstruuj las L rozpinający graf G.
2. Znajdź poprawne dwukolorowanie lasu L.
3. Sprawdź, czy znalezione dwukolorowanie jest poprawnym dwukolorowaniem grafu G.

Realizacja punktu 2 jest bardzo łatwa — wystarczy przeszukać las L algorytmem przeszukiwania w głąb, kolorując napotykane wierzchołki na przemian liczbami 1 i 2. W punkcie 3 oczywiście nie możemy sprawdzać osobno każdej krawędzi grafu G, gdyż prowadziłoby to ponownie do rozwiązania kwadratowego. Jednak z Twierdzenia 1 wiemy, że w poprawnym dwukolorowaniu kolor każdego wierzchołka i jest numerem stosu, na który należy odłożyć element a_i w algorytmie sortowania permutacji. Sprawdzamy zatem, czy otrzymany z dwukolorowania lasu L algorytm sortowania permutacji jest poprawny.

```
1:  S[1].push(∞); S[2].push(∞);
2:  next := 1; { następny element do zdjęcia }
3:  for i := 1 to n do begin
4:    if a[i] > S[k[i]].top then return NIE;
5:    S[k[i]].push(a[i]);
6:    { zdejmowanie ze stosów kolejnych elementów permutacji posortowanej: }
7:    while (S[1].top = next) or (S[2].top = next) do begin
8:      if S[1].top = next then S[1].pop else S[2].pop;
9:      next := next + 1;
10:   end
11: end
12: return TAK, k[1..n];
```

Pozostaje jeszcze do zrealizowania punkt 1 przedstawionego powyżej schematu rozwiązania.

Warto przy okazji zaznaczyć, że ta sama idea (dwukolorowanie lasu rozpinającego graf pomocniczy) wystąpiła w rozwiązaniu zadania *Autostrady* z II etapu X Olimpiady Informatycznej [10].

Konstrukcja lasu rozpinającego – rozwiązanie wzorcowe

Zauważmy, że zwykłe przeszukiwanie w głąb grafu G konstruuje las rozpinający graf G — jest to tak zwany las przeszukiwania w głąb. Krawędź (i, j) w tym lesie oznacza, że procedura przeszukiwania w głąb, przeglądając sąsiedztwo wierzchołka i, znalazła nieodwiedzony wierzchołek j (albo na odwrót) i wywołała się dla niego rekurencyjnie. Spróbujmy więc zasymulować przeszukiwanie w głąb grafu G. Zauważmy przy tym, że poza znajdowaniem nieodwiedzonego sąsiada algorytm DFS działa w czasie $O(n)$. Potrzebujemy zatem odpowiednich struktur przyspieszających to wąskie gardło rozwiązania. Poniższy pseudokod pokazuje, jak za ich pomocą znaleźć las rozpinający i od razu pokolorować go na dwa kolory.

```
1:  procedure modified_dfs(i, color)
2:  begin
3:    k[i] := color;
4:    delete(i);
5:    while true do begin
6:      j := get_edge(i);
7:      if j = nil then break;
8:      modified_dfs(j, 3 − color);
9:    end
10: end
```

Funkcja *get_edge*(i) zwraca dowolnego nieodwiedzonego sąsiada wierzchołka i (albo **nil**, jeżeli wszystkie sąsiednie wierzchołki są już odwiedzone), natomiast procedura *delete*(i) usuwa wierzchołek i ze struktury przechowującej nieodwiedzone wierzchołki.

Implementacja operacji *get_edge* i *delete* musi oczywiście wykorzystywać szczególny sposób powstawania grafu G z wejściowej permutacji. Dla wygody wprowadźmy relację porządku $\prec$ na wierzchołkach grafu G: niech $i \prec j$ oznacza, że $a_i < a_j$. Dla każdego wierzchołka i grafu G incydentne do niego krawędzie dzielimy na dwie kategorie:

- *krawędzie w przód*, czyli krawędzie $(i, j) \in E$, takie że $i < j \leqslant \ell(i)$ oraz $i \prec j$,
- *krawędzie w tył*, czyli krawędzie $(i, j) \in E$, takie że $j < i \leqslant \ell(j)$ oraz $j \prec i$.

Do przechowywania obu typów krawędzi prowadzących do nieodwiedzonych sąsiadów użyjemy dwóch osobnych struktur danych.

Krawędzie w przód Funkcja *get_edge*(i) ograniczona do krawędzi w przód powinna zwrócić nieodwiedzony wierzchołek j, taki że $i < j \leqslant \ell(i)$ oraz $j \succ i$. W tym celu wystarczy znaleźć wśród nieodwiedzonych wierzchołków w przedziale $[i+1, \ell(i)]$ maksymalny w porządku $\prec$ wierzchołek j, po czym sprawdzić, czy $a_j > a_i$.

Strukturę, która pozwala efektywnie znajdować maksimum na przedziale, można zbudować na schemacie *drzewa przedziałowego*[2]. To powszechne jego zastosowanie bywa nazywane *drzewem turniejowym*. Każdy jego liść i posiada flagę *visited*$[i]$, która mówi, czy wierzchołek i grafu G jest odwiedzony. Ponadto w każdym węźle w jest zapisany indeks $max[w]$ maksymalnego (w porządku $\prec$) nieodwiedzonego liścia w poddrzewie o korzeniu w. Istota drzewa turniejowego polega na łatwości obliczenia $max[w]$ — dla liścia jest to w, gdy *visited*$[w]$ = **false**, oraz **nil** w przeciwnym razie, zaś dla węzła wewnętrznego jest to po prostu większy (w porządku $\prec$) z dwóch indeksów *max* zapisanych w jego dzieciach (zakładamy przy tym, że **nil** jest mniejszy od jakiegokolwiek innego indeksu).

Korzystając z opisanej struktury, operację wyszukiwania maksymalnego nieodwiedzonego wierzchołka $j \in [i+1, \ell(i)]$ można zrealizować w czasie $O(\log n)$. Wystarczy bowiem sprawdzić indeksy $max[w]$ zapisane w korzeniach maksymalnych (w sensie zawierania) poddrzew, których przedział $[x_w, y_w]$ zawiera się w $[i+1, \ell(i)]$, a takich poddrzew jest $O(\log n)$. Zmiana wartości flagi *visited*$[i]$ (wykonywana w procedurze *delete*(i)) również zajmuje czas $O(\log n)$, gdyż wymaga ona aktualizacji indeksów $max[w]$ tylko w węzłach w leżących na ścieżce od korzenia do liścia i (a każda taka ścieżka ma długość $O(\log n)$).

Krawędzie w tył Wyszukiwanie krawędzi w tył prowadzących do nieodwiedzonych sąsiadów jest nieco bardziej skomplikowane. Funkcja *get_edge*(i) ograniczona do krawędzi w tył musi dla danego i znaleźć taki nieodwiedzony wierzchołek j, że $j < i \leqslant \ell(j)$ oraz $j \prec i$. Będziemy więc szukać minimalnego w porządku $\prec$ nieodwiedzonego wierzchołka j, takiego że $j < i \leqslant \ell(j)$.

Tu wykorzystamy strukturę zwaną *drzewem z nasłuchiwaczami*, również opartą na drzewie przedziałowym. W każdym jego węźle znajduje się lista *nasłuchiwaczy*. Każdy nasłuchiwacz na liście węzła w reprezentuje nieodwiedzony wierzchołek j, taki że $[x_w, y_w] \subseteq [j+1, \ell(j)]$. Każdy nieodwiedzony wierzchołek j ma swoje nasłuchiwacze tylko w tych węzłach w, które są korzeniami maksymalnych (w sensie zawierania) poddrzew zawierających się w $[j+1, \ell(j)]$. Zatem każdy nieodwiedzony wierzchołek j generuje $O(\log n)$ nasłuchiwaczy oraz na każdej ścieżce od korzenia do liścia w przedziale $[j+1, \ell(j)]$ znajduje się nasłuchiwacz reprezentujący j.

Aby znaleźć krawędź w tył z wierzchołka i do nieodwiedzonego sąsiada, szukamy na ścieżce od korzenia do liścia i nasłuchiwacza minimalnego w porządku $\prec$ wierzchołka j. Jako że listy nasłuchiwaczy mogą być długie, utrzymujemy dodatkowy niezmiennik: nasłuchiwacze na każdej liście są posortowane zgodnie z porządkiem $\prec$. Wówczas znalezienie minimalnego nasłuchiwacza wymaga przejrzenia tylko pierwszych elementów list w węzłach na ścieżce od korzenia do i, co zajmuje łącznie czas

[2] Jest to pełne, całkowicie zrównoważone drzewo binarne, którego liśćmi są kolejne elementy przedziału $[1, n]$, zaś każdy węzeł wewnętrzny w reprezentuje przedział $[x_w, y_w]$ wszystkich elementów, które są liśćmi poddrzewa o korzeniu w. Bardziej szczegółowy opis drzew przedziałowych można znaleźć np. w opracowaniu zadania *Latarnia* w tej książeczce oraz w podanych tam odnośnikach.

$O(\log n)$. Aby w procedurze budowania drzewa skonstruować posortowane listy nasłuchiwaczy, przeglądamy wierzchołki $j \in [1, n]$ w kolejności rosnącej względem $\prec$ i dla każdego z nich dodajemy wszystkie jego nasłuchiwacze na koniec odpowiednich list. Tym samym konstrukcja drzewa z nasłuchiwaczami zajmuje czas $\Theta(n \log n)$.

Procedura *delete*(i) wymaga usunięcia wszystkich nasłuchiwaczy wierzchołka i. Aby umożliwić efektywne wykonanie tej operacji, podczas konstrukcji drzewa łączymy wszystkie nasłuchiwacze wierzchołka i w dodatkową listę. Wówczas usunięcie każdego nasłuchiwacza zajmuje czas stały, więc łączny czas uaktualnienia drzewa przez procedurę *delete*(i) wynosi $O(\log n)$ [3].

Pokazaliśmy, że obie operacje *get_edge* i *delete* możemy wykonać w czasie $O(\log n)$, używając struktur danych, które mają całkowity rozmiar $\Theta(n \log n)$ i których inicjalizacja zajmuje czas $\Theta(n \log n)$. Ponieważ przeszukiwanie całego grafu procedurą *modified_dfs* wykonuje liniową liczbę operacji *get_edge* i *delete*, więc złożoność czasowa i pamięciowa rozwiązania wzorcowego wynosi $\Theta(n \log n)$. Rozwiązanie to zostało zaimplementowane w pliku `kol.cpp`.

Konstrukcja lasu rozpinającego – algorytm on-line

Oznaczmy przez G_i podgraf grafu G, który powstaje przez obcięcie G do podzbioru wierzchołków $\{1, \ldots, i\}$. Rozwiązanie, które teraz pokażemy, działa *on-line* — konstruuje kolejno lasy rozpinające grafy $G_1, \ldots, G_n$, przy czym do obliczenia lasu rozpinającego graf G_i wykorzystuje tylko informację o elementach $a_1, \ldots, a_i$ wejściowej permutacji.

W i-tym kroku algorytmu dokładamy wierzchołek i oraz uaktualniamy las rozpinający G_{i-1} do lasu rozpinającego G_i. W tym celu musimy wierzchołek i połączyć krawędzią z pewnym wierzchołkiem każdej spójnej składowej grafu G_{i-1}, która w grafie G_i ma krawędź do i. Oznaczmy zbiór $U_i = \{j : \ell(j) > i\}$. Łatwo sprawdzić, że wszystkie wierzchołki j, które są w grafie G połączone z którymkolwiek z wierzchołków $i, \ldots, n$, mają $\ell(j) \geqslant i$, więc należą do U_{i-1}. Zatem bezpośrednio przed i-tym krokiem algorytmu możemy na zawsze zapomnieć o wierzchołkach nienależących do U_{i-1} — już nie mogą być wykorzystane przy łączeniu składowych.

Mimo że do obliczenia indeksów $\ell(i)$ musimy znać całą permutację $a_1, \ldots, a_n$, zbiory U_i możemy wyliczać on-line. Zdefiniujmy $next(i) = \min\{a_{i+1}, \ldots, a_n\}$: jest to najmniejszy element, który nie występuje wśród $a_1, \ldots, a_i$. Z definicji $\ell(j) > i$ wtedy i tylko wtedy, gdy $a_j \geqslant next(i)$. Zatem $U_i = \{j : a_j \geqslant next(i)\}$.

Przypomnijmy nasz pomocniczy porządek na indeksach (wierzchołkach): $i \prec j$ oznacza, że $a_i < a_j$. Wierzchołek i jest w grafie G_i połączony z tymi wierzchołkami $j < i$, dla których $\ell(j) \geqslant i$ oraz $j \prec i$. Wybierzmy z każdej spójnej składowej grafu G_{i-1} ograniczonej do zbioru U_{i-1} minimalny w porządku $\prec$ wierzchołek j. Jeżeli $j \prec i$, to do lasu rozpinającego dodajemy krawędź (i, j). Jeżeli natomiast $j \succ i$, to rozpatrywana składowa grafu G_{i-1} nie jest połączona z i w grafie G_i. Wszystkie

[3] Można także zrealizować tę operację w zamortyzowanym czasie $O(\log n)$, ale za to bez konieczności użycia dodatkowych list. W tym celu, w operacji *delete* zaznaczamy jedynie, że wierzchołek j został usunięty, natomiast podczas wykonywania operacji *get_edge* usuwamy z początków przeglądanych list wszystkie nieaktualne nasłuchiwacze.

składowe grafu G_{i-1}, które połączyliśmy krawędzią z wierzchołkiem i, tworzą wraz z nim jedną spójną składową w grafie G_i.

Okazuje się, że spójne składowe grafu G_i ograniczone do zbioru U_i nie „przecinają się" w porządku $\prec$ — jeżeli C_1, C_2 są dwiema różnymi składowymi grafu G_i ograniczonymi do zbioru U_i, to albo wszystkie wierzchołki z C_1 są mniejsze od wszystkich z C_2 ($C_1 \prec C_2$), albo wszystkie wierzchołki z C_2 są mniejsze od wszystkich z C_1 ($C_2 \prec C_1$). Nie będziemy osobno dowodzić tej własności — algorytm, który pokażemy, po prostu utrzymuje taki niezmiennik.

Algorytm konstrukcji lasu rozpinającego on-line wygląda następująco. Na początku i-tego kroku na stosie S znajdują się spójne składowe grafu G_{i-1} ograniczone do zbioru U_{i-1}, uporządkowane malejąco względem $\prec$ (tzn. najmniejsza składowa jest na szczycie stosu). Algorytm zdejmuje ze stosu wszystkie ograniczone składowe, których najmniejszy w porządku $\prec$ wierzchołek jest mniejszy od i, łącząc te wierzchołki z i krawędziami lasu rozpinającego. Zbiory zdjęte ze stosu łączy wraz z wierzchołkiem i w jedną składową grafu G_i ograniczoną do zbioru U_{i-1}, którą następnie odkłada na stos. Wszystkie inne zbiory, które pozostały na stosie, są w całości większe od i, więc niezmiennik jest zachowany. Na koniec ze zbiorów na stosie usuwane są wszystkie wierzchołki nienależące do U_i. Zbiór U_i ma postać $\{j : j \succ M\}$, przy czym M to najmniejszy w porządku $\prec$ wierzchołek spośród $\{i+1, \ldots, n\}$, więc wystarczy usuwać najmniejszy (według $\prec$) element ze zbioru na szczycie stosu.

```
1:  next := 1;
2:  for i := 1 to n do begin
3:    current := {i};
4:    while (not S.empty) and (a[S.top.min] < a[i]) do begin
5:      add_edge(i, S.top.min);
6:      current := current ∪ S.top; S.pop;
7:    end
8:    S.push(current);
9:    while (not S.empty) and (a[S.top.min] = next) do begin
10:     S.top.extract_min;
11:     if S.top = ∅ then S.pop;
12:     next := next + 1;
13:   end
14: end
```

W algorytmie tym kluczowa jest efektywna implementacja następujących operacji na zbiorach przechowywanych na stosie S i w zmiennej *current*:

- Utwórz zbiór złożony z pojedynczego wierzchołka (operacja w wierszu 3).
- Połącz dwa zbiory w jeden (operacja w wierszu 6).
- Pobierz najmniejszy (według $\prec$) wierzchołek zbioru (operacja *min*).
- Usuń najmniejszy wierzchołek ze zbioru (operacja *extract_min*).

Są to operacje tak zwanej *kolejki złączalnej*. Przykładem struktury, która pozwala uzyskać ograniczenie $O(\log n)$ na czas wykonania każdej z nich, jest *drzewo lewicowe*.

Pojedyncze drzewo lewicowe reprezentuje jeden zbiór. Jest to pełne drzewo binarne, którego każdy węzeł wewnętrzny przechowuje jeden element zbioru oraz którego liście nie przechowują żadnej informacji (są sztucznie dodane dla czytelności opisu). Każde jego poddrzewo spełnia następujące własności:

(1) *Własność kopca*: element w korzeniu poddrzewa jest najmniejszy (w porządku $\prec$) ze wszystkich elementów w poddrzewie.

(2) Skrajnie prawa ścieżka poddrzewa (czyli ścieżka wychodząca z korzenia poddrzewa i przechodząca w każdym węźle do prawego dziecka) jest najkrótsza ze wszystkich ścieżek od korzenia do liścia (w tym poddrzewie).

Utworzenie jednoelementowego drzewa nie przedstawia żadnej trudności. Operacja *min* zwraca po prostu korzeń drzewa. Operację *extract_min* możemy sprowadzić do łączenia dwóch drzew: usuwamy korzeń i łączymy poddrzewa w jego dzieciach w jedno drzewo. Pozostaje do zrealizowania operacja połączenia dwóch drzew.

W każdym węźle w zapisujemy $length[w]$ — długość skrajnie prawej ścieżki poprowadzonej z w. Z własności (2) natychmiast wynika, że jest ona logarytmiczna względem rozmiaru poddrzewa o korzeniu w. Jeżeli w jest liściem, to $length[w] = 0$. Połączenie dwóch drzew lewicowych w jedno realizuje następująca funkcja rekurencyjna:

```
1: function union(u, w)
2: begin
3:    if w ≺ u then zamień u ↔ w;
4:    if right[u] jest liściem then right[u] := w else right[u] := union(right[u], w);
5:    if length[left[u]] < length[right[u]] then zamień left[u] ↔ right[u];
6:    length[u] := length[right[u]] + 1;
7:    return u;
8: end
```

Funkcja ta przyjmuje jako argumenty korzenie dwóch drzew lewicowych i zwraca korzeń nowego drzewa powstałego z ich połączenia. Liczba wywołań rekurencyjnych funkcji *union* dla pojedynczej operacji łączenia drzew jest ograniczona przez $length[u] + length[w] = O(\log n)$.

Uzyskany algorytm konstrukcji lasu rozpinającego działa w czasie $O(n \log n)$ i pamięci liniowej. Implementacja kolejki złączalnej za pomocą kopców dwumianowych (patrz [21]) dałaby taki sam czas działania. Opisane rozwiązanie znajduje się w pliku `kol2.cpp`.

Ulepszony algorytm on-line

Poprzednie rozwiązanie konstruuje las rozpinający graf G bez względu na to, czy graf ten jest dwukolorowalny. Jednak w przypadku, gdy graf G nie jest dwukolorowalny, jego las rozpinający nie jest nam w sumie potrzebny — możemy od razu odpowiedzieć `NIE`.

Załóżmy zatem, że graf G_{i-1} jest dwukolorowalny. Oczywiście, dwukolorowanie każdej spójnej składowej grafu G_{i-1} jest wyznaczone jednoznacznie (z dokładnością

do zamiany kolorów). Każdy zbiór występujący w algorytmie (na stosie S lub w zmiennej *current*) reprezentujemy za pomocą dwóch list: każda z nich zawiera wierzchołki jednego koloru uporządkowane zgodnie z relacją $\prec$ (z minimalnym na początku listy). Operacja utworzenia nowego zbioru $\{i\}$ wstawia i do jednej z list, a drugą listę pozostawia pustą. Operacja *min* pobiera mniejszy wierzchołek z początków obu list, zaś operacja *extract_min* usuwa ten wierzchołek. Natomiast operacja łączenia dwóch zbiorów w wierszu 6 dokleja listy reprezentujące zbiór $S.top$ na koniec list reprezentujących zbiór *current* w taki sposób, aby zachowany był rosnący porządek elementów w obu wynikowych listach. Pokażemy, że takie połączenie list jest możliwe, o ile graf G_i jest dwukolorowalny.

Niech j_1 będzie minimalnym wierzchołkiem w zbiorze $S.top$. Połączenie zbiorów *current* i $S.top$ następuje wtedy, gdy $j_1 \prec i$. Wszystkie wierzchołki zbioru *current* różne od i pochodzą z połączenia składowych zdjętych ze stosu S — są mniejsze od wszystkich wierzchołków w zbiorze $S.top$ (w szczególności od j_1), a więc także od i. Zatem i jest ostatnim elementem swojej listy[4]. Jeżeli w zbiorze $S.top$ druga lista (tzn. ta, która nie zaczyna się od j_1) jest pusta lub jej początkowy (minimalny) element j_2 jest większy od i, nie ma problemu: doklejamy tę listę do listy zawierającej i, a listę zaczynającą się elementem j_1 doklejamy do drugiej listy zbioru *current*. Jeżeli natomiast $j_1, j_2 \prec i$, to graf G_i nie jest dwukolorowalny — mamy w nim krawędzie (i, j_1) i (i, j_2), a więc w poprawnym dwukolorowaniu wierzchołki j_1, j_2 powinny otrzymać ten sam kolor, co nie jest możliwe, gdyż w dwukolorowaniu składowej $S.top$ dostały różne kolory. W takim przypadku przerywamy algorytm i odpowiadamy `NIE`.

W ten sposób wszystkie operacje na zbiorach wykonywane przez nasz algorytm udało nam się zaimplementować w czasie stałym. Otrzymaliśmy rozwiązanie zadania działające w czasie *liniowym*! Co więcej, jeżeli opisany algorytm kończy swoje działanie bez odpowiedzi `NIE`, to graf $G_n = G$ na pewno jest dwukolorowalny — nie musimy w ogóle sprawdzać, czy dwukolorowanie otrzymanego lasu rozpinającego daje poprawny algorytm sortowania permutacji.

Jeszcze jeden algorytm dwukolorowania

Spójrzmy na przedstawiony powyżej ulepszony algorytm on-line z innej strony. Obsługuje on kolejno przychodzące elementy permutacji $a_1, \ldots, a_n$, starając się przydzielać je do jednego z dwóch stosów. Element a_1 oczywiście można odłożyć na dowolny stos. Jeżeli w momencie przyjścia elementu a_i elementy na szczytach obu stosów są mniejsze, odpowiedzią jest `NIE`. Jeżeli element na szczycie jednego stosu jest mniejszy od a_i, a element na szczycie drugiego stosu jest większy od a_i, to trzeba odłożyć a_i na drugi stos. Natomiast jeżeli elementy na szczytach obu stosów są większe od a_i, wówczas nie wiadomo (bez dodatkowej informacji), na który stos należy odłożyć a_i. Algorytm on-line odkłada tę decyzję na później, gdyż potrzebna informacja może nadejść dopiero wraz z dalszymi elementami permutacji. Przedstawimy teraz algorytm, który na podstawie bardzo prostej informacji dodatkowej potrafi od razu stwierdzić, na który stos powinien odłożyć element a_i.

[4] Jednocześnie jest pierwszym (a więc jedynym) elementem swojej listy, gdyż znajduje się w niej od początku istnienia listy, a kolejne elementy mogą być dokładane tylko na koniec.

Dla każdego i niech $m(i)$ będzie indeksem maksymalnego elementu permutacji między i a $\ell(i)$ włącznie: $a_{m(i)} = \max\{a_i, a_{i+1}, \ldots, a_{\ell(i)}\}$. Algorytm wygląda następująco:

```
 1: S[1].push(∞); S[2].push(∞);
 2: next := 1;
 3: for i := 1 to n do begin
 4:   if (a[i] > S[1].top) and (a[i] > S[2].top) then return NIE;
 5:   least := min(S[1].top, S[2].top);
 6:   if S[1].top < S[2].top then greater_stack := 2 else greater_stack := 1;
 7:   j := i;
 8:   same := true; { czy i, j muszą znaleźć się na tym samym stosie? }
 9:   while (a[j] < least) and (m[j] ≠ j) do begin
10:     j := m[j];
11:     same := not same;
12:   end
13:   { j znajdzie się na stosie greater_stack }
14:   if (same) then k[i] := greater_stack else k[i] := 3 − greater_stack;
15:   S[k[i]].push(a[i]);
16:   { zdejmowanie ze stosów kolejnych elementów permutacji posortowanej: }
17:   while (S[1].top = next) or (S[2].top = next) do begin
18:     if S[1].top = next then S[1].pop else S[2].pop;
19:     next := next + 1;
20:   end
21: end
22: return TAK, k[1..n];
```

Przyjrzyjmy się pojedynczej iteracji pętli **for**. Zmienna *same* mówi nam, czy elementy o indeksach i, j muszą znaleźć się na tym samym stosie (wartość **true**), czy na różnych stosach (wartość **false**). Na początku $i = j$, więc *same* = **true**. Zauważmy, że jeżeli $m(j) \neq j$, to $j < m(j) \leqslant \ell(j)$ oraz $a_j < a_{m(j)}$, więc w grafie G istnieje krawędź $(j, m(j))$, a zatem elementy o indeksach $j, m[j]$ muszą znaleźć się na różnych stosach. Dlatego w pojedynczej iteracji pętli **while** zmienna *same* zmienia wartość na przeciwną. Są dwie możliwości zakończenia pętli **while** w wierszach 9-12:

- $a_j >$ *least*. Niech *least* $= a_r$. Pokażemy, że $r < j \leqslant \ell(r)$. Mamy $r < i \leqslant j$, gdyż zawsze $x \leqslant m(x)$. Jeśli $i = j$, oczywista jest również prawa nierówność — element a_r nie został jeszcze zdjęty z S. Gdy natomiast $j \neq i$, to $j = m(j')$ dla pewnego j' spełniającego warunki w wierszu 8. Wobec tego $a_{\ell(j')} \leqslant a_{j'} < a_r$, a więc $j = m(j') \leqslant \ell(j') \leqslant \ell(r)$. Ostatecznie otrzymujemy, że (j, r) jest krawędzią grafu G, czyli elementy o indeksach j, r muszą znaleźć się na różnych stosach. To oraz wartość zmiennej *same* jednoznacznie wyznacza numer stosu, na który należy odłożyć i-ty element (wiersz 14).

- $a_j <$ *least* i $m(j) = j$. Wówczas $a_j = \max\{a_i, \ldots, a_{\ell(j)}\}$, ponieważ dla dowolnego t zachodzi $a_{m^t(i)} = \max\{a_i, \ldots, a_{\ell(m^{t-1}(i))}\}$ (łatwa indukcja). Stąd dla każdego $h \in \{i, \ldots, \ell(j)\}$ zachodzi $\ell(h) \leqslant \ell(j)$. Zatem w grafie G nie ma krawędzi łączącej zbiór $\{i, \ldots, \ell(j)\}$ z $\{\ell(j) + 1, \ldots, n\}$. Co więcej, wszystkie $a_i, \ldots, a_{\ell(j)}$

są mniejsze od wszystkich elementów na stosach oraz, oczywiście, większe od wszystkich elementów już zdjętych ze stosów, wobec czego żadna krawędź nie opuszcza zbioru $\{i, \ldots, \ell(j)\}$. Możemy zatem odłożyć i-ty element na dowolny stos.

Powyższy algorytm, mając obliczone indeksy $m(i)$, działa w czasie liniowym. Jest tak, ponieważ dla każdego j przypisanie $j := m[j]$ w wierszu 10 jest wykonywane co najwyżej raz. Istotnie, jeżeli w pewnym kroku i pętli **for** wykonuje się to przypisanie (dla wybranego j), to w kolejnych krokach $i+1, \ldots, j$ co najmniej jeden z elementów $a_i, a_{m(i)}, \ldots, a_{m^{s-1}(i)}$ (gdzie $m^s(i) = j$), wszystkich mniejszych od a_j, znajduje się na stosie, więc pętla **while** zatrzyma się na warunku w wierszu 9, zanim dojdzie do podstawienia $j := m[j]$.

Wszystkie indeksy $m(i)$ można wyliczyć z definicji w czasie $O(n \log n)$ za pomocą takich samych drzew turniejowych, jakich używaliśmy w rozwiązywaniu wzorcowym do wyszukiwania krawędzi w przód. Znajdowanie maksimów na przedziałach off-line można też łatwo sprowadzić do problemu szukania najniższych wspólnych przodków w drzewie, który można rozwiązać w czasie $O(n \log^\star n)$ algorytmem Tarjana [21], a nawet w czasie liniowym. Opis tego sprowadzenia oraz rozwiązania liniowego można znaleźć w dwuczęściowym artykule poświęconym problemom RMQ i LCA, w numerach 9/2007 i 11/2007 czasopisma *Delta*[5].

Rozwiązania niepoprawne

Próba zachłannego rozwiązywania zadania — przydzielania elementów a_i do stosów na podstawie samego tylko początkowego fragmentu permutacji $a_1, \ldots, a_i$ — jest z góry skazana na niepowodzenie. Wystarczy rozważyć dwie następujące permutacje:

$$5, 2, 6, 1, 4, 3 \quad \text{oraz} \quad 5, 2, 4, 1, 6, 3.$$

Każdy poprawny algorytm sortowania odkłada elementy 5 i 2 na ten sam stos w przypadku pierwszej permutacji oraz na różne stosy w przypadku drugiej.

Błędne są również rozwiązania, które decyzję w i-tym kroku podejmują na podstawie samych tylko elementów $a_1, \ldots, a_{\ell(i)}$. Świadczą o tym na przykład takie dwie permutacje:

$$7, 2, 4, 1, 6, 3, 8, 5 \quad \text{oraz} \quad 7, 2, 4, 1, 8, 3, 6, 5.$$

W obu $\ell(2) = 4$ oraz początkowe fragmenty a_1, a_2, a_3, a_4 są identyczne. Mimo to każdy poprawny algorytm sortowania odkłada elementy 7 i 2 na ten sam stos w pierwszym przypadku oraz na różne stosy w drugim przypadku.

Testy

Programy nadesłane przez zawodników były sprawdzane na 10 grupach testów. Każda z tych grup składa się z trzech testów: losowego testu typu a z odpowiedzią pozytywną, losowego testu typu b z odpowiedzią negatywną oraz testu typu c z odpowiedzią

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

pozytywną, który odrzuca niepoprawne rozwiązania drugiego typu opisanego powyżej (wyłączając test *1c*, reprezentujący przypadek brzegowy). Oto zestawienie wielkości danych wejściowych w poszczególnych testach:

Nazwa	n
kol1a.in	19
kol1b.in	15
kol1c.in	1
kol2a.in	60
kol2b.in	60
kol2c.in	60
kol3a.in	286
kol3b.in	268
kol3c.in	290

Nazwa	n
kol4a.in	813
kol4b.in	770
kol4c.in	813
kol5a.in	12 196
kol5b.in	12 155
kol5c.in	12 000
kol6a.in	27 193
kol6b.in	27 122
kol6c.in	25 000
kol7a.in	49 178
kol7b.in	49 126

Nazwa	n
kol7c.in	50 000
kol8a.in	69 281
kol8b.in	69 187
kol8c.in	70 000
kol9a.in	95 398
kol9b.in	95 402
kol9c.in	90 000
kol10a.in	99 838
kol10b.in	96 051
kol10c.in	99 000

Brutalne (wykładnicze) rozwiązania mogły zaliczyć tylko testy z grup 1 i 2, przy czym rozwiązanie sprawdzające wszystkie 2^n możliwości zaliczało tylko testy z grupy 1. Testy z grup 3 i 4 były dostępne dla rozwiązań działających w czasie $O(n^2)$. Testy z grupy 5 przechodziło rozwiązanie konstruujące cały graf G w czasie $O((n+m)\log n)$, gdzie m jest liczbą krawędzi w grafie G. Natomiast testy w grupach 5-10 zostały tak dobrane, aby graf G miał rozmiar kwadratowy, i wobec tego nie zaliczały ich rozwiązania oparte na generowaniu całego grafu G. Przewidziane przez organizatorów rozwiązania niepoprawne nie przechodziły żadnej grupy testów.