Michał Włodarczyk
Treść zadania, Opracowanie

Wojciech Nadara
Program

Dostępna pamięć: 256 MB. *OI, etap I, 19.10–16.11.2015*

Hydrorozgrywka

Ekipy hydrauliczne Bajtazara i Bajtoniego wygrały przetarg na doprowadzenie wody do Bajtołów Dolnych. Sieć drogowa w tym mieście składa się z n skrzyżowań połączonych m odcinkami dróg. Z każdego skrzyżowania można dojechać do każdego innego skrzyżowania za pomocą sieci dróg. Pod każdym odcinkiem drogi należy zakopać rurę wodociągową.

Aby urozmaicić sobie pracę, Bajtazar i Bajtoni postanowili zagrać w grę. Na początek ekipy obu bohaterów stają na jednym ze skrzyżowań. Przez całą grę obie ekipy będą podążać razem. Gracze wykonują ruchy na przemian, począwszy od Bajtazara. W swoim ruchu gracz wskazuje swojej ekipie odcinek drogi (pod którym jeszcze nie ma rury) wychodzący ze skrzyżowania, na którym znajdują się obie ekipy. Ekipa gracza zakopuje rurę pod tym odcinkiem drogi i następnie obie ekipy przemieszczają się do drugiego ze skrzyżowań, które łączy ten odcinek.

Gracz, który nie może wykonać ruchu, przegrywa i za karę jego ekipa musi zakopać rury pod pozostałymi odcinkami dróg. Bajtazar zastanawia się, od którego skrzyżowania może zacząć się gra, aby był w stanie wygrać niezależnie od ruchów Bajtoniego. Poprosił Cię o pomoc w ustaleniu listy takich skrzyżowań. Dodatkowo zauważył, że sieć drogowa w Bajtołach ma ciekawą własność: **wyjeżdżając ze środka dowolnego odcinka drogi, na dokładnie jeden sposób możemy zrobić „pętlę" i wrócić do punktu wyjścia, jeśli nigdy nie zawracamy i nie odwiedzamy żadnego skrzyżowania dwukrotnie**.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n i m oddzielone pojedynczym odstępem, oznaczające liczbę skrzyżowań i liczbę odcinków dróg. Skrzyżowania numerujemy liczbami od 1 do n. Kolejne m wierszy opisuje sieć drogową: każdy z nich zawiera dwie liczby całkowite a, b ($1 \leqslant a, b \leqslant n$, $a \neq b$) oddzielone pojedynczym odstępem, oznaczające, że skrzyżowania o numerach a i b są połączone odcinkiem drogi. Możesz założyć, że żadne dwa skrzyżowania nie są połączone więcej niż jednym odcinkiem drogi.

Wyjście

Na standardowe wyjście należy wypisać dokładnie n wierszy: i-ty z nich ma zawierać liczbę 1, jeśli Bajtazar może wygrać, gdy gra zacznie się ze skrzyżowania numer i; w przeciwnym wypadku ma zawierać liczbę 2.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`6 7`	`1`
`1 2`	`1`
`2 3`	`1`
`3 1`	`2`
`3 4`	`1`
`4 5`	`2`
`5 6`	
`6 3`	

Testy „ocen”:

`1ocen`: *$n = 9$, $m = 12$, sieć drogowa składa się z czterech „pętli”: 1–2–3–1, 1–4–5–1, 1–6–7–1 oraz 1–8–9–1.*

`2ocen`: *$n = 998$, $m = 999$, dla każdego j takiego, że $1 \leqslant j < n$, istnieje odcinek drogi pomiędzy j-tym i $(j + 1)$-wszym skrzyżowaniem; istnieją również odcinki drogi łączące skrzyżowania 1 z 499 oraz 499 z 998.*

`3ocen`: *$n = 500\,000$, $m = 500\,000$, sieć dróg tworzy jedną „pętlę”.*

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$3 \leqslant n, m \leqslant 20$	*21*
2	$3 \leqslant n, m \leqslant 1000$	*39*
3	$3 \leqslant n, m \leqslant 500\,000$	*40*

Rozwiązanie

Hydrorozgrywka to jedno z tych zadań, które trafiają na I etap Olimpiady głównie dlatego, że wymyślenie rozwiązania i ominięcie wszystkich pułapek w implementacji w ciągu zaledwie pięciu godzin wymagałoby nadludzkich umiejętności. Z drugiej strony część doświadczonych zawodników umie uporać się ze standardowymi zadaniami z I etapu w kilka dni i wielką stratą byłoby zmarnować ich chęci do spędzenia tygodnia walki z bardziej wymagającym przeciwnikiem. Autor podpisuje się pod tezą, że najlepszym sposobem treningu w takich dziedzinach jak algorytmika jest podejmowanie *nieco za trudnych* wyzwań.

Na początek zinterpretujmy grę w układanie rur w języku teorii grafów. Zaczynając od ustalonego wierzchołka v_0, gracze na przemian wybierają krawędzie tak, aby każda kolejna miała wspólny wierzchołek z poprzednią. Każda krawędź może zostać wybrana co najwyżej raz. Innymi słowy, gracze w trakcie rozgrywki budują *marszrutę*

w grafie. Krawędzie na marszrucie nie powtarzają się, jednak ten sam wierzchołek może na niej wystąpić wielokrotnie. Gracz, który nie może wykonać poprawnego ruchu, przegrywa.

Tak zdefiniowana gra jest skończona, ponieważ liczba ruchów jest ograniczona przez liczbę krawędzi w grafie. Liczba stanów tej gry jest wykładnicza ze względu na liczbę krawędzi. Stany w grze zadane są przez marszruty zaczynające się w wierzchołku v_0, natomiast każde przejście pomiędzy stanami odpowiada wydłużeniu marszruty o jedną krawędź. Łatwo zauważyć, że taka gra jest *zdeterminowana*, tzn. dla każdego stanu jeden z graczy może doprowadzić do zwycięstwa, niezależnie od tego, jakie ruchy będzie wykonywał jego rywal. Mówimy, że gracz X posiada *strategię wygrywającą*, jeśli powyższy warunek zachodzi dla X w stanie początkowym.

Najprostszy algorytm obliczający strategie wygrywające przegląda cały graf stanów gry i klasyfikuje stany na wygrywające i przegrywające. Jest to zazwyczaj mało praktyczne podejście z powodu ogromnej liczby stanów do przeanalizowania. Częściowe rozwiązanie tego problemu opiera się na obserwacji, że niektóre stany gry są równoważne. Zauważmy, że informacja o tym, które ruchy będzie można wykonać w przyszłości, jest w całości zakodowana przez zbiór dotąd wykorzystanych krawędzi oraz aktualny końcowy wierzchołek marszruty. Możemy zatem rozważać stany opisane przez podzbiór krawędzi grafu i jeden wierzchołek. Aby sprawdzić, czy pierwszy gracz ma strategię wygrywającą w grze rozpoczynającej się w wierzchołku v_0, wystarczy sprawdzić wartość obliczoną dla stanu $(\emptyset, v_0)$. Liczba stanów zostaje w ten sposób ograniczona do $O(n2^m)$ i tyle też wynosi złożoność pamięciowa naiwnego algorytmu. Rozwiązanie oparte na tym podejściu (zaimplementowane w pliku `hyds1.cpp`) działa w czasie $O(m2^m)$. Na zawodach było ono warte 21 punktów. Nie korzysta ono jednak w żaden sposób ze specjalnej struktury grafu.

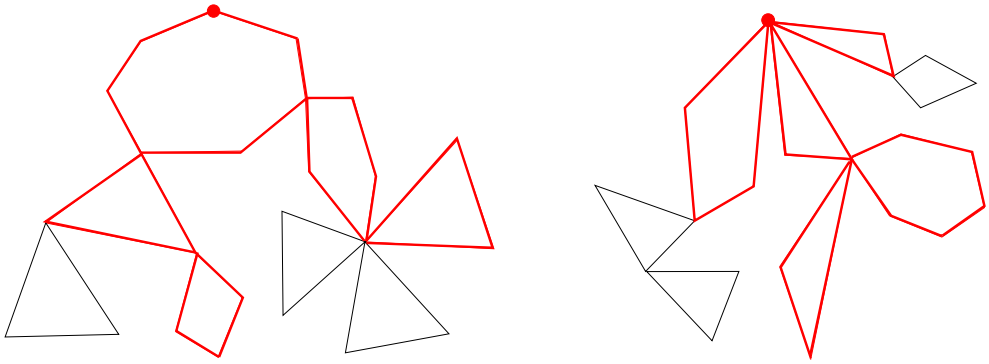
Cykle, marszruty i kaktusy

Poza gwarancją spójności grafu w treści zadania pojawia się pozornie niewiele mówiący warunek: *wyjeżdżając ze środka dowolnego odcinka drogi, na dokładnie jeden sposób możemy zrobić „pętlę" i wrócić do punktu wyjścia, jeśli nigdy nie zawracamy i nie odwiedzamy żadnego skrzyżowania dwukrotnie.* Równoważnie możemy napisać, że każda krawędź leży na dokładnie jednym *cyklu prostym*, gdzie cykl prosty to cykl, który przechodzi przez każdy wierzchołek co najwyżej jednokrotnie. Jako że często będziemy używać tego pojęcia, zamiast *cykl prosty* pisać będziemy po prostu *cykl*.

Grafy spełniające warunek z zadania będziemy nazywać *kaktusami* ze względu na podobieństwo ich graficznych reprezentacji do krzewu opuncji. Jako że rysunki potrafią czasem prowadzić do mylnych intuicji, udowodnijmy formalnie kilka własności kaktusów.

Po pierwsze, jeśli marszruta prowadzi po cyklu C, następnie opuszcza go w wierzchołku v_i, po czym wraca na C, wchodząc do wierzchołka v_j, to $v_i = v_j$.

Lemat 1. Rozważmy marszrutę $(v_i)_{i=0}^{k}$ w kaktusie. Ustalmy dowolny cykl prosty C. Jeśli $v_i \in C$, $v_{i+1} \notin C$, a v_j jest następnym po v_i wierzchołkiem należącym do C, to $v_i = v_j$.



Rys. 1: Przykłady kaktusów z wyróżnioną potencjalną marszrutą gry.

Dowód: Przypuśćmy $v_i \neq v_j$. Na podstawie cyklu C konstruujemy cykl prosty C', zastępując odcinek C pomiędzy v_i oraz v_j przez ścieżkę $v_i, v_{i+1}, \ldots, v_j$ z usuniętymi pętlami. Otrzymujemy dwa różne cykle proste C i C', które mają co najmniej jedną wspólną krawędź. Dla tej krawędzi nie jest spełniony warunek z zadania, który definiuje kaktus. ■

Lemat 2. Cykle proste kaktusa są krawędziowo rozłączne.

Dowód: Przypuśćmy, że dwa różne cykle proste C_1, C_2 posiadają wspólną krawędź. Zatem istnieje ścieżka łącząca dwa wierzchołki v_i i v_j należące do cyklu C_1, składająca się z krawędzi cyklu C_2, które nie należą do C_1. Gdyby $v_i = v_j$, to ponieważ cykl C_2 jest prosty, ścieżka ta przebiegałaby po całym cyklu C_2. Wówczas jednak C_2 nie miałby krawędzi wspólnych z C_1. Z kolei gdy $v_i \neq v_j$, otrzymujemy ścieżkę przeczącą tezie lematu 1. Zatem w obydwóch przypadkach otrzymujemy sprzeczność. ■

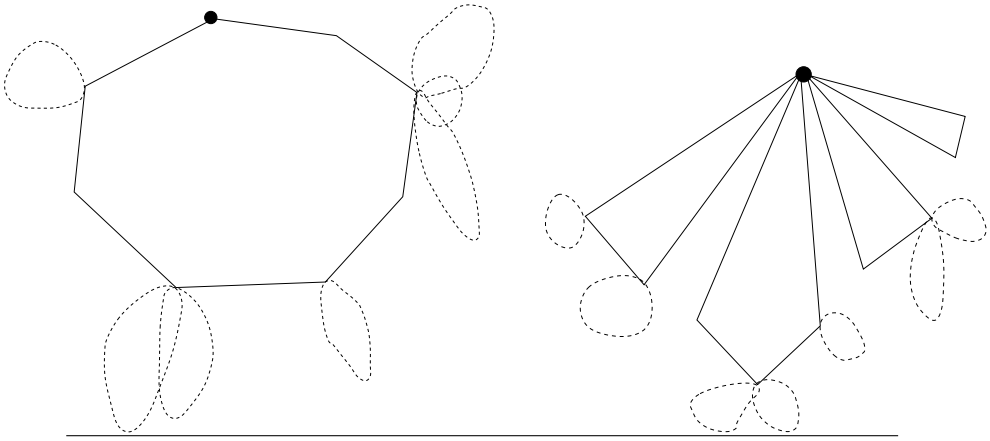
Lemat 3. Wszystkie wierzchołki kaktusa mają parzysty stopień.

Dowód: Indukcja po liczbie cykli prostych w grafie. Kaktus składający się tylko z jednego cyklu oczywiście spełnia tezę lematu. Przypuśćmy zatem, że spełniają ją kaktusy o liczbie cykli mniejszej od k, i rozważmy kaktus o k cyklach. Usuńmy z niego krawędzie dowolnego cyklu. W ten sposób stopnie niektórych wierzchołków zmniejszą się o 2, co nie wpływa na ich parzystość.

Liczba cykli przechodzących przez dowolną zachowaną krawędź na pewno nie wzrosła, a istniejący cykl nie posiadał części wspólnej z usuniętym cyklem na mocy lematu 2. Powstały graf, choć potencjalnie niespójny, nadal jest kaktusem i, z założenia indukcyjnego, wszystkie jego wierzchołki mają parzysty stopień. Zatem po przywróceniu usuniętego cyklu wszystkie wierzchołki mają nadal parzysty stopień. ■

Lemat 4. Każda rozgrywka na kaktusie kończy się w wierzchołku początkowym (patrz rys. 1).

Dowód: Oznaczmy wierzchołek początkowy przez v_0. Niech v będzie wierzchołkiem, po dojściu do którego gracz nie może wykonać ruchu. Jeśli $v \neq v_0$, to każde przejście przez wierzchołek v „zużywa” dwie krawędzie incydentne z v. Z lematu 3 wiemy, że



Rys. 2: Po lewej: kaktus prosty z zaznaczonymi przerywaną linią wewnętrznymi pękami. Po prawej: pęk składający się z czterech kaktusów prostych.

stopień v jest parzysty. Zatem tuż po wejściu do wierzchołka v mamy do wyboru nieparzystą (a więc niezerową) liczbę krawędzi incydentnych z v. ■

Lematy 1 oraz 4 charakteryzują marszruty, jakie mogą pojawić się w trakcie gry. Idąc po cyklu zawierającym v_0, gracze mogą zdecydować, czy przejść do innego cyklu. Tam rozpoczyna się wewnętrzna rozgrywka, która (z lematu 1) skończy się w tym samym punkcie. Jedyną istotną informację stanowi to, który z graczy będzie wykonywał ruch po powrocie na pierwotny cykl.

Jak się dobrać do kaktusa?

Aby móc wykorzystać specjalną strukturę grafu, musimy umieć reprezentować kaktusy w wygodny sposób; patrz rys. 2.

Definicja 1. Kaktusem prostym ukorzenionym w wierzchołku v_0 nazwiemy kaktus z wyróżnionym wierzchołkiem v_0 (korzeniem) o stopniu 2.

Definicja 2. Pękiem nazwiemy rodzinę kaktusów prostych ukorzenionych w tym samym wierzchołku v_0. Kaktusy tworzące pęk są rozłączne poza posiadaniem wspólnego korzenia. W szczególności, pęk może składać się z jednego kaktusa prostego.

Każdy kaktus prosty możemy reprezentować rekurencyjnie jako cykl przechodzący przez v_0, zawierający informacje o pękach ukorzenionych w wierzchołkach cyklu. W ten sposób otrzymujemy drzewiastą strukturę danych dla kaktusa ukorzenionego w konkretnym wierzchołku. Poniżej przedstawiamy pseudokod zmodyfikowanej procedury *DFS* (*Depth First Search*), tłumaczącej standardową reprezentację kaktusa przy pomocy list sąsiedztwa na postać rekurencyjną. Jako argumenty przekazywane są numer analizowanego i poprzednio odwiedzonego wierzchołka, zwanego *rodzicem*. W tablicy dynamicznej *stack* trzymamy stos odwiedzonych wierzchołków i kiedy trafimy do wierzchołka znajdującego się już na stosie, zapamiętujemy znaleziony cykl.

Lista *cycles*[v] zawiera pęk kaktusów ukorzenionych w v, natomiast w tablicy *belong* zapamiętujemy dla każdego wierzchołka, przez który cykl można do niego trafić, idąc od korzenia. Nie używamy tej informacji bezpośrednio w omówieniu, jednak jest ona przydatna w implementacji.

```
 1: function DFS(v, parent)
 2: begin
 3:   stack.push(v);
 4:   onStack[v] := true;
 5:   visited[v] := true;
 6:   for w ∈ neighbors[v] do begin
 7:     if onStack[w] and (w ≠ parent) then begin
 8:       C := new Cycle;
 9:       C.insert(w);
10:       i := stack.size() − 1;
11:       while stack[i] ≠ w do begin
12:         C.insert(stack[i]);
13:         belong[stack[i]] := C;
14:         i := i − 1;
15:       end
16:       cycles[w].insert(C);
17:     end
18:     if not visited[w] then
19:       DFS(w, v);
20:   end
21:   stack.pop(v);
22:   onStack[v] := false;
23: end
```

Należy zaznaczyć, że korzeń DFS-a, tzn. wierzchołek, od którego zaczęliśmy przeszukiwanie grafu, pozostanie z pustym polem w tablicy *belong* i należy rozważyć go jako przypadek szczególny. Można tego uniknąć, reprezentując graf jako kaktus prosty, jeśli rozpoczniemy przeszukiwanie grafu w wierzchołku stopnia 2 (zachęcamy Czytelnika do udowodnienia, że taki wierzchołek istnieje w każdym kaktusie) i przypiszemy go do jedynego cyklu, który z niego wyrasta.

Klasyfikacja kaktusów

W wielu teoriach matematycznych centralne miejsce zajmują twierdzenia o klasyfikacji, rozstrzygające, które obiekty możemy traktować jako równoważne, a pomiędzy którymi zachodzą istotne strukturalne różnice. Takie twierdzenia przydadzą się nam, aby uprościć opis rozwiązania wzorcowego.

Gracza rozpoczynającego rozgrywkę w interesującym nas podkaktusie nazwiemy graczem I, a jego rywala – graczem II. Pęki podzielimy na wygrywające i przegrywające, w zależności od tego, czy gracz I może zagwarantować sobie zebranie ostatniej krawędzi w grze ograniczonej do rozważanego pęku. W przypadku kaktusów prostych sytuacja jest (wbrew nazwie) bardziej skomplikowana, ponieważ niekiedy graczowi I

opłaca się zmusić rywala do wzięcia ostatniej krawędzi, aby samemu kontynuować grę w korzystnym dla siebie stanie. Dlatego wprowadzimy dwa rodzaje gier na kaktusach prostych: rodzaj A, w którym wygrywa gracz zabierający ostatnią krawędź, oraz rodzaj B, w którym taki gracz przegrywa. W zależności od istnienia strategii wygrywających w tych grach, kaktus prosty może należeć do jednego z czterech typów: **00**, **10**, **01**, **11**, gdzie kolejne bity kodują odpowiednio wynik gry A oraz B – naturalnie 1 oznacza zwycięstwo[1]. Przykładowo, typ **10** oznacza, że w grze rodzaju A gracz I ma strategię wygrywającą, a w grze rodzaju B to gracz II ma strategię wygrywającą. Zauważmy, że dla pęków rozważamy tylko grę rodzaju A i takiej gry dotyczy oryginalne pytanie z zadania.

Twierdzenie 1. *Rozważmy kaktus prosty ukorzeniony w v_0. Niech C będzie cyklem zawierającym v_0. Wówczas kaktus jest typu:*

10, *jeśli na cyklu C nie ma wygrywających pęków oraz C jest nieparzystej długości,*

01, *jeśli na cyklu C nie ma wygrywających pęków oraz C jest parzystej długości,*

11, *jeśli wychodząc z v_0 w pewną stronę, dojdziemy do wygrywającego pęku po parzystej liczbie kroków,*

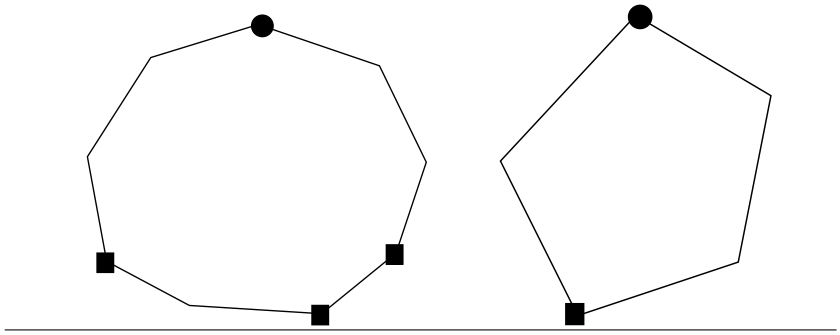
00, *jeśli wychodząc z v_0 w dowolną stronę, dojdziemy do wygrywającego pęku po nieparzystej liczbie kroków.*

Dowód: Załóżmy wpierw, że na cyklu C nie znajduje się żaden wygrywający pęk. W przypadku, gdy C jest nieparzystej długości, zaznaczmy na C krawędzie, których odległość od v_0 jest parzysta. W ten sposób zaznaczone zostaną obydwie krawędzie incydentne z v_0, a poza nimi co druga krawędź na C. Gracz I może zagwarantować sobie wzięcie wszystkich zaznaczonych krawędzi. Gracz II może potencjalnie starać się mu przeszkodzić i zamiast zebrać niezaznaczoną krawędź na C, rozpocząć wewnętrzną rozgrywkę na pęku ukorzenionym w pewnym wierzchołku v. Aby zmusić gracza I do wybrania niezaznaczonej krawędzi, gracz II musi doprowadzić do sytuacji, w której gracz I będzie miał ruch w v i wszystkie kaktusy proste wyrastające z v będą już odwiedzone. Jednak aby do tego doprowadzić, gracz II musiałby mieć strategię wygrywającą w pęku ukorzenionym w v.

Z drugiej strony, gdyby gracz I chciał zmienić kolejność ruchów i zmusić gracza II do wzięcia ostatniej krawędzi, potrzebowałby strategii wygrywającej na jednym z pęków. Zatem również gracz II jest w stanie zagwarantować, że gracz I wykona ostatni ruch na cyklu C. Analogicznie, jeżeli cykl C jest parzystej długości, gracz II potrafi zagwarantować sobie jedynie wzięcie ostatniej krawędzi, natomiast gracz I – przedostatniej. W ten sposób udało nam się sklasyfikować dwa pierwsze przypadki.

W dwóch pozostałych przypadkach w co najmniej jednym wierzchołku cyklu C ukorzeniony jest wygrywający pęk. Gracz, który jako pierwszy będzie wykonywał ruch w takim wierzchołku – oznaczmy ten wierzchołek przez v – jest w stanie zdeterminować dalszy przebieg rozgrywki na cyklu. Jako że jedna z krawędzi idących do v jest wykorzystana, o wyniku rozgrywki decyduje, który z graczy opuści wierzchołek v.

[1] W filmowym omówieniu zadania pojawiły się alternatywne definicje typów kaktusów. Kaktusy typu 01, 10 to takie, w których gracz zabierający ostatnią krawędź jest zdeterminowany, zaś typy 11, 00 odpowiadają kaktusom, w których o finale rozgrywki decyduje odpowiednio gracz I lub II. Po przemyśleniu sprawy, autor uważa, że obecna notacja jest prostsza i bardziej precyzyjna.



Rys. 3: Ilustracja do twierdzenia 1: korzeń zaznaczono kółkiem, a wierzchołki zawierające wygrywające pęki – kwadratami. Po lewej stronie kaktus typu **00**, po prawej **11**.

Jeśli ruch ten prowadzi do stanu ze strategią wygrywającą w grze A bądź B, gracz wykonujący ruch może ten ruch wykonać od razu. W przeciwnym przypadku, może rozpocząć grę na wygrywającym pęku i zagwarantować sobie wzięcie ostatniej krawędzi. Drugi gracz może próbować opuścić grę na pęku przed odwiedzeniem wszystkich wewnętrznych kaktusów, lecz w ten sposób również zbierze niechcianą krawędź.

Załóżmy, że wychodząc z v_0 w pewną stronę, można w parzystej liczbie kroków dojść do korzenia wygrywającego pęku. Oznaczmy ten korzeń przez v. Gracz I rozpocznie grę w tym właśnie kierunku i zagwarantuje sobie wzięcie krawędzi o nieparzystych indeksach, co zapewni mu ruch w wierzchołku v i zwycięstwo w obu rodzajach gry. Jeżeli zaś idąc w obu kierunkach, pierwszy wygrywający pęk znajdziemy po nieparzystej liczbie kroków, to niezależnie od początkowego ruchu, gracz II zagwarantuje sobie wzięcie krawędzi o parzystych indeksach i to on będzie miał ruch przy wygrywającym pęku. Powyższa obserwacja kończy analizę ostatnich dwóch przypadków. ■

Wynik gry na pęku zależy tylko od liczby kaktusów poszczególnych typów. Czwórkę liczb $(x_{00}, x_{10}, x_{01}, x_{11})$ opisującą, ile kaktusów typu odpowiednio **00**, **10**, **01**, **11** znajduje się w pęku, nazwiemy *konfiguracją*. Naturalnym pomysłem jest wykorzystanie programowania dynamicznego do pogrupowania konfiguracji na wygrywające i przegrywające. Aby rozstrzygnąć, jakiego typu jest dana konfiguracja, można rozważyć wszystkie ruchy gracza I, polegające na wyborze początkowego kaktusa i rodzaju wewnętrznej gry (A lub B), i przyjrzeć się wcześniej obliczonym wynikom gier dla konfiguracji z pewnym x_i pomniejszonym o 1. Strategią wygrywającą może być rozpoczęcie gry A w kaktusie gwarantującym zwycięstwo, jeśli prowadzi to do przegrywającej konfiguracji, lub podjęcie gry B, kiedy otrzymana konfiguracja zapewnia zwycięstwo. Niestety liczba konfiguracji jest rzędu $\Theta(n^4)$, co wymusza na nas znalezienie sprytniejszego sposobu klasyfikacji pęków. Takiego sposobu dostarcza poniższe kryterium.

Twierdzenie 2. *Pęk jest wygrywający wtedy i tylko wtedy, gdy zawiera kaktus prosty typu* **11** *lub liczba kaktusów typu* **10** *jest nieparzysta.*

Dowód: Indukcja po liczbie kaktusów w pęku. Konfiguracja (0,0,0,0) jest zgodnie z definicją przegrywająca, ponieważ gracz I nie może zagwarantować sobie zabrania

ostatniej krawędzi. Sytuacja jest równoważna wzięciu ostatniej krawędzi przez gracza II, ponieważ wewnętrzna gra na pęku jest zakończona i ruch należy do gracza I. Załóżmy teraz, że rozważamy konfigurację $(x_{00}, x_{10}, x_{01}, x_{11})$ i teza indukcyjna zachodzi dla wszystkich pęków składających się z mniej niż $x_{00}+x_{10}+x_{01}+x_{11}$ kaktusów.

Jeżeli pęk zawiera kaktus typu **11**, to gracz I może na nim zagrać zarówno tak, by wziąć ostatnią krawędź, jak i tak, by ostatnią krawędź wziął gracz II. W zależności od tego, czy konfiguracja $(x_{00}, x_{10}, x_{01}, x_{11}-1)$ jest wygrywająca, czy przegrywająca, wybierze on odpowiednią strategię i w ten sposób zapewni sobie zwycięstwo. Jeśli zaś $x_{11}=0$, ale x_{10} jest nieparzyste, strategią gracza I jest takie granie na kaktusie typu **10**, by wziąć ostatnią krawędź. Wówczas gracz II będzie zmuszony wykonać ruch w konfiguracji $(x_{00}, x_{10}-1, x_{01}, 0)$. Jednak zgodnie z założeniem indukcyjnym taka konfiguracja jest przegrywająca.

Pozostaje nam pokazać, że pozostałe konfiguracje są przegrywające. Przypuśćmy $x_{11}=0$ oraz $2 \mid x_{10}$. Jeśli gra rozpocznie się w kaktusie typu **10**, gracz II zadba o to, by gracz I wziął ostatnią krawędź. Prowadzi to do konfiguracji, w której x_{10} jest nieparzyste i rozpoczyna gracz II, co gwarantuje mu zwycięstwo. Jeśli zaś wybrany zostanie kaktus typu **00** lub **01**, wówczas gracz II może sprawić, że weźmie on ostatnią krawędź, przez co gracz I znajdzie się w konfiguracji $(x_{00}-1, x_{10}, x_{01}, 0)$ lub, odpowiednio, $(x_{00}, x_{10}, x_{01}-1, 0)$, która zgodnie z tezą indukcyjną jest przegrywająca. ■

Uzbrojeni w reprezentację cyklową grafu, możemy z łatwością rekurencyjnie sklasyfikować wszystkie pęki i kaktusy proste. Twierdzenia 1 i 2 pozwalają nam wykonać wszystkie obliczenia w złożoności obliczeniowej $O(m)$ i odczytać, kto posiada strategię wygrywającą w grze rozpoczynającej się w korzeniu kaktusa. To jednak nie koniec zadania – wszak w treści jesteśmy proszeni o rozstrzygnięcie wyniku gry dla wszystkich możliwych wierzchołków początkowych. Możemy obliczyć całą dekompozycję cyklową dla wszystkich możliwych wierzchołków w grafie, lecz takie rozwiązanie obarczone jest złożonością obliczeniową[2] $O(nm)$. Za zaimplementowanie tego algorytmu można było otrzymać 60 punktów.

Rozwiązanie wzorcowe

Kluczem do przyspieszenia obliczeń jest dogłębne wykorzystanie raz obliczonej dekompozycji cyklowej. Przypomnijmy, że ma ona rekurencyjną strukturę drzewiastą. Kaktus przedstawiamy jako cykl, w którego wierzchołkach zaczepione są inne kaktusy proste (być może wiele w tym samym wierzchołku). Każdy taki kaktus prosty będziemy nazywać *podkaktusem* przez analogię do poddrzewa. Za pomocą twierdzeń 1 i 2 umiemy rozstrzygnąć, jak potoczy się gra rozpoczynająca się w korzeniu v pewnego podkaktusa, o ile znamy typy podkaktusów poniżej v. Ostatecznie w ten sposób otrzymujemy wynik dla wierzchołka startowego v_0, w którym ukorzeniony jest cały kaktus.

Obliczenie wyniku gry rozpoczynającej się gdzie indziej niż w v_0 jest nieco trudniejsze, bo musimy wziąć pod uwagę kaktus, który zawiera v_0. Ustalmy wierzchołek

[2] W istocie dla każdego kaktusa zachodzi $m \leqslant \frac{3}{2}n$ (zachęcamy Czytelnika do przekonania się o tym osobiście), zatem złożoność obliczeniową tego rozwiązania można oszacować jako $O(n^2)$.

$v \neq v_0$ będący korzeniem pewnego podkaktusa. Aby rozstrzygnąć, jak potoczy się gra rozpoczynająca się w v, musimy znać typy wszystkich kaktusów ukorzenionych w v, jak również typ „nadkaktusa" v. Mówiąc formalnie, *nadkaktus* v powstaje przez usunięcie podkaktusów ukorzenionych w v (z wyjątkiem samego wierzchołka v) i następnie ukorzenienie powstałego grafu właśnie w v.

Potrzebujemy zatem wykonać dwa dodatkowe kroki: ustalić typy wszystkich nadkaktusów oraz wykorzystać te informacje do obliczenia ostatecznej odpowiedzi. Algorytm zaprezentowany został na poniższym pseudokodzie. Procedura *analyze* zostaje wywołana rekurencyjnie dla każdego podkaktusa podanego grafu. Pierwszy argument to cykl C zawierający korzeń podkaktusa (w implementacji może to być numer lub wskaźnik do obiektu). Cykl reprezentowany jest jako tablica wierzchołków, gdzie $C[0]$ to korzeń podkaktusa. Drugi argument procedury określa typ nadkaktusa (**00**, **10**, **01** lub **11**). W pierwszym wywołaniu C to cykl zaczynający się od v_0, a $overcactusType = \mathbf{00}$.

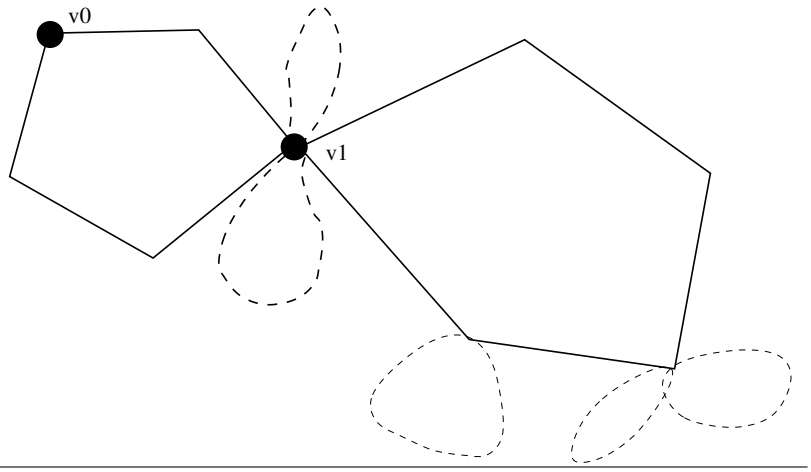
```
 1: function analyze(C, overcactusType)
 2: begin
 3:   externalGame[C[0]] := getGameResult(cycles[C[0]] \ C, overcactusType);
 4:   for i := 1 to C.size() − 1 do
 5:     externalGame[C[i]] := getGameResult(cycles[C[i]], 00);
 6:   cycleType := getCycleTypes(C, externalGame);
 7:   for i := 1 to C.size() − 1 do begin
 8:     result[C[i]] := getGameResult(cycles[C[i]], cycleType[C[i]]);
 9:     foreach C' ∈ cycles[C[i]] do
10:       analyze(C', cycleType[C[i]]);
11:   end
12: end
```

Pierwszy krok (wiersze 3-5) to wypełnienie tablicy *externalGame*, w której dla każdego wierzchołka $v \in C$ zapisujemy wynik gry rozpoczynającej się w v, przy założeniu, że z grafu usunęliśmy krawędzie cyklu C. Innymi słowy, dla każdego v rozpatrujemy jedynie kaktusy *wyrastające* z v.

Korzystamy tu z funkcji *getGameResult*, która oblicza (na podstawie twierdzenia 2), czy dany pęk jest wygrywający. Pierwszy argument funkcji to zbiór cykli stanowiących wyjściowe cykle wszystkich kaktusów w pęku poza jednym. Drugi argument to typ ostatniego kaktusa w pęku. W wierszu 5 nie potrzebujemy uwzględniać typu żadnego dodatkowego kaktusa, dlatego jako argument przekazujemy typ **00**, który nie wpływa na wynik.

W drugim kroku (wiersz 6), dla każdego $v \in C$ rozpatrujemy graf powstały przez usunięcie wszystkich kaktusów wyrastających z v. Wynik gry rozpoczynającej się w v zapisujemy w $cycleType[v]$. Za wykonanie tego kroku odpowiada funkcja *getCycleTypes* zaimplementowana na podstawie twierdzenia 1.

W ostatnim kroku łączymy wcześniej obliczone wartości, używając funkcji *getGameResult* (wiersz 8). Wyniki obliczone w tablicy *result* stanowią końcowe odpowiedzi dla wszystkich wierzchołków początkowych. Warto zaznaczyć, że wartość $result[v]$ jest obliczana podczas analizy cyklu $belong[v]$ (z wyjątkiem wyniku dla korzenia całego kaktusa, który musimy obliczyć osobno). Następnie wywołujemy pro-



Rys. 4: Cykl C widać na prawo od wierzchołka v_1. Wszystkie cykle wewnętrzne C niewyrastające z v_1 zostaną podstawione pod zmienną C' w wierszu 9. Kaktus po lewej wyrastający w stronę korzenia v_0 został narysowany ciągłą linią.

cedurę *analyze* rekurencyjnie dla wszystkich wewnętrznych cykli. Zauważmy, że typ nadkaktusa v jest równy $cycleType[v]$.

Kilku słów komentarza wymaga jeszcze funkcja *getCycleTypes*. Bezpośrednia implementacja twierdzenia 1 może działać w czasie $O(d^2)$, gdzie d równa się długości cyklu, która może być tego samego rzędu co n. Aby tego uniknąć, należy obliczyć odległość od najbliższego wygrywającego pęku w lewo oraz w prawo przy pomocy programowania dynamicznego. Przypadkiem szczególnym, na który trzeba uważać, jest cykl z tylko jednym pękiem wygrywającym, zakorzenionym w v. Patrząc z perspektywy v, na cyklu nie ma pęków wygrywających, zaś dla pozostałych wierzchołków taki kaktus będzie typu **00** lub **11**.

Ostatnia uwaga: w przedstawionych pseudokodach beztrosko przekazywaliśmy tablice jako parametry funkcji, jednak należy zadbać o to, by przekazywanymi obiektami były jedynie indeksy lub wskaźniki. W przeciwnym razie kopiowanie obiektów może prowadzić do złożoności obliczeniowej $\Omega(n^2)$. Podobnie może się skończyć przekazywanie do funkcji *getGameResult* tablicy cykli zamiast operowania na samych konfiguracjach. Poradzenie sobie z tą ostatnią pułapką pozwala na osiągnięcie złożoności liniowej i taki algorytm można znaleźć w pliku `hyd.cpp`.

Karol Pokorski
Treść zadania, Opracowanie

Karol Pokorski, Piotr Smulewicz
Program

Dostępna pamięć: 256 MB. *OI, etap I, 19.10–16.11.2015*

Korale

Bajtyna ma n korali ponumerowanych liczbami od 1 do n. Korale są parami różne. Pewne z nich są bardziej wartościowe od innych – dla każdego z korali znana jest jego wartość w bajtalarach.

Bajtyna chciałaby stworzyć naszyjnik z niektórych ze swoich korali. Jest wiele sposobów utworzenia takiego naszyjnika. Powiemy, że dwa sposoby są różne, jeśli zbiory korali użytych do ich konstrukcji są różne. Aby nieco ułatwić sobie wybór, Bajtyna postanowiła uporządkować wszystkie sposoby utworzenia naszyjnika.

Najważniejszym kryterium jest suma wartości korali w naszyjniku. Im większa suma, tym sposób powinien być późniejszy w uporządkowaniu. Jeśli zaś mamy dwa różne sposoby utworzenia naszyjnika, które mają równą sumę wartości, to porównujemy je według porządku leksykograficznego posortowanych rosnąco list numerów użytych korali[1]*.*

Dla przykładu rozważmy sytuację, w której są cztery korale warte kolejno (zgodnie z numeracją) 3, 7, 4 i 3 bajtalary. Z takich korali naszyjnik można utworzyć na 16 sposobów. Poniżej znajduje się uporządkowanie tych sposobów zgodnie z pomysłem Bajtyny.

Numer sposobu	*Wartości wybranych korali*	*Suma wartości wybranych korali*	*Numery wybranych korali*
1	*brak*	*0*	*brak*
2	*3*	*3*	*1*
3	*3*	*3*	*4*
4	*4*	*4*	*3*
5	*3 3*	*6*	*1 4*
6	*3 4*	*7*	*1 3*
7	*7*	*7*	*2*
8	*4 3*	*7*	*3 4*
9	*3 7*	*10*	*1 2*
10	*3 4 3*	*10*	*1 3 4*
11	*7 3*	*10*	*2 4*
12	*7 4*	*11*	*2 3*
13	*3 7 3*	*13*	*1 2 4*
14	*3 7 4*	*14*	*1 2 3*
15	*7 4 3*	*14*	*2 3 4*
16	*3 7 4 3*	*17*	*1 2 3 4*

Bajtyna chciałaby stworzyć naszyjnik, który ma k-ty numer w uporządkowaniu. Pomóż jej!

[1] Ciąg numerów korali $i_1, \ldots, i_p$ jest mniejszy leksykograficznie od ciągu numerów korali $j_1, \ldots, j_q$, jeśli albo pierwszy ciąg jest początkowym fragmentem drugiego (czyli $p < q$, $i_1 = j_1, \ldots, i_p = j_p$), albo na pierwszej pozycji, na której ciągi te różnią się, pierwszy ciąg ma mniejszy element niż drugi (czyli istnieje takie $u \in \{1, \ldots, \min(p, q)\}$, że $i_1 = j_1, \ldots, i_{u-1} = j_{u-1}$ oraz $i_u < j_u$).

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie dodatnie liczby całkowite n i k oddzielone pojedynczym odstępem, określające liczbę korali oraz żądany numer sposobu utworzenia naszyjnika według uporządkowania opisanego powyżej. W drugim wierszu wejścia znajduje się ciąg n dodatnich liczb całkowitych $a_1, a_2, \ldots, a_n$ pooddzielanych pojedynczymi odstępami – wartości kolejnych korali.

Możesz założyć, że Bajtyna nie pomyliła się i rzeczywiście istnieje co najmniej k różnych sposobów utworzenia jej naszyjnika.

Wyjście

W pierwszym wierszu standardowego wyjścia należy wypisać jedną liczbę całkowitą, oznaczającą sumę wartości korali w znalezionym rozwiązaniu. W drugim wierszu wyjścia należy wypisać ciąg numerów korali użytych w naszyjniku w kolejności rosnącej, rozdzielając liczby pojedynczymi odstępami.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`4 10`	`10`
`3 7 4 3`	`1 3 4`

Testy „ocen":

`1ocen`: *$n = 10$, wszystkie korale mają wartość 1,*

`2ocen`: *$n = 9$, wartości korali są kolejnymi potęgami dwójki,*

`3ocen`: *$n = 11$, jest jeden koral wart 1 bajtalar oraz 10 korali wartych 10^9 bajtalarów, zaś poprawne rozwiązanie używa wszystkich jedenastu korali.*

`4ocen`: *$n = 1\,000\,000$, $k = 10$, wartości kolejnych korali są kolejnymi liczbami od 1 do $1\,000\,000$.*

Ocenianie

Zestaw testów dzieli się na podane poniżej podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby grup testów.

We wszystkich podzadaniach zachodzą warunki $n, k \leqslant 1\,000\,000$ oraz $a_i \leqslant 10^9$.

Jeśli odpowiedź dla danego testu nie jest prawidłowa, jednak pierwszy wiersz wyjścia (suma wartości korali w znalezionym rozwiązaniu) jest prawidłowy, wówczas przyznaje się połowę liczby punktów za dany test (oczywiście odpowiednio przeskalowaną w przypadku przekroczenia przez program połowy limitu czasowego). Dzieje się tak, nawet jeśli drugi wiersz wyjścia jest nieprawidłowy, nie został wypisany lub gdy wypisano więcej niż dwa wiersze wyjścia.

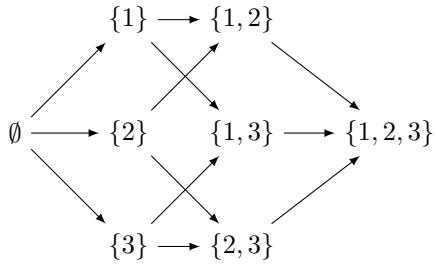
Podzadanie	Dodatkowe warunki	Liczba punktów
1	$n \leqslant 20$, $k \leqslant 500\,000$	*8*
2	$n \leqslant 60$, $k \leqslant 50\,000$	*12*
3	$n \leqslant 3\,000$, $n \cdot k \leqslant 10^6$, $a_i \leqslant 100$	*14*
4	$n \cdot k \leqslant 10^6$	*16*
5	$n \cdot k \leqslant 10^7$	*20*
6	*brak*	*30*

Rozwiązanie

Graf naszyjników

Liczba naszyjników, które można utworzyć z podanych na wejściu korali, wynosi aż 2^n, jednak zadanie dotyczy wypisania k-tego w kolejności naszyjnika. Z uwagi na stosunkowo niewielki limit na wartość liczby k (do miliona) rozsądnym podejściem wydaje się rozpatrywanie naszyjników po kolei.

Rozważmy skierowany graf, w którym wierzchołki reprezentują różne naszyjniki. Każda krawędź prowadzi do wierzchołka reprezentującego naszyjnik zawierający jeden dodatkowy koral względem naszyjnika reprezentowanego przez początek krawędzi.



Rys. 1: Graf naszyjników dla $n = 3$.

Krawędziom można przypisać wagi równe różnicy wartości naszyjników na końcach krawędzi (czyli wartości dodanego korala). Zadanie wówczas sprowadza się do znalezienia k-tego najbliższego wierzchołka od źródła (przy odpowiednim rozpatrywaniu remisów), przy czym źródłem jest wierzchołek reprezentujący pusty naszyjnik.

Algorytm Dijkstry

W tym miejscu wypada przypomnieć algorytm Dijkstry, który służy do znajdowania najkrótszych ścieżek z ustalonego źródła s do wszystkich pozostałych wierzchołków grafu $G = (V, E)$, przy założeniu nieujemnych wag krawędzi:

```
1: procedure Dijkstra(s, (V, E))
2: begin
3:     foreach v ∈ V do d[v] := ∞;
```

```
4:    d[s] := 0;
5:    C := ∅;
6:    for i := 1 to |V| do begin
7:      u := argmin{d[x] : x ∈ V \ C};
8:      foreach uv ∈ E do
9:        if d[v] > d[u] + c(u, v) then
10:         d[v] := d[u] + c(u, v);
11:     C := C ∪ {u};
12:   end
13: end
```

W każdym obiegu głównej pętli jako u przyjmuje się wierzchołek o najmniejszym oszacowaniu odległości ze źródła (najmniejszej wartości $d[u]$), który nie został jeszcze przetworzony (wierzchołki przetworzone dodawane są do zbioru C). Przetwarzanie wierzchołka sprowadza się do przeanalizowania możliwości poprawy oszacowania odległości ze źródła dla wszystkich końców krawędzi z niego wychodzących.

Poprawność algorytmu wynika z faktu, że w każdym obiegu głównej pętli oszacowanie odległości ze źródła s do przetwarzanego wierzchołka u jest w istocie już poprawnie obliczoną odległością z s do u. Dowód tego faktu można przeprowadzić indukcyjnie; patrz np. książka [6]. Dobrym ćwiczeniem na sprawdzenie zrozumienia tego dowodu jest zastanowienie się, w których miejscach dowodu korzysta się z faktu, że wagi krawędzi są nieujemne.

W standardowych implementacjach algorytmu Dijkstry zbiór $V \setminus C$ przechowywany jest na kolejce priorytetowej, implementowanej na przykład za pomocą kopca binarnego lub, w języku C++, struktury `std::priority_queue` czy struktury `std::set`. Przechowywanie zbioru w kolejce priorytetowej pozwala znajdować wierzchołek u o najmniejszym oszacowaniu odległości w czasie $O(\log |V|)$. Należy jeszcze pamiętać, że po zmianie oszacowania odległości dowolnego wierzchołka uporządkowanie zbioru nieprzetworzonych wierzchołków może się zmienić. Implementacje kolejek priorytetowych z użyciem kopca binarnego lub struktury `std::set` potrafią sobie z tym radzić z użyciem $O(\log |V|)$ porównań i zmian w pamięci, co powoduje, że algorytm ma złożoność czasową $O((|V| + |E|) \log |V|)$. Istnieją implementacje kolejek priorytetowych, które przy zmniejszeniu oszacowania odległości wierzchołka potrafią uporządkować zbiór w zamortyzowanym czasie $O(1)$[1], co poprawia złożoność algorytmu do $O(|V| \log |V| + |E|)$. Są one jednak dość skomplikowane, a w praktycznym zastosowaniu nie można zaobserwować ich asymptotycznej przewagi.

W naszym zadaniu należałoby odrobinę zmodyfikować algorytm Dijkstry, aby poprawnie rozpatrywał remisy, czyli sytuacje, w których oszacowanie odległości dla dwóch lub więcej różnych wierzchołków jest takie samo. Oczywiście należy po prostu porównać leksykograficznie naszyjniki reprezentowane przez te wierzchołki zgodnie z treścią zadania. Problemem jest jednak rozmiar grafu. Wierzchołki reprezentują bowiem naszyjniki, a tych jest (jak już wcześniej zauważyliśmy) 2^n. Grafu naszyjników nie trzeba jednak tworzyć w pamięci, a na kolejce priorytetowej wystarczy pamiętać wierzchołki, dla których została już znaleziona (niekoniecznie najkrótsza) ścieżka ze źródła, którym jest wierzchołek reprezentujący pusty naszyjnik.

[1] Chodzi między innymi o kopiec Fibonacciego ([6]).

Zastosowanie najkrótszych ścieżek do rozwiązania zadania

Siła algorytmu Dijkstry w zastosowaniu do tego zadania tkwi w tym, że przetwarza on wierzchołki grafu (naszyjniki) w kolejności takiej, której chciała Bajtyna. Algorytm zatem można przerwać po przetworzeniu k wierzchołków, a naszyjnik reprezentowany przez ów k-ty wierzchołek zwrócić na wyjście.

Łącznie liczbę operacji wykonywanych na kolejce priorytetowej będzie można oszacować przez $O(nk \log(nk))$, ponieważ przetworzenie każdego naszyjnika wprowadzi co najwyżej n zmian do kolejki priorytetowej (zmianą jest tutaj dodanie nowego naszyjnika lub poprawienie oszacowania odległości). Takie rozwiązanie powinno zaliczyć pierwsze dwa podzadania, a przy efektywnej implementacji miało szansę zaliczyć także podzadania trzecie i czwarte, co prowadziło do wyniku rzędu 20–50% punktów.

Na koniec, warto jeszcze zastanowić się nad tym, ile czasu zajmuje porównanie naszyjników. Jeśli naszyjniki mają różną wartość, a implementacja ją przechowuje, wówczas porównanie może się odbyć w czasie stałym. Jeśli jednak naszyjniki mają równą wartość, wówczas należy je porównać leksykograficznie względem numerów korali, co kosztuje czas proporcjonalny do długości krótszego z naszyjników. Teoretycznie, w najgorszym przypadku porównywane naszyjniki mają $O(n)$ korali. W naszej sytuacji możemy jednak podać dużo lepsze oszacowanie. Zauważmy, że jeśli naszyjnik ma więcej niż $\lceil \log k \rceil$ korali, to jest więcej niż $2^{\lceil \log k \rceil} - 1 \geqslant k - 1$ podzbiorów jego korali, które na pewno będą miały mniejszą wartość. A zatem k najwcześniejszych w kolejności naszyjników ma $O(\log k)$ korali i tyle czasu w najgorszym przypadku zajmuje ich porównanie.

Redukcja liczby naszyjników na kolejce priorytetowej

Powyższe rozwiązanie ma istotny problem: dla każdego przetwarzanego naszyjnika liczba zmian w kolejce priorytetowej jest rzędu $O(n)$. Wynika to stąd, że do naszyjnika, który ma c korali, kolejny koral można dołożyć na $n - c$ sposobów. Intuicja podpowiada, że tak duża liczba krawędzi grafu nie jest konieczna. Do każdego wierzchołka reprezentującego naszyjnik c-koralowy prowadzi bowiem $c!$ ścieżek, bo dla każdej permutacji korali w naszyjniku, można je dokładać w tej właśnie kolejności. Jedna ścieżka byłaby wystarczająca.

Niech $I = (i_1, i_2, \ldots, i_n)$ oznacza listę numerów korali w kolejności od najmniejszych do największych wartości (remisy rozstrzygamy dowolnie). Zmieńmy teraz definicję krawędzi w grafie naszyjników. Z wierzchołka reprezentującego naszyjnik $B = \{i_{j_1}, i_{j_2}, \ldots, i_{j_c}\}$, przy czym $j_1 < j_2 < \ldots < j_c$, będą wychodzić dwie krawędzie (o ile $j_c \neq n$):

a) do naszyjnika, w którym koral i_{j_c} zastępujemy koralem i_{j_c+1},

b) do naszyjnika z dodanym koralem i_{j_c+1}.

Tym razem źródłem przeszukiwania będzie wierzchołek reprezentujący naszyjnik $\{i_1\}$ i będziemy poszukiwali $(k-1)$-szego najbliższego wierzchołka ze źródła. Zauważmy, że teraz do wierzchołka reprezentującego naszyjnik $\{i_{j_1}, i_{j_2}, \ldots, i_{j_c}\}$ prowadzi już tylko jedna ścieżka: taka, w której najpierw koral i_1 zamieniamy sukcesywnie w koral i_{j_1},

następnie dodajemy koral i_{j_1+1}, po czym sukcesywnie zamieniamy go w i_{j_2}, itd. Trzeba będzie tylko pamiętać, aby przypadek $k = 1$ obsłużyć jako przypadek szczególny.

W nowym grafie waga krawędzi ponownie będzie równa różnicy wartości naszyjników reprezentowanych przez jej końce. Tym razem nie będzie to zawsze wartość dodanego korala, gdyż niektóre krawędzie wymieniają jeden koral na inny. Nadal jednak wszystkie krawędzie mają nieujemne wagi, bo ewentualna wymiana korala zastępuje go koralem o co najmniej takiej samej wartości.

W nowym grafie również można zastosować algorytm Dijkstry. Tym razem na kolejce priorytetowej znajdzie się tylko $O(k)$ naszyjników, gdyż przetworzenie każdego naszyjnika może dodać do kolejki co najwyżej dwa nowe. Na początku potrzebne jest posortowanie ciągu n korali według wartości. Następnie wykonujemy $O(k)$ operacji na kolejce priorytetowej, z których każda wymaga wykonania $O(\log k)$ porównań naszyjników. Jedno porównanie naszyjników kosztuje $O(\log k)$ operacji, zakładając, że korale w każdym naszyjniku pamiętamy zarówno w porządku niemalejących wartości (co służy do generowania krawędzi wychodzących), jak i w porządku numerów (co jest potrzebne właśnie do wykonywania porównań). Dodatkowo, wygenerowanie krawędzi wychodzących z danego wierzchołka zajmuje czas $O(\log k)$. Ostatecznie otrzymamy rozwiązanie działające w czasie $O(n \log n + k \log^2 k)$. Tego typu rozwiązanie otrzymywało 80–100% punktów w zależności od jakości implementacji. Przykładowy kod znajduje się w pliku `kors5.cpp`.

Szybsze generowanie naszyjników

W rozwiązaniu wzorcowym najpierw wyznaczymy listę L, w której dla każdego z k pierwszych naszyjników zapamiętamy jego wartość (v) oraz najmniejszy numer korala (y). Pokażemy dalej, że taka lista pozwoli nam odtworzyć numery wszystkich korali k-tego z kolei naszyjnika. Zysk, jaki z tego osiągniemy, będzie taki, że podczas generowania listy L nie będziemy musieli pamiętać wszystkich korali w poszczególnych naszyjnikach, dzięki czemu czas porównywania naszyjników zmniejszy się do stałego.

Przypomnijmy, że do generowania krawędzi w zmodyfikowanym grafie naszyjników wystarczy pamiętać dla danego naszyjnika numer korala o największej wadze, czyli numer najpóźniejszego korala w kolejności listy I. Oznaczmy numer tego korala w danym naszyjniku jako x. Dodatkowo, wymiana tego korala na inny (krawędź typu a)) może czasami spowodować zmianę numeru korala y. Aby sobie z tym poradzić, będziemy także pamiętać najmniejszy numer korala w naszyjniku z pominięciem tego o największej wadze (z). (Jeśli koral z nie istnieje, przyjmiemy, że $z = \infty$). Wreszcie poza numerem korala x zapamiętamy także jego pozycję p na liście I (tj. $i_p = x$). Ostatecznie dla każdego naszyjnika będziemy pamiętali tylko pięć liczb: v, x, y, z i p.

Upewnijmy się teraz, że przy przejściu krawędzią w grafie naszyjników będziemy w stanie zaktualizować przechowywane informacje. Przechodząc krawędzią typu a), koral x zastępujemy koralem $x' = i_{p+1}$ (i mamy $p' = p + 1$), co może wpłynąć na zmianę y: $y' = \min(z, x')$; natomiast z nie zmienia się ($z' = z$). Natomiast wskutek przejścia krawędzią typu b) po prostu dochodzi nam nowy koral $x' = i_{p+1}$ (i znów mamy $p' = p + 1$); to oznacza, że $z' = y$, natomiast $y' = \min(y, x')$. W obydwu przypadkach łatwo aktualizujemy wartość naszyjnika.

W kolejce priorytetowej naszyjniki porównujemy tylko według wartości v oraz, w drugiej kolejności, numeru korala y. Zauważmy, że to *nie wystarcza* do porównania naszyjników według kryterium Bajtyny, jednak pozwala w poprawny sposób wygenerować wszystkie elementy listy L jako $k-1$ pierwszych wierzchołków przetworzonych w algorytmie Dijkstry (na początek listy dodajemy pusty naszyjnik).

Przyjrzyjmy się teraz, jak za pomocą listy L możemy odzyskać wynik. Niech v_i oraz y_i oznaczają wartość i najmniejszy numer korala w i-tym naszyjniku z listy. Wiemy wówczas, że pierwszy koral k-tego naszyjnika to v_k, i możemy go wypisać. Niech m oznacza liczbę elementów listy L o wartości v_k i koralu y_k. Wówczas kolejne korale k-tego naszyjnika będą takie same jak korale m-tego naszyjnika na liście spośród tych o wartości $w = v_k - a_{y_k}$ i najmniejszym numerze korala większym niż v_k. Naszyjnik ten możemy znaleźć, cofając się na liście L do pierwszego naszyjnika o wartości w i najmniejszym koralu większym niż v_k, a następnie idąc o $m-1$ naszyjników do przodu; niech i oznacza numer tak określonego naszyjnika. Wówczas numer korala y_i tego naszyjnika to zarazem drugi w kolejności numer korala k-tego naszyjnika. Rozumowanie to powtarzamy, począwszy od naszyjnika numer i, aż do znalezienia wszystkich numerów korali k-tego naszyjnika. Całe odzyskiwanie działa w czasie $O(k)$.

Przykład 1. Oto jak wygląda lista L dla przykładu w treści zadania ($k = 10$):

i	1	2	**3**	4	5	6	7	**8**	9	**10**
v_i	0	3	**3**	4	6	7	7	**7**	10	**10**
y_i	-	1	**4**	3	1	1	2	**3**	1	**1**

Przypomnijmy, że $a_1 = 3$, $a_2 = 7$, $a_3 = 4$ i $a_4 = 3$.

Pierwszy koral szukanego, dziesiątego w kolejności naszyjnika ma numer $y_{10} = 1$. Dalej, są $m = 2$ naszyjniki na liście o wartości $v_i = v_{10} = 10$ i koralu $y_i = y_{10} = 1$. Stąd kolejny koral dziesiątego naszyjnika jest taki sam jak pierwszy koral drugiego spośród naszyjników, które mają wartość $v_i = v_{10} - a_{y_{10}} = 7$ i najmniejszy numer korala $y_i > y_{10} = 1$. Szukany koral to zatem $y_8 = 3$ z naszyjnika numer 8. Kontynuując ten proces, stwierdzamy, że jest tylko $m = 1$ naszyjnik o wartości $v_i = v_8 = 7$ i koralu $y_i = y_8 = 3$. Tak więc kolejny koral dziesiątego naszyjnika jest taki sam jak pierwszy koral jedynego naszyjnika, który ma wartość $v_i = v_8 - a_{y_8} = 3$ i najmniejszy numer korala $y_i > y_8 = 3$. Jest to zatem koral numer $y_3 = 4$ z naszyjnika numer 3.

Złożoność rozwiązania zmniejsza się do $O(n \log n + k \log k)$, gdyż każde porównanie realizowane jest w czasie stałym, a liczba operacji na kolejce priorytetowej nie zmieniła się. Implementacja znajduje się w pliku `kor.cpp` i otrzymuje oczywiście maksymalną punktację. W praktyce jednak rozwiązanie to nie jest o wiele szybsze od poprzedniego, gdyż stały czynnik ukryty w notacji asymptotycznej jest dość duży – dla danych z zadania niewiele mniejszy od czynnika logarytmicznego w poprzednim rozwiązaniu.

Rozwiązanie alternatywne

Na zadanie można spojrzeć inaczej, bez rozważania grafu naszyjników. Zamiast tego, można próbować obliczać listy k pierwszych naszyjników w kolejności wyznaczonej

przez Bajtynę zbudowanych z wykorzystaniem wyłącznie korali o numerach $i, \ldots, n$. Listy te wyznaczymy kolejno dla $i = n + 1, \ldots, 1$.

Na początku, dla pustego ciągu dostępnych korali jest tylko jeden możliwy do uzyskania naszyjnik – pusty, o wartości równej 0.

Niech $l_1, l_2, \ldots, l_{k'}$ $(k' \leqslant k)$ oznaczają kolejne naszyjniki zgodnie z uporządkowaniem Bajtyny utworzone dla korali $a_{i+1}, a_{i+2}, \ldots, a_n$. Dodamy teraz do rozważań koral o numerze i. W tym celu utworzymy drugą listę naszyjników $m_1, m_2, \ldots, m_{k'}$, w której $m_i = l_i \cup \{i\}$ (do każdego naszyjnika dodaliśmy koral i).

Teraz wystarczy złączyć listy naszyjników l oraz m w jedną, podobnie jak scala się dwie posortowane listy w jedną w algorytmie sortowania przez scalanie. Jeśli długość listy wynikowej przekroczyła k, to można bezpiecznie odrzucić jej koniec po k-tym naszyjniku, bo żaden z usuwanych naszyjników nie będzie k-ty w kolejności (skoro jest co najmniej k wcześniejszych od każdego z nich). Rozpatrywanie sufiksów ciągu korali (zamiast np. prefiksów) pomaga w utrzymywaniu poprawnej kolejności leksykograficznej ciągu korali w przypadku rozstrzygania remisów – zawsze bowiem pierwszeństwo w przypadku równej wartości naszyjników w scalanych listach będzie miał naszyjnik z listy m (z dodanym koralem i) nad naszyjnikiem z listy l (z najwcześniejszym koralem o numerze co najmniej $i + 1$).

Rozwiązanie to można zaimplementować, aby działało w czasie $O(nk)$. Powinno zaliczać wszystkie podzadania poza ostatnim i uzyskiwać około 70% punktów. Implementacja znajduje się w pliku `kors3.cpp`.

Inne rozwiązania

Rozwiązanie wykładnicze

Narzucającym się rozwiązaniem jest wygenerowanie wszystkich 2^n naszyjników. Wówczas można je posortować lub skorzystać z algorytmu selekcji[2], co prowadzi do algorytmu o złożoności $\Omega(2^n)$, który mógł zaliczyć jedynie pierwsze podzadanie.

Implementacja znajduje się w pliku `kors1.cpp`.

Rozwiązanie oparte o wydawanie reszty

Na zadanie można też spojrzeć nieco inaczej, jako na problem wydawania reszty, gdzie nominałami są wartości korali. Z użyciem algorytmu opartego o programowanie dynamiczne możliwe jest wyznaczenie dla każdej kwoty (wartości naszyjnika) liczby sposobów jej wydania (liczby różnych naszyjników o tej wartości).

Niestety, odzyskanie k-tego naszyjnika tą metodą nie jest możliwe; możliwe jest jedynie odzyskanie jego wartości.

Rozwiązanie to pozwalało uzyskać połowę punktów za trzecie podzadanie. Jest ono zaimplementowane w pliku `korb2.cpp`.

[2] Algorytm magicznych piątek lub algorytm Hoare'a [6]

Jacek Tomasiewicz Treść zadania | **Krzysztof Kiljan** Opracowanie | **Kamil Dębowski** Program

Dostępna pamięć: 256 MB. *OI, etap I, 19.10–16.11.2015*

Nadajniki

Bajtazar został nowym dyrektorem zabytkowej kopalni soli pod Bajtowem. Aby zwiększyć popularność tego obiektu wśród turystów, postanowił zainstalować w korytarzach kopalni bezprzewodowy Internet.

Kopalnia składa się z n komór połączonych $n - 1$ korytarzami. Z każdej komory można przejść do każdej innej, używając korytarzy. Bajtazar postanowił rozmieścić w komorach nadajniki wi-fi tak, by Internet był dostępny w każdym z korytarzy kopalni. Aby można było korzystać z Internetu w korytarzu łączącym komory a i b, musi być spełniony co najmniej jeden z poniższych warunków:

- *w komorze a lub w komorze b znajduje się nadajnik, lub*
- *w zbiorze komór, do których można dojść z komory a lub komory b, używając co najwyżej jednego korytarza, znajdują się co najmniej dwa nadajniki.*

Bajtazar zastanawia się teraz, jaka jest minimalna liczba nadajników wi-fi, które musi rozmieścić, aby można było korzystać z Internetu w każdym korytarzu. W każdej komorze można umieścić dowolną liczbę nadajników.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dodatnią liczbę całkowitą n oznaczającą liczbę komór w kopalni. Komory numerujemy liczbami od 1 do n.

Kolejne $n - 1$ wierszy opisuje korytarze w kopalni. Każdy z nich zawiera dwie liczby całkowite a i b ($1 \leqslant a, b \leqslant n$, $a \neq b$) oddzielone pojedynczym odstępem, oznaczające, że komory o numerach a i b są połączone korytarzem.

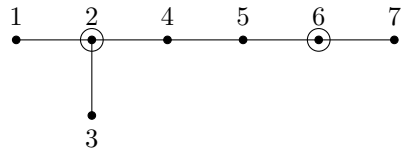
Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, oznaczającą minimalną liczbę nadajników, które musi rozmieścić Bajtazar.

Przykład

Dla danych wejściowych:

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

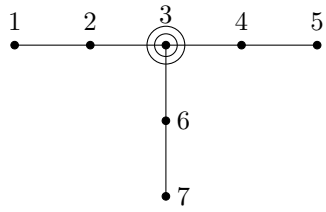


poprawnym wynikiem jest:

```
2
```

i dla danych wejściowych:

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



poprawnym wynikiem jest:

```
2
```

Wyjaśnienie do przykładów: *W pierwszym przykładzie wystarczy umieścić nadajniki w komorach o numerach 2 i 6, natomiast w drugim przykładzie wystarczy umieścić dwa nadajniki w komorze numer 3.*

Testy „ocen":

`1ocen`: $n = 16$. *Komora* i *jest połączona z komorą* $\lfloor i/2 \rfloor$ *dla* $2 \leqslant i \leqslant n$.

`2ocen`: $n = 303$. *Komora 2 jest połączona z komorami 1 oraz 3. Każda z komór 1, 2, 3 jest dodatkowo połączona z setką komór. Optymalnym rozwiązaniem jest umieszczenie dwóch nadajników w komorze 2.*

`3ocen`: $n = 200\,000$. *Komory* i *oraz* $i + 1$ *są połączone korytarzem dla* $1 \leqslant i \leqslant n - 1$.

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n \leqslant 10$	*15*
2	$n \leqslant 500$	*20*
3	$n \leqslant 200\,000$, *do każdej komory prowadzą co najwyżej trzy korytarze*	*25*
4	$n \leqslant 200\,000$	*40*

Rozwiązanie

Wprowadzenie

W zadaniu mamy dany spójny graf nieskierowany o n wierzchołkach oraz $n-1$ krawędziach. Powszechnie przyjętą nazwą dla takich grafów jest *drzewo*. Drzewa charakteryzują się brakiem jakichkolwiek cykli oraz tym, że dla każdej pary wierzchołków istnieje dokładnie jedna ścieżka je łącząca.

Na potrzeby tego zadania, jako *zlewisko* krawędzi $e = (u, v)$ zdefiniujemy zbiór wszystkich wierzchołków sąsiadujących z u lub z v, co matematycznie można zapisać na przykład w następujący sposób: $Z(u, v) = \{w : (u, w) \in E \vee (v, w) \in E\}$, gdzie E to zbiór wszystkich krawędzi naszego drzewa.

W każdym z wierzchołków wstawiamy pewną nieujemną liczbę nadajników. Krawędź $e = (u, v)$ nazwiemy *spełnioną*, jeżeli zachodzi przynajmniej jeden z poniższych warunków:

- W wierzchołku u lub w wierzchołku v znajduje się co najmniej jeden nadajnik.
- W zlewisku krawędzi e znajdują się co najmniej dwa nadajniki.

W pierwszym przypadku mówimy, że krawędź jest spełniona *bezpośrednio*, a w drugim, że jest spełniona *pośrednio*.

Naszym zadaniem jest wyznaczenie najmniejszej liczby nadajników, których wstawienie gwarantuje, że wszystkie krawędzie w drzewie będą spełnione.

Rozwiązanie wykładnicze

Rozwiązywanie zadania zacznijmy od prostego spostrzeżenia.

Obserwacja 1. W żadnym wierzchołku nie warto umieścić więcej niż 2 nadajników.

Obserwacja ma oczywiste uzasadnienie – z warunków zadania wynika, że postawienie więcej niż dwóch nadajników nie może w żaden sposób wpłynąć na spełnialność którejkolwiek krawędzi. Pomysł ten pozwala nam na stworzenie pierwszego wolnego rozwiązania. W każdym z wierzchołków mamy trzy możliwości – możemy ustawić 0 nadajników, 1 albo 2.

Jako że mamy n wierzchołków, wszystkich możliwych rozstawień nadajników w wierzchołkach jest 3^n. Jeśli dane rozstawienie jest poprawne, to sprawdzamy, czy wymaga ono mniejszej liczby nadajników niż najlepsze znalezione przez nas dotychczas. Jeśli tak, to aktualizujemy potrzebną liczbę nadajników. Najprościej zaimplementować takie rozwiązanie jako funkcję rekurencyjną, którą przedstawiamy poniżej.

Przyjmujemy, że mamy napisane funkcje *calcCnt*() oraz *check*(). Pierwsza z nich ma za zadanie wyznaczyć liczbę nadajników używanych przez aktualne rozstawienie. Można ją napisać w złożoności $O(n)$, po prostu obliczając sumę wartości w tablicy *cnt*[] przechowującej liczby nadajników umieszczonych w poszczególnych wierzchołkach. Druga z funkcji sprawdza, czy dane rozstawienie nadajników powoduje spełnienie wszystkich krawędzi. Można to zrealizować, dla każdego wierzchołka zliczając

nadajniki umieszczone w jego sąsiadach (oznaczmy taką wartość przez $cntInNeigh[v]$), a potem dla każdej krawędzi (u, v) sprawdzając, czy $cnt[u] + cnt[v] \geqslant 1$ lub $cntInNeigh[u] + cntInNeigh[v] \geqslant 2$. Przy takim podejściu każde sprawdzenie zajmuje czas $O(n)$, tak więc otrzymujemy łączną złożoność czasową $O(3^n \cdot n)$.

```
 1: bestAns := ∞;
 2: function genRec(v)
 3: begin
 4:   for i := 0 to 2 do begin
 5:     cnt[v] := i;
 6:     if v = n then begin
 7:       if check() then
 8:         bestAns := min(bestAns, calcCnt());
 9:     end else genRec(v + 1);
10:   end
11: end
```

Funkcję wywołujemy jako $genRec(1)$. Rozwiązania podobne do powyższego można znaleźć w plikach `nads1.cpp` oraz `nads2.cpp`. Na zawodach poprawna implementacja takiego rozwiązania pozwalała na zdobycie między 10 a 15 punktów.

Rozwiązanie wzorcowe

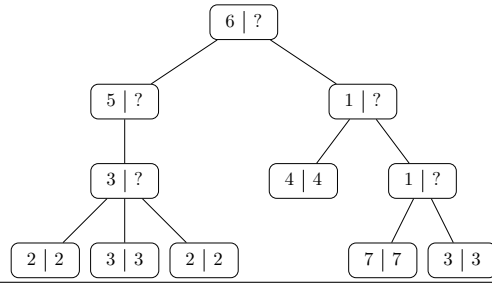
Nie jest trudno zauważyć, że przy ograniczeniach, jakie mamy dane w zadaniu ($n \leqslant 200\,000$), rozwiązania wykładnicze nie mają prawa skończyć się w żadnym sensownym czasie. Musimy więc poszukać czegoś o wiele szybszego.

Programowanie dynamiczne na drzewach

Na początku postaramy się przybliżyć, na czym polega technika programowania dynamicznego na drzewach. Jeśli Czytelnik jest z nią obeznany, może przejść do kolejnego podrozdziału, w którym będziemy próbować ją zastosować do naszego zadania. W ogólności programowanie dynamiczne polega na obliczaniu jakichś wartości dla większych egzemplarzy problemu na podstawie wyników dla mniejszych egzemplarzy. Zaczyna się od bardzo prostych przypadków, a następnie pokazuje się, jak wyniki dla mniejszych egzemplarzy złożyć w wyniki dla bardziej skomplikowanych egzemplarzy. Przykładami podstawowych problemów rozwiązywanych za pomocą programowania dynamicznego jest najdłuższy wspólny podciąg oraz (dyskretny) problem plecakowy.

Technikę tę da się również zaadaptować do wielu problemów, w których mamy do czynienia z drzewami. Wystarczy, że ukorzenimy drzewo w jednym z wierzchołków i będziemy je przechodzić od liści do korzenia. Wartość dla poddrzewa ukorzenionego w ojcu będziemy wyznaczać na podstawie wartości dla poddrzew jego dzieci. Takie podejście nazywane jest programowaniem dynamicznym *z dołu do góry*.

Dla lepszego zrozumienia, spróbujemy zobrazować to na przykładzie następującego prostego zadania: Mamy dane drzewo ukorzenione w wierzchołku numer 1. Każdy wierzchołek zawiera pewną liczbę monet v. Chcemy być w stanie w czasie $O(1)$ odpowiadać na zapytania postaci „Ile jest monet w danym poddrzewie?”.

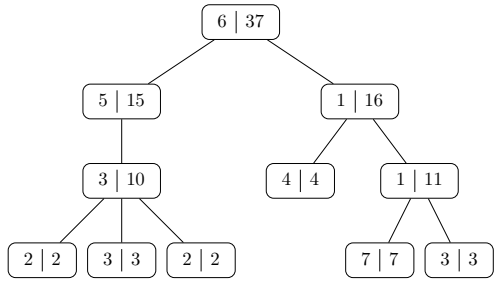


Rys. 1: Lewa wartość oznacza liczbę monet w wierzchołku, prawa – w całym poddrzewie.

W tym przypadku wnioski są oczywiste – dla każdego wierzchołka v szukana wartość jest równa jego liczbie monet $coins[v]$ powiększonej o liczbę monet w poddrzewach jego dzieci. Takie rozwiązanie możemy zrealizować za pomocą poniższego pseudokodu:

```
1:  function dfs(v)
2:  begin
3:    vis[v] := true;
4:    dp[v] := coins[v];
5:    foreach (v, child) ∈ E do
6:      if not vis[child] then begin
7:        dfs(child);
8:        merge(v, child);
9:      end
10: end
11: function merge(parent, child)
12: begin
13:   dp[parent] += dp[child];
14: end
```

Procedurę wywołujemy jako $dfs(1)$. Wartości wyznaczone za pomocą powyższego algorytmu dla drzewa z rysunku 1 można obejrzeć na rysunku 2.



Rys. 2: Drzewo z wyznaczonymi wartościami.

Stanem w programowaniu dynamicznym nazywamy egzemplarz podproblemu, dla którego obliczamy wartość. W powyższym przykładzie stan odpowiada poddrzewu oryginalnego drzewa, identyfikowanemu poprzez wierzchołek będący korzeniem poddrzewa.

Warto wspomnieć, że w programowaniu dynamicznym można wyróżnić dwa podejścia, tzw. *w tył* oraz *w przód*.

W podejściu *w tył* (częściej spotykanym) ustalamy pewien stan, którego wartość chcemy obliczyć, i wyznaczamy ją na podstawie już znanych wartości wyliczonych dla jego poprzedników, czyli stanów, od wartości których zależy jego wartość.

W podejściu *w przód* ustalamy pewien stan, którego wartość już znamy, i przeglądamy wszystkie stany, których wartości jeszcze nie znamy, a na które wpływ ma wartość owego ustalonego stanu (tzn. następników stanu).

Podejście *w tył* możemy zastosować, gdy dla stanu potrafimy wyznaczyć jego poprzedników, a podejście *w przód*, gdy dla stanu potrafimy wyznaczyć jego następników. W zależności od problemu, być może możemy zastosować oba podejścia lub tylko jedno z nich. Przykładowo, w przypadku naszego prostego zadania możliwe są oba podejścia. W powyższym pseudokodzie zaimplementowaliśmy podejście *w tył*, co wymaga założenia, że możemy efektywnie wyznaczyć wszystkie dzieci wierzchołka – a zatem należy je utrzymywać np. na liście sąsiedztwa.

Aby przećwiczyć technikę programowania dynamicznego na drzewie, można zmierzyć się na przykład z zadaniem *Farmercraft* z dnia próbnego finałów XXI Olimpiady Informatycznej [1].

Programowanie dynamiczne na drzewach w zadaniu *Nadajniki*

Do rozwiązania naszego zadania również można zastosować metodę programowania dynamicznego, jednak nie jest zupełnie oczywiste jak to zrobić. Dużym utrudnieniem może okazać się to, że nadajniki powodujące spełnienie krawędzi łączącej aktualnie przetwarzany wierzchołek z jego synem mogą znajdować się nie tylko w poddrzewie wierzchołka, ale również gdzieś wyżej. Łatwo jest też wpaść w pułapkę i wymyślić stany, których łączenie jest bardzo trudne lub wręcz niemożliwe do wykonania.

Całość zaczynamy oczywiście od ukorzenienia naszego drzewa w dowolnym wierzchołku (np. w tym o numerze 1). W rozwiązaniu wzorcowym dla każdego wierzchołka v będziemy wyliczać trójwymiarową tablicę o indeksach od 0 do 2. Dla ustalenia uwagi, będziemy ją oznaczać $t[inMe][inPar][later]$. Parametry używane w tej tablicy reprezentują odpowiednio:

- *inMe* – liczbę nadajników, które wstawiamy w wierzchołku v;
- *inPar* – liczbę nadajników, które są nam potrzebne w ojcu wierzchołka v ze względu na to, że któraś z krawędzi między wierzchołkiem v a jego synem jeszcze nie została spełniona;
- *later* – liczbę nadajników, których brakuje do pośredniego spełnienia krawędzi między wierzchołkiem v a jego ojcem, przy założeniu, że krawędź ta istnieje i nie jest spełniona bezpośrednio. Mogą być w tym celu wykorzystane nadajniki umieszczone w innych dzieciach ojca wierzchołka v lub w wierzchołku będącym dziadkiem wierzchołka v. Rozważana krawędź będzie spełniona bez względu na

parametr *later*, jeżeli w wierzchołku v lub w jego ojcu postawimy co najmniej jeden nadajnik.

Wartość $t[inMe][inPar][later]$ oznacza minimalną liczbę nadajników umieszczonych w poddrzewie wierzchołka v, które są zgodne z podanymi wartościami parametrów i spełniają wszystkie krawędzie w poddrzewach synów wierzchołka v.

Mając pewien korzeń v poddrzewa oraz wyliczone stany dla każdego z jego synów, będziemy chcieli wyliczyć jego tablicę w następujący sposób:

1. Składamy wyniki wszystkich synów v.
2. Wyliczamy tablicę dla v na podstawie scalonego wyniku dla synów, próbując umieszczenia w v każdej możliwej liczby nadajników.

Warto w tym momencie od razu zastanowić się, jakimi wartościami powinniśmy takie tablice zainicjować dla pojedynczych wierzchołków (co będzie wykorzystywane w liściach). Krawędzie od wierzchołka v do synów nie istnieją, zatem parametr *inPar* powinien być równy zeru. Jeżeli parametr *inMe* jest dodatni, to spełniamy także krawędź od wierzchołka v do jego ojca, więc w takim przypadku $later = 0$. Jeżeli jednak $inMe = 0$, to potrzebujemy dwóch nadajników, aby spełnić tę krawędź pośrednio, czyli $later = 2$. Jeżeli zatem $(inMe, inPar, later)$ przyjmuje którąś z wartości $(0, 0, 2)$, $(1, 0, 0)$, $(2, 0, 0)$, to wartość w tablicy programowania dynamicznego wynosi *inMe*. Wszystkie pozostałe stany reprezentują niedopuszczalny zbiór parametrów, zatem ustawiamy w nich bardzo dużą wartość, np. 10^9 (równoważną nieskończoności w tym zadaniu). Można dodatkowo poczynić obserwację, że jeżeli w dopuszczalnym rozstawieniu dla całego drzewa wszystkie nadajniki z konkretnego liścia przesuniemy do jego ojca, to także otrzymamy dopuszczalne rozstawienie o takim samym koszcie. Pozwala to założyć, że w liściach nie wstawiamy żadnych nadajników, co zmniejsza zbiór stanów, które musimy uwzględnić, do $(0, 0, 2)$.

Scalanie synów

Będziemy scalać dwie tablice – *t1* oraz *t2* – w jedną nową. Pierwsza z nich będzie odpowiadała wszystkim dotychczas scalonym synom, druga zaś temu, którego aktualnie będziemy dołączać. Parametry *inMe*, *inPar* oraz *later* będą otrzymywać przyrostki 1 i 2 w zależności od tablicy, z której pochodzą. Trzeba w tym momencie podkreślić, że wynikowa tablica nie jest tablicą odpowiadającą żadnemu konkretnemu poddrzewu, a zbiorowi poddrzew posiadających tego samego ojca (bez uwzględnienia samego ojca). Definicje parametrów *inMe*, *inPar* i *later* zmieniają się, ale w dość intuicyjny sposób:

- *inMe* – liczba nadajników, które wstawiliśmy w korzeniach tych poddrzew;
- *inPar* – liczba nadajników, które są nam potrzebne we wspólnym ojcu wszystkich uwzględnionych poddrzew, która zapewnia spełnienie wszystkich krawędzi pomiędzy korzeniami tych poddrzew a ich synami;
- *later* – liczba nadajników, których brakuje do spełnienia w sposób pośredni tych krawędzi między korzeniami poddrzew a ich wspólnych ojcem, które nie

są spełnione bezpośrednio. Mogą być one uzupełniane przez nadajniki z jeszcze nieuwzględnionych poddrzew, które mają tego samego ojca, lub z ojca wspólnego ojca.

Zastosujemy programowanie dynamiczne *w przód* (definicja znajduje się we wcześniejszej sekcji). Scalanie zrealizujemy, iterując po wszystkich możliwych wartościach parametrów tablic wejściowych i poprawiając, jeśli to możliwe, odpowiednie komórki tablicy wynikowej *res* (której wartości początkowo ustawiamy na nieskończoność).

Wyznaczanie parametrów *newInMe*, *newInPar* oraz *newLater* tablicy wynikowej przebiega następująco:

- *newInMe* jest równe $inMe1 + inMe2$.
- *newInPar* jest równe maksimum spośród *inPar1* oraz *inPar2*. Uzasadnienie jest jasne – oba warunki są spełnione wtedy i tylko wtedy, kiedy większy z nich jest spełniony.
- *newLater* równa się maksimum z wartości $later1 - inMe2$ i $later2 - inMe1$. Wartości te wynikają z tego, że nadajniki z korzenia sąsiedniego poddrzewa (brat aktualnego wierzchołka) mogą pomóc spełnić krawędź, której dotyczy *later*.

Jeśli wykonujemy jakieś operacje dodawania lub odejmowania na parametrach, to zawsze na koniec musimy je obciąć do przedziału $[0, 2]$ poprzez wzięcie minimów bądź maksimów z odpowiednich wartości. Zostało to zilustrowane w poniższym pseudokodzie.

```
1:  function merge(t1, t2)
2:  begin
3:    res := tablica wypełniona nieskończonościami;
4:    for inMe1 := 0 to 2 do
5:      for inPar1 := 0 to 2 do
6:        for later1 := 0 to 2 do
7:          for inMe2 := 0 to 2 do
8:            for inPar2 := 0 to 2 do
9:              for later2 := 0 to 2 do begin
10:               newInMe := min(2, inMe1 + inMe2);
11:               newInPar := max(inPar1, inPar2);
12:               newLater := max(0, max(later1 − inMe2, later2 − inMe1));
13:               res[newInMe][newInPar][newLater] :=
14:                 min(res[newInMe][newInPar][newLater],
15:                     t1[inMe1][inPar1][later1] + t2[inMe2][inPar2][later2]);
16:             end
17:   return res;
18: end
```

Obliczanie stanu ojca

Po scaleniu tablic t wszystkich poddrzew w jedną musimy ją jeszcze przełożyć na wynik dla samego korzenia v. Indeksy *newInMe*, *newInPar* oraz *newLater* będą określać wartości parametrów dla wierzchołka v. Będziemy poprawiać wartości w odpowiednich komórkach tablicy wynikowej *res*, iterując po wszystkich możliwych indeksach *inMe*, *inPar*, *later* zdefiniowanych jak w poprzedniej sekcji oraz po parametrze *newInMe*.

Zwróćmy uwagę na kilka faktów:

- Z definicji *inPar* wynika, że *newInMe* musi być co najmniej tak duże jak *inPar*.
- Jeśli $newInMe = 0$, to *newInPar* będzie równe staremu *later* – nadajniki, które wcześniej musieliśmy wstawić gdzieś wyżej, teraz musimy już wstawić w ojcu wierzchołka v.
- Jeśli $newInMe = 0$, to *newLater* będzie równe $2 - inMe$, czyli liczbie nadajników, których potrzebujemy do spełnienia krawędzi od wierzchołka v do jego ojca minus te, które otrzymaliśmy już z synów wierzchołka v.
- Jeśli *newInMe* jest większe od zera, to *newInPar* oraz *newLater* są równe zeru. Faktycznie, jeśli w wierzchołku v mamy jakiś nadajnik, to automatycznie wszystkie interesujące nas krawędzie stają się spełnione.

Po tych obserwacjach możemy już napisać pseudokod zmieniający tablicę sumy poddrzew na tablicę stanu przedstawiającego poddrzewo ukorzenione w wierzchołku v:

```
1:  function createParent(t)
2:  begin
3:    res := tablica wypełniona nieskończonościami;
4:    for newInMe := inPar to 2 do
5:      for inMe := 0 to 2 do
6:        for inPar := 0 to 2 do
7:          for later := 0 to 2 do begin
8:            if newInMe = 0 then begin
9:              newInPar := later;
10:             newLater := 2 - inMe;
11:           end else begin
12:             newInPar := 0;
13:             newLater := 0;
14:           end
15:           res[newInMe][newInPar][newLater] :=
16:             min(res[newInMe][newInPar][newLater],
17:                 newInMe + t[inMe][inPar][later]);
18:       end
19:   return res;
20: end
```

Dostosowanie funkcji *dfs* z sekcji o programowaniu dynamicznym do warunków naszego zadania pozostawiamy już jako ćwiczenie dla Czytelnika.

Wynik

Ostatnim krokiem, który musimy wykonać po przetworzeniu całego drzewa, jest odczytanie wyniku. Jak pamiętamy, parametr *inPar* oznacza, ile jeszcze potrzeba nadajników w ojcu wierzchołka v. W przypadku korzenia całego drzewa, ze względu na brak możliwości umieszczenia jakichkolwiek nadajników wyżej, musi być on równy zeru. Parametr *later* odnosił się do nadajników, które są wymagane, aby spełnić warunek dla krawędzi ponad wierzchołkiem v. Jednak dla korzenia całego drzewa takiej krawędzi nie ma, zatem wartość tego parametru w korzeniu jest dla nas nieistotna. W związku z tym, wybieramy stan o minimalnym wyniku spośród stanów $res[inMe][0][later]$, gdzie *res* jest tablicą wyników dla korzenia, a *inMe* i *later* przyjmują dowolne całkowite wartości z przedziału $[0, 2]$.

Całe rozwiązanie wykonuje liniową liczbę scaleń poddrzew i w każdym sprawdza $3^3 \cdot 3^3 = 729$ możliwości. Daje nam to asymptotycznie złożoność $O(n)$. Trzeba jednak być świadomym, że jest to rozwiązanie z wyjątkowo dużą stałą, zatem będzie znacznie wolniejsze od większości algorytmów o złożonościach liniowych dla podobnych rozmiarów danych.

Rozwiązanie opisane powyżej było wystarczająco dobre, żeby uzyskać maksymalną punktację. Jego kod można znaleźć w plikach `nad.cpp` oraz `nad2.cpp`.

Trochę szybciej

Zawodnicy, którzy chcieli przyspieszyć swój program, mogli to zrobić poprzez zmniejszenie stałego czynnika w czasie działania programu. Jedna z optymalizacji, które można zastosować, aby przyspieszyć najbardziej czasochłonny proces scalania poddrzew, jest taka, że jeżeli dla pojedynczego poddrzewa (ale już nie ich zbioru) zachodzi $inMe2 > 0$, to pozostałe parametry (*inPar2* oraz *later2*) są zerowe. Druga z optymalizacji polega na tym, że jeżeli $inPar2 > 0$, to spełniamy automatycznie wszystkie krawędzie od korzeni poddrzew do wspólnego ojca, zatem można założyć, że $later2 = 0$. Daje to rozwiązanie działające oczywiście cały czas w złożoności $O(n)$, jednak w każdym scaleniu zamiast 729 rozważamy jedynie $27 \cdot 7 = 189$ przypadków. Takie rozwiązania na zawodach nie były odróżniane od powyższego i również dostawały 100 punktów. Implementację takiego podejścia można znaleźć w pliku `nad1.cpp`.

Podzadanie 3: każda komora ma maksymalnie trzech sąsiadów

W zadaniu znalazło się podzadanie, w którym dodatkowo było nałożone ograniczenie górne na liczbę sąsiadów danego wierzchołka. W rzeczywistości oznacza ono, że jeśli dobrze ukorzenimy drzewo (w wierzchołku o maksymalnie dwóch sąsiadach), to otrzymamy drzewo, w którym każdy wierzchołek będzie miał co najwyżej dwóch synów.

Dzięki temu jesteśmy w stanie scalać wszystkie poddrzewa naraz i nie jest nam potrzebna obserwacja (być może trochę nieintuicyjna), że można je dołączać jedno po drugim.

Takie rozwiązanie przechodziło całe trzecie podzadanie. Można je obejrzeć w pliku `nadb2.cpp`.

Marek Sokołowski
Treść zadania, Opracowanie

Paweł Parys
Program

Dostępna pamięć: 256/64 MB.

OI, etap I, 19.10–16.11.2015

Nim z utrudnieniem

Ulubioną rozrywką Alicji i Bajtazara jest gra Nim. Do gry potrzebne są żetony, podzielone na kilka stosów. Dwaj gracze na przemian zabierają żetony ze stosów – ten, na którego przypada kolej, wybiera dowolny stos i usuwa z niego dowolną dodatnią liczbę żetonów. Gracz, który nie może wykonać ruchu, przegrywa[1].

Alicja zaproponowała Bajtazarowi kolejną partyjkę Nima. Aby jednak tym razem uczynić grę ciekawszą, gracze umówili się między sobą na dodatkowe warunki. Żetony, których było m, Alicja podzieliła na n stosów o licznościach $a_1, a_2, \ldots, a_n$. Zanim rozpocznie się rozgrywka, Bajtazar może wskazać niektóre spośród stosów, które zostaną natychmiast usunięte z gry. Liczba usuniętych stosów musi być jednak podzielna przez pewną ustaloną liczbę d, a ponadto Bajtazar nie może usunąć wszystkich stosów. Potem rozgrywka będzie toczyć się już normalnie, a rozpocznie ją Alicja.

Niech k oznacza liczbę sposobów, na które Bajtazar może wskazać stosy do usunięcia tak, aby mieć pewność, że wygra partię niezależnie od posunięć Alicji. Twoim zadaniem jest podanie reszty z dzielenia k przez $10^9 + 7$.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie dodatnie liczby całkowite n i d oddzielone pojedynczym odstępem, oznaczające odpowiednio liczbę stosów i ograniczenie „podzielnościowe" zabieranych stosów.

Drugi wiersz opisuje stosy i zawiera ciąg n dodatnich liczb całkowitych $a_1, a_2, \ldots, a_n$ pooddzielanych pojedynczymi odstępami, gdzie a_i oznacza liczbę żetonów na i-tym stosie.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, równą liczbie sposobów (modulo $10^9 + 7$), na które Bajtazar może usunąć stosy tak, aby później na pewno zwyciężyć.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
5 2 1 3 4 1 2	2

Wyjaśnienie do przykładu: *Bajtazar może zabrać 2 lub 4 stosy. Wygra tylko wtedy, gdy zabierze stosy o licznościach 1 i 4 (może to zrobić na dwa sposoby).*

[1] W Internecie łatwo znaleźć więcej informacji na temat gry Nim, a w szczególności opis strategii wygrywającej w tej grze.

Testy „ocen”:

`1ocen`: *$n = 9$, $d = 2$, wynikiem jest 0,*

`2ocen`: *$n = 12$, $d = 4$,*

`3ocen`: *$n = 30$, $d = 10$, wszystkie stosy mają po 30 żetonów,*

`4ocen`: *$n = 500\,000$, $d = 2$, wszystkie stosy mają wysokość 1.*

Ocenianie

Zestaw testów dzieli się na podane poniżej podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

We wszystkich podzadaniach zachodzą warunki $n \leqslant 500\,000$, $d \leqslant 10$, $a_i \leqslant 1\,000\,000$. Ponadto sumaryczna liczba żetonów $m = a_1 + a_2 + \ldots + a_n$ jest nie większa niż 10 000 000. Zwróć uwagę, że limit pamięci jest różny dla różnych podzadań.

Podzadanie	Dodatkowe warunki	Limit pamięci	Liczba punktów
1	$n \leqslant 20$, $a_1, \ldots, a_n \leqslant 1000$	*256 MB*	*10*
2	$n \leqslant 10\,000$, $a_1 \ldots, a_n \leqslant 1000$	*256 MB*	*18*
3	$d \leqslant 2$	*256 MB*	*25*
4	*brak*	*256 MB*	*27*
5	*brak*	*64 MB*	*20*

Rozwiązanie

W zadaniu prosiliśmy Uczestników o pomoc Bajtazarowi w wygraniu niewielkiej modyfikacji gry Nim. Aby więc móc rozwiązać ten problem, powinniśmy najpierw poznać kilka informacji na temat tej dość znanej gry matematycznej.

Teoria gier dla początkujących

Najpierw potrzebujemy definicji funkcji alternatywy wykluczającej (`xor`); dalej będziemy oznaczać ją $\oplus$. Aby wyznaczyć $a \oplus b$, czyli wartość `xor` liczb całkowitych a oraz b, zapisujemy obie liczby w systemie binarnym, a następnie dodajemy je pisemnie – z tą różnicą, że nie wykonujemy przeniesień. Poniższy przykład opisuje wykonanie operacji `xor` dla liczb $9 = 1001_{(2)}$ oraz $19 = 10011_{(2)}$:

$$\begin{array}{cccccccc} & & & 1 & 0 & 0 & 1 & \\ \oplus & 1 & 0 & 0 & 1 & 1 & & \\ \hline & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & = 26_{(10)} & \end{array}$$

Odpowiada to oczywiście wstawieniu jedynki w danej kolumnie wyniku, gdy znajdowało się w niej nieparzyście wiele jedynek. W przeciwnym przypadku wstawiamy zero.

Funkcję tę można uogólnić na więcej liczb całkowitych – dokładnie w ten sam sposób definiujemy `xor` wielu liczb $a_1 \oplus a_2 \oplus \ldots \oplus a_n$. Łatwo zauważyć, że działanie $\oplus$ jest łączne oraz przemienne, zatem nie ma znaczenia, w jakiej kolejności obliczamy `xor` wielu liczb.

Operacja $a \oplus b$ jest wbudowana w dostępne języki programowania: w C/C++ jest to `a ^ b`, zaś w Pascalu `a xor b`.

Kto wygra Nima?

Okazuje się, że istnieje bardzo prosty sposób na sprawdzenie, kto wygra rozgrywkę Nima:

Twierdzenie 1. *Niech $a_1, a_2, \ldots, a_n$ będą liczbami żetonów na kolejnych stosach w grze Nim. Gracz rozpoczynający grę może wygrać przy optymalnej grze przeciwnika wtedy i tylko wtedy, gdy $a_1 \oplus a_2 \oplus \ldots \oplus a_n \neq 0$.*

Dowód tego twierdzenia można znaleźć w Internecie, jak i w opracowaniu zadania *Kamyki* z XVI Olimpiady Informatycznej [1].

Jako że w naszym problemie Bajtazar jest drugim graczem, dąży on do tego, by na początku właściwej rozgrywki `xor` wszystkich wysokości stosów wynosił 0. Tak więc zadanie sprowadza się do tego, by policzyć liczbę sposobów usunięcia pewnej liczby stosów z rozgrywki tak, aby:

- liczba usuniętych stosów była podzielna przez d ($d \leqslant 10$),
- pozostał co najmniej jeden stos,
- `xor` wysokości wszystkich stosów wynosił 0.

Rozwiązanie przeglądające wszystkie możliwe podzbiory usuniętych stosów i sprawdzające, czy wszystkie trzy warunki zachodzą, zostało zaimplementowane w pliku `nims12.cpp`. Działa w czasie $O(2^n)$ i przechodzi testy w pierwszym podzadaniu.

Programowanie dynamiczne

Znacznie lepsze rozwiązanie można uzyskać, stosując metodę programowania dynamicznego. Zauważmy, że drugi warunek z poprzedniego podrozdziału można łatwo usunąć, ponieważ wpływa on na wynik tylko wtedy, gdy łączna liczba stosów n jest podzielna przez d. Jeśli zignorujemy drugi warunek, otrzymamy wówczas wynik o 1 za duży (wliczymy bowiem do wyniku sytuację, w której na początku usuniemy wszystkie stosy; jest ona wygrywająca dla Bajtazara). Możemy więc wymazać drugi warunek, a potem na samym końcu w razie potrzeby odjąć od wyniku 1.

Teraz próbujmy dodawać stosy po jednym – dla przykładu, od lewej do prawej. W każdym momencie decydujemy, czy pozostawić stos na planszy, czy go usunąć. Zauważmy, że w każdym momencie wystarczy nam pamiętać jedynie resztę z dzielenia przez d liczby usuniętych stosów oraz `xor` pozostawionych do tej pory stosów.

Mamy więc już szkielet programowania dynamicznego. Przez $dp[i][r][x]$ oznaczymy liczbę sposobów usunięcia spośród pierwszych i stosów podzbioru stosów o liczności

dającej resztę r z dzielenia przez d w taki sposób, że pozostałe z tych i stosów mają `xor` równy x. Jeśli oznaczymy przez A najmniejszą potęgę dwójki przekraczającą największą z wysokości stosów, to zachodzi $0 \leqslant i \leqslant n$, $0 \leqslant r < d$, $0 \leqslant x < A$. Aby nie musieć operować na dużych liczbach, w zadaniu jesteśmy proszeni jedynie o podanie reszty z dzielenia wyniku przez $M = 10^9 + 7$. Wobec tego w tablicy dp wystarczy pamiętać jedynie wartości modulo M.

Dla zerowej liczby stosów ($i = 0$) wszystko jest proste – musi zajść $r = 0$ (usuwamy zero stosów) oraz $x = 0$ (`xor` zerowej liczby stosów to 0). Tak więc $dp[0][0][0] = 1$ oraz $dp[0][\star][\star] = 0$ dla pozostałych wartości w tablicy.

Przypuśćmy, że rozważyliśmy do tej pory $i-1$ stosów i dokładamy i-ty, o wysokości a_i. Chcemy obliczyć $dp[i][r][x]$. Mamy dwie możliwości:

1. Zachowujemy i-ty stos. Liczba usuniętych stosów w porównaniu do poprzedniego stanu pozostaje niezmieniona. Natomiast poprzedni `xor` wysokości pozostawionych stosów musiał wynosić $x \oplus a_i$ (teraz dokładamy do poprzedniego `xor`-a wartość a_i i musi wyjść x; mamy zaś $(x \oplus a_i) \oplus a_i = x \oplus (a_i \oplus a_i) = x$). Do wyniku dodajemy liczbę sposobów dojścia do poprzedniego stanu równą $dp[i-1][r][x \oplus a_i]$.

2. Odrzucamy i-ty stos. Liczba usuniętych stosów krok wcześniej była o jeden mniejsza, za to `xor` się nie zmienił. Dodajemy więc do wyniku wartość $dp[i-1][(r-1) \bmod d][x]$. Zakładamy tożsamość $(-1) \bmod d = d-1$, która jest prawdziwa w matematyce, ale nie zachodzi w większości języków programowania.

Ostatecznie więc

$$dp[i][r][x] = (dp[i-1][r][x \oplus a_i] + dp[i-1][(r-1) \bmod d][x]) \bmod M.$$

Odpowiedź odczytujemy jako $dp[n][0][0]$ (rozpatrzyliśmy n stosów, wyrzuciliśmy liczbę stosów podzielną przez d, pozostałe stosy mają `xor` równy 0). W razie potrzeby odejmujemy od wyniku 1.

Złożoność czasowa i pamięciowa takiego rozwiązania wynosi $O(ndA)$. Niestety, proste obliczenia prowadzą do wniosku, że tablica dp zajmie 390 MB pamięci w drugim podzadaniu. Można jednak mocno ograniczyć zużycie pamięci na jeden z dwóch poniższych sposobów:

1. Zauważmy, że $dp[i][\star][\star]$ zależy tylko od $dp[i-1][\star][\star]$. Możemy więc utrzymywać w pamięci tylko dwie ostatnie podtablice: $dp[i][\star][\star]$ i $dp[i-1][\star][\star]$. Po obliczeniu nowej podtablicy możemy nadpisać poprzednią, ponieważ nie będzie ona nam już do niczego potrzebna.

   Metodę tę można zaimplementować, zmniejszając pierwszy wymiar do 2 elementów i utrzymując w nim resztę z dzielenia wartości i przez 2. Wtedy zamiast dokonywać przejścia $dp[i-1][\star][\star] \to dp[i][\star][\star]$, zerujemy podtablicę $dp[i \bmod 2][\star][\star]$ i wykonujemy przejście $dp[(i-1) \bmod 2][\star][\star] \to dp[i \bmod 2][\star][\star]$. Wynik znajdziemy w pozycji $dp[n \bmod 2][0][0]$.

2. Druga obserwacja jest bardziej wnikliwa. Na początek przypomnijmy, że $(x \oplus a_i) \oplus a_i = x$. Wartości $dp[i][\star][x]$ oraz $dp[i][\star][x \oplus a_i]$ zależą tylko od

$dp[i-1][\star][x]$ i $dp[i-1][\star][x \oplus a_i]$. Możemy więc wykonywać obliczenia niezależnie dla rozłącznych par $(x, x \oplus a_i)$, każdorazowo tworząc dwuwymiarową tablicę dp' o wymiarach $d \times 2$ i następie nadpisując fragment oryginalnej tablicy. W ten sposób możemy całkowicie zignorować pierwszy wymiar tablicy (i).

Obie metody ograniczają zapotrzebowanie na pamięć do $O(dA)$. Implementacja drugiego sposobu znajduje się w pliku `nims3.cpp`. Wystarcza do zdobycia punktów za dwa pierwsze podzadania.

Rozwiązanie wzorcowe

W rozwiązaniu wzorcowym skorzystamy w końcu z ograniczenia $m \leqslant 10^7$ na sumę wysokości stosów $a_1 + a_2 + \ldots + a_n$. Wykorzystamy następujący fakt:

Fakt 1. *Niech* $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_i$. *Wtedy* $a_1 \oplus a_2 \oplus \ldots \oplus a_i < 2a_i$.

Dowód: Wybierzmy potęgę dwójki 2^t taką, że $2^t \leqslant a_i < 2^{t+1}$. Ze względu na to, że wszystkie `xor`-owane liczby są mniejsze niż potęga dwójki 2^{t+1} (gdyż w szczególności nie przekraczają a_i), to ich `xor` też jest mniejszy niż 2^{t+1}. Jednak $2a_i \geqslant 2^{t+1}$. To kończy dowód. ■

Będziemy postępować następująco: posortujmy wszystkie wysokości stosów. Teraz $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_n$. W momencie, gdy liczymy $dp[i][\star][\star]$, znajdujemy potęgę dwójki 2^t taką, że $2^t \leqslant a_i < 2^{t+1}$, a następnie liczymy wszystkie wartości $dp[i][r][x]$ tylko dla $0 \leqslant x < 2^{t+1}$. Nie jesteśmy w stanie uzyskać $x \geqslant 2^{t+1}$ za pomocą dotychczas rozważonych stosów $a_1, \ldots, a_i$, więc dla większych x wartości dp będą równe 0.

Czas obliczenia $dp[i][\star][\star]$ nie przekroczy $2da_i$, ponieważ na podstawie powyższego faktu liczymy maksymalnie $2a_i$ podtablic $dp[i][\star][x]$, zaś czas obliczenia pojedynczej to $O(d)$. Łączny czas działania programowania dynamicznego możemy więc oszacować z góry przez

$$d \cdot (2a_1 + 2a_2 + \ldots + 2a_n) = 2dm = O(dm).$$

Do tego dochodzi czas sortowania wysokości stosów ($O(n \log n)$) oraz zaniedbywalny czas wyznaczania odpowiednich potęg dwójki 2^{t+1}.

Złożoność obliczeniowa jest już w zupełności wystarczająca do zaliczenia wszystkich testów, musimy jednak znów poradzić sobie z problemem ograniczonej pamięci. Możemy zaprząc do tego zadania te same sposoby, które omawialiśmy przy poprzednim rozwiązaniu.

1. Utrzymujemy dwie ostatnie podtablice $dp[i-1][\star][\star]$ oraz $dp[i][\star][\star]$. Jako że pojedyncza liczba 32-bitowa zajmuje 4 bajty, to zużycie pamięci wynosi $2 \cdot \max_d \cdot \max_A \cdot 4\,\mathrm{B} = 2 \cdot 10 \cdot 2^{20} \cdot 4\,\mathrm{B} = 80\,\mathrm{MB}$. Pozwala to na zaliczenie podzadań 1–4 z podwyższonym limitem pamięci. Takie rozwiązanie zaimplementowano w plikach `nims2.cpp`, `nims4.cpp` oraz `nims14.cpp`.

2. Usuwamy pierwszy wymiar tablicy i liczymy $dp[\star][x], dp[\star][x \oplus a_i]$ niezależnie dla każdej pary $(x, x \oplus a_i)$ na podstawie poprzednich wartości tablicy. Zużycie pamięci jest oczywiście dwukrotnie niższe (40 MB). Pozwala to już na zdobycie pełnej punktacji. Przykładowe implementacje znajdują się w plikach `nim.cpp`, `nim1.pas` oraz `nim15.cpp`.

Inne rozwiązanie – przypadek $d \leqslant 2$

Istnieje dość prosty sposób na rozwiązanie trzeciego podzadania. Zauważmy najpierw, że przypadek $d = 2$ można prosto sprowadzić do przypadku $d = 1$, do każdej liczby dodając dodatkowy bit (2^{20}) zawsze równy 1. Wtedy `xor` podzbioru liczb jest równy 0 tylko wtedy, gdy w oryginalnych wysokościach stosów `xor` był równy 0 oraz liczba stosów była parzysta (ten warunek jest wymuszany przez dodatkowy bit). Tak samo jak poprzednio, odejmujemy 1 od wyniku tylko w przypadku, gdy d jest dzielnikiem n. Pozostało więc rozwiązać przypadek $d = 1$.

Udowodnijmy następujące fakty:

Fakt 2. *Niech S będzie zbiorem liczb, które można uzyskać za pomocą* `xor`*-a pewnej (być może zerowej) liczby wysokości stosów. Jeśli $x, y \in S$, to też $x \oplus y \in S$.*

Dowód: Niech $X = \{x_1, \ldots, x_p\}$ będzie podzbiorem stosów wykorzystanych do uzyskania `xor` równego x, zaś $Y = \{y_1, \ldots, y_q\}$ – `xor` równego y. Wtedy oczywiście $x_1 \oplus \ldots \oplus x_p \oplus y_1 \oplus \ldots \oplus y_q = x \oplus y$. Jeśli pewien stos a_i znajduje się zarówno w X, jak i w Y, to w powyższym równaniu wystąpi dwukrotnie i można go pominąć, gdyż $a_i \oplus a_i = 0$. Rozważmy zbiór stosów $Z = \{x : x \text{ jest doiładnie w jednym ze zbiorów } X, Y\}$. Wtedy `xor` wysokości stosów ze zbioru Z jest równy dokładnie $x \oplus y$. ■

Fakt 3. *Każdy z elementów S można uzyskać na tyle samo sposobów.*

Dowód: Prosta indukcja po liczbie stosów i. Na początku $i = 0$ i baza jest oczywista (S jest jednoelementowy, tj. $S = \{0\}$). Weźmy teraz zbiór S uzyskany dla stosów $a_1, \ldots, a_{i-1}$, zaś z niech będzie liczbą sposobów na uzyskanie każdego spośród elementów S. Dołóżmy stos a_i. W opisie dp mamy

$$dp[i][0][x] = dp[i-1][0][x] + dp[i-1][0][x \oplus a_i].$$

1. Jeśli $a_i \in S$, to nie może zajść jednocześnie $x \in S$, $x \oplus a_i \notin S$ (natychmiastowy wniosek z faktu 2) – podobnie nie zajdzie jednocześnie $x \notin S$, $x \oplus a_i \in S$. Tak więc albo oba składniki będą równe z, albo oba będą równe 0. Wobec tego $dp[i][0][\star] \in \{0, 2z\}$.

2. Jeśli $a_i \notin S$, to nie może być jednocześnie $x, x \oplus a_i \in S$ (wtedy $a_i = x \oplus (x \oplus a_i) \in S$). Stąd maksymalnie jeden składnik jest równy z. Tak więc $dp[i][0][\star] \in \{0, z\}$.

W obu przypadkach wszystkie niezerowe wartości dp są równe. ■

Z powyższych faktów i ich dowodów wynika bardzo prosty algorytm – będziemy utrzymywać zbiór S niezerowych pozycji w dp oraz wartość tych pozycji z modulo M. Początkowo $S = \{0\}$, $z = 1$. Dla każdego nowego stosu a_i:

1. jeśli $a_i \in S$, to z rośnie dwukrotnie;

2. jeśli $a_i \notin S$, to do S dodajemy wszystkie wysokości stosów postaci $x \oplus a_i$ dla $x \in S$ (widzimy ze wzoru na dp, że tylko one staną się niezerowe).

Wynikiem jest ostatecznie $z \bmod M$ (minus jeden w razie potrzeby). Rozwiązanie bazujące na podobnych pomysłach zostało zaimplementowane w pliku `nimb11.cpp`. Przechodzi ono wszystkie testy w trzecim podzadaniu.

Miłośnikom matematyki podpowiemy, że S posiada strukturę przestrzeni liniowej nad ciałem rzędu 2 ($1 \oplus 1 = 0$), rozpinanej przez zbiór wysokości stosów. Przedstawiony wyżej algorytm można interpretować jako metodę eliminacji Gaussa na odpowiednio skonstruowanej macierzy. Każdą liczbę a_i zapisujemy w postaci binarnej: $a_i = a_{i,0} \cdot 2^0 + a_{i,1} \cdot 2^1 + \ldots + a_{i,20} \cdot 2^{20}$. Konstruujemy macierz binarną A (z dodawaniem bitów takim, jak w operacji `xor`):

$$A = \begin{pmatrix} a_{1,0} & a_{1,1} & \ldots & a_{1,20} \\ a_{2,0} & a_{2,1} & \ldots & a_{2,20} \\ a_{3,0} & a_{3,1} & \ldots & a_{3,20} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,0} & a_{n,1} & \ldots & a_{n,20} \end{pmatrix}.$$

Okazuje się, że wynikiem jest $2^{n-\operatorname{rk} A}$, gdzie $\operatorname{rk} A$ jest tak zwanym *rzędem* macierzy A, który można łatwo wyznaczyć metodą eliminacji Gaussa.

Rozwiązania błędne

- Uczestnicy mogli zapomnieć o odjęciu 1 w programowaniu dynamicznym w przypadku $d \mid n$. Przykład takiego rozwiązania jest w pliku `nimb5.cpp`. Rozwiązanie nie otrzymuje żadnych punktów, ale błąd jest wyłapywany przez dostępne publicznie testy `ocen`.

- Program zapominający o tym, iż trzeba podać resztę z dzielenia wyniku przez $10^9 + 7$, znajduje się w pliku `nimb6.cpp`. Przechodzi pierwsze podzadanie.

- Bardzo trudnym do wychwycenia przypadkiem (w szczególności podczas testowania z wygenerowanymi losowo testami) była sytuacja, w której musimy odjąć 1 od wyniku podzielnego przez $10^9 + 7$. Reszta przed odjęciem jedynki wynosiła 0. Rozwiązanie wypisujące w tym przypadku -1 zamiast 10^9+6 można znaleźć w pliku `nimb7.cpp`. Nie przechodzi pojedynczego testu w podzadaniu 2.

Jacek Tomasiewicz
Treść zadania, Opracowanie

Bartosz Kostka
Program

Dostępna pamięć: 256 MB.

OI, etap I, 19.10–16.11.2015

Park wodny

Park wodny Aquabajt bierze udział w konkursie na największy basen. Teren parku, na którym zlokalizowane są baseny, ma kształt kwadratu o boku długości n i jest podzielony na n^2 segmentów, z których każdy jest kwadratem o boku długości 1. Każdy z segmentów może być basenikiem albo alejką między basenikami. Baseniki połączone bezpośrednio ze sobą (czyli będące segmentami stykającymi się bokami) tworzą większe baseny. Obecnie w parku wodnym każdy basen ma kształt prostokąta.

Dyrekcja Aquabajtu postanowiła zwiększyć swoje szanse na wygraną w konkursie, przebudowując park. Ze względu na ograniczony czas i fundusze zdecydowano o przekształceniu co najwyżej dwóch segmentów z alejkami na baseniki. Pomóż władzom parku uzyskać basen złożony z maksymalnej liczby baseników. Zakładamy, że po przebudowie największy basen nie musi być już prostokątem.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną dodatnią liczbę całkowitą n oznaczającą wielkość parku wodnego.

W następnych n wierszach znajduje się dwuwymiarowa mapa parku: każdy z tych wierszy zawiera słowo złożone z n liter. Litera `A` *oznacza segment z alejką, natomiast litera* `B` *oznacza basenik. Możesz założyć, że w opisie znajduje się co najmniej jedna litera* `B`.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą oznaczającą wielkość największego basenu, jaki można uzyskać.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`5`
`BBBAB`
`BBBAB`
`AAAAA`
`BBABA`
`BBAAB`
	`14`

Testy „ocen”:

`1ocen`: $n = 10$, *tylko jeden basen na planszy,*

`2ocen`: $n = 10$, *cała plansza pokryta basenem,*

`3ocen`: $n = 1000$, *szachownica.*

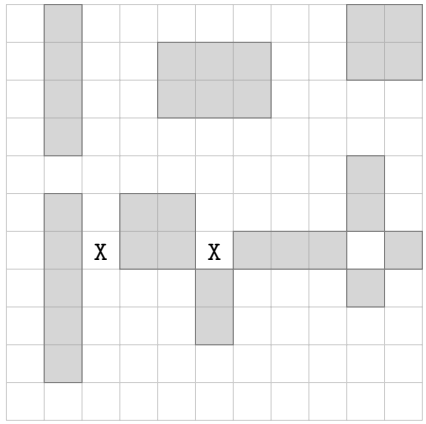
Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n \leqslant 10$	*11*
2	$n \leqslant 50$, *liczba basenów na początku nie przekroczy 80*	*11*
3	$n \leqslant 60$	*22*
4	$n \leqslant 1000$, *na początku każdy basen jest prostokątem* 1×1	*22*
5	$n \leqslant 1000$	*34*

Rozwiązanie

W zadaniu mamy daną mapę podzieloną na n^2 segmentów. Każdy z segmentów może być basenikiem albo alejką między basenikami. Baseniki połączone bezpośrednio ze sobą tworzą większe baseny, które są początkowo prostokątami (zaznaczone na rysunku kolorem szarym). Chcielibyśmy zamienić dwie alejki na baseniki, tak aby utworzyć jak największy basen (który nie musi być już prostokątem).



W optymalnym rozwiązaniu dla powyższego przykładu zbudujemy dwa baseniki oznaczone znakiem `X`, otrzymując jeden basen złożony z $5+1+4+1+2+3 = 16$ baseników.

Rozwiązanie brutalne $O(n^6)$

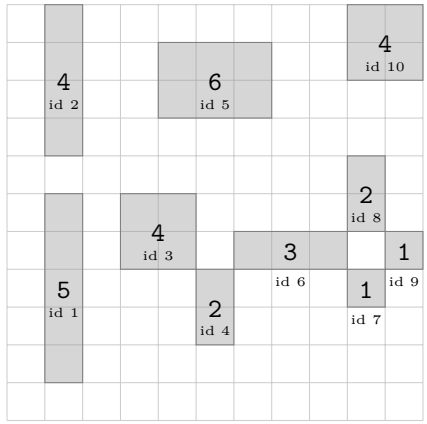
Sprawdzamy każdą możliwą parę segmentów i wybieramy tę, której zamiana na baseniki utworzy największy basen. Aby znaleźć największy basen po takiej zamianie, możemy wyobrazić sobie graf, w którym wierzchołkami są segmenty zawierające baseniki, a krawędzie łączą wierzchołki odpowiadające sąsiednim segmentom, a następnie

użyć algorytmu przeszukiwania grafu w głąb (DFS). Jako że wszystkich par segmentów jest $O(n^4)$, natomiast znalezienie największego basenu zajmuje czas liniowy względem liczby segmentów, złożonością czasową tego rozwiązania jest $O(n^6)$. Warto zwrócić uwagę na przypadek szczególny (patrz test `2ocen`), w którym cały park jest od początku pokryty basenem, więc nie ma żadnego segmentu niebędącego basenikiem (natomiast nie jest możliwe, żeby początkowo tylko jeden segment nie był basenikiem).

Rozwiązanie to zaimplementowane jest w pliku `pars1.cpp`. Za poprawne zaprogramowanie takiego rozwiązania na zawodach można było uzyskać około 10% punktów.

Rozwiązanie wolne $O(n^4)$

Na początku obliczamy wielkość wszystkich basenów w czasie $O(n^2)$, zapisując dla każdego basenu jego identyfikator i rozmiar. Można to zrobić na kilka sposobów, np. używając opisanego wcześniej grafu.



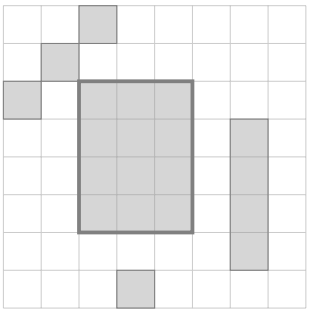
W praktyce informacje te najwygodniej zapisać w każdym segmencie basenu. Następnie rozpatrujemy każdą możliwą parę segmentów i sprawdzamy, które baseny zostaną połączone, gdy te segmenty zmienimy w baseniki. Zauważmy, że każdy segment sąsiaduje z co najwyżej czterema basenami, więc dodanie dwóch segmentów powoduje połączenie tylko stałej liczby basenów. Jako że znamy identyfikatory basenów i ich rozmiary, to w czasie stałym obliczamy całkowity rozmiar powstałego basenu.

Implementacja takiego rozwiązania znajduje się w plikach `pars3.cpp` i `pars9.cpp`. Rozwiązanie tego typu otrzymywało na zawodach około 40% punktów.

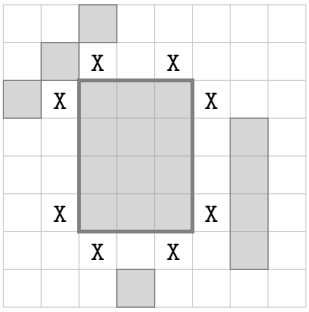
Rozwiązanie wzorcowe $O(n^2)$

Załóżmy na początek, że zmieniana para segmentów sąsiaduje ze sobą. Możemy rozważyć wszystkie takie pary, ponieważ będzie ich tylko $O(n^2)$. Dla każdej pary w czasie stałym sprawdzamy jak wyżej, które baseny zostaną połączone, i zapamiętujemy najlepszy wynik.

Dalej zakładamy, że zmieniane segmenty nie będą ze sobą sąsiadować. Zauważmy, że w tym przypadku w optymalnym rozwiązaniu segmenty te będą sąsiadować z jednym, wybranym basenem. Każdy taki basen będziemy rozpatrywali oddzielnie. Przykładowo, na rysunku poniżej rozważmy basen zaznaczony pogrubioną linią.



Znajdźmy wszystkie segmenty, za pomocą których możemy rozbudować wybrany basen. Liczba takich segmentów będzie liniowa względem obwodu rozpatrywanego basenu. Prawie wszystkie segmenty łączą się z co najwyżej jednym innym basenem. Jedyne segmenty, które mogą się łączyć z dwoma (lub nawet trzema) basenami, znajdują się na rogach. Dokładniej, są po dwa takie segmenty na każdym z rogów, co daje łącznie maksymalnie osiem różnych segmentów (oznaczonych na rysunku znakiem `X`).



Niech k oznacza obwód rozważanego basenu. Wszystkie narożne segmenty rozpatrujemy ze wszystkimi innymi segmentami (również tymi, które nie są na rogach), co daje maksymalnie $8k$ par segmentów. Natomiast z pozostałych segmentów (które nie są na rogach) wybieramy te dwa, które łączą się z największymi różnymi basenami – można je znaleźć w czasie $O(k)$.

Jako że suma obwodów wszystkich basenów nie przekroczy $O(n^2)$, otrzymujemy rozwiązanie wzorcowe, działające w czasie kwadratowym. Przykładową implementację można znaleźć w plikach `par.cpp` i `par2.cpp`.

Zawody II stopnia

opracowania zadań

Jakub Radoszewski — Treść zadania, Opracowanie
Marek Sommer — Program

Dostępna pamięć: 1024 MB. *OI, etap II, dzień próbny, 9.02.2016*

Świąteczny łańcuch

Każdego roku na święta Bożego Narodzenia Bajtazar dekoruje swój dom łańcuchem złożonym z różnokolorowych lampek. Tym razem Bajtazar zamierza samemu dobrać kolory lampek, które będą wchodziły w skład łańcucha. Bajtazar ma w głowie pewne wymagania estetyczne, które streszczają się w tym, że pewne fragmenty łańcucha powinny mieć identyczny układ lampek jak inne. Ponadto żona Bajtazara poprosiła go, aby tegoroczny łańcuch był jak najbardziej urozmaicony, co Bajtazar rozumie tak, że powinno w nim być jak najwięcej różnych kolorów lampek. Pomóż naszemu bohaterowi stwierdzić, ile kolorów lampek będzie musiał kupić.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n oraz m ($n \geqslant 2$, $m \geqslant 1$) oddzielone pojedynczym odstępem, określające liczbę lampek w planowanym łańcuchu i liczbę wymagań estetycznych Bajtazara. Zakładamy, że kolejne lampki łańcucha będą ponumerowane od 1 do n. Każdy z m kolejnych wierszy opisuje jedno z wymagań za pomocą trzech liczb całkowitych a_i, b_i i l_i ($1 \leqslant a_i, b_i, l_i$; $a_i \neq b_i$; $a_i, b_i \leqslant n - l_i + 1$) oddzielonych pojedynczymi odstępami. Taki opis oznacza, że fragmenty łańcucha złożone z lampek o numerach $\{a_i, \ldots, a_i + l_i - 1\}$ oraz $\{b_i, \ldots, b_i + l_i - 1\}$ powinny być jednakowe. Innymi słowy, lampki o numerach a_i oraz b_i powinny mieć taki sam kolor, podobnie lampki o numerach $a_i + 1$ oraz $b_i + 1$, i tak dalej aż do lampek o numerach $a_i + l_i - 1$ i $b_i + l_i - 1$.

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną dodatnią liczbę całkowitą k oznaczającą maksymalną liczbę różnych kolorów lampek, jakie mogą wystąpić w łańcuchu spełniającym wymagania estetyczne opisane na wejściu.

Przykład

Dla danych wejściowych:

```
10 3
1 6 3
5 7 4
3 8 1
```

poprawnym wynikiem jest:

```
3
```

natomiast dla danych wejściowych:

```
4 2
1 2 2
2 3 2
```

poprawnym wynikiem jest:

```
1
```

Wyjaśnienie do pierwszego przykładu: *Niech* `a`, `b` *i* `c` *oznaczają trzy różne kolory lampek. Przykładowy łańcuch spełniający wymagania Bajtazara i jego żony to* `abacbababa`.

Testy „ocen”:

`1ocen`: $n = 2000$, $m = 2$; *Bajtazar wymaga, aby fragmenty* $\{1, \ldots, 1000\}$ *i* $\{1001, \ldots, 2000\}$ *były równe oraz aby fragmenty* $\{1, \ldots, 500\}$ *i* $\{501, \ldots, 1000\}$ *były równe; w łańcuchu może wystąpić maksymalnie 500 kolorów lampek.*

`2ocen`: $n = 500\,000$, $m = 499\,900$; *i-te wymaganie jest postaci* $a_i = i$, $b_i = i + 100$, $l_i = 1$; *w łańcuchu może wystąpić maksymalnie 100 kolorów lampek.*

`3ocen`: $n = 80\,000$, $m = 79\,995$, *i-te wymaganie jest postaci* $a_i = i$, $b_i = i + 2$, $l_i = 4$; *w łańcuchu mogą wystąpić maksymalnie dwa kolory lampek.*

`4ocen`: $n = 500\,000$, $m = 250\,000$, *i-te wymaganie jest postaci* $a_i = 1$, $b_i = i + 1$, $l_i = i$; *łańcuch może składać się jedynie z lampek o tym samym kolorze.*

Ocenianie

Zestaw testów dzieli się na podzadania spełniające poniższe warunki. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n, m \leqslant 2000$	*30*
2	$n, m \leqslant 500\,000$, *wszystkie liczby* l_i *są równe 1*	*20*
3	$n, m \leqslant 80\,000$	*30*
4	$n, m \leqslant 500\,000$	*20*

Rozwiązanie

W zadaniu występuje ciąg n *lampek* ponumerowanych od 1 do n. Mamy danych m *wymagań* określających równość kolorów lampek w pewnych fragmentach tego ciągu. Naszym celem jest dobrać kolory wszystkich lampek w ciągu tak, aby spełnione były wszystkie wymagania i ponadto aby liczba użytych kolorów była jak największa. Pula kolorów jest nieograniczona. Wystarczy nam podać liczbę wykorzystanych kolorów.

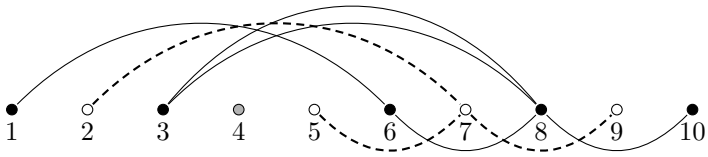
Pierwsze rozwiązanie: graf lampek

Spróbujmy najpierw zaproponować jakiekolwiek rozwiązanie zadania, nie bacząc na to, na ile będzie ono efektywne.

Zauważmy, że każde wymaganie polega na wymuszeniu równości kolorów pewnych par lampek. Dokładniej, wymaganie postaci (a, b, l) oznacza, że lampki o numerach $a + i$ oraz $b + i$, dla każdego $i = 0, \ldots, l - 1$, mają taki sam kolor. Jako że nie interesuje nas na razie efektywność rozwiązania, możemy na starcie rozbić każde takie wymaganie na l pojedynczych wymagań, postaci $(a + i, b + i, 1)$ dla $i = 0, \ldots, l - 1$.

W tym momencie przydatna okazuje się interpretacja grafowa. Skonstruujmy multigraf nieskierowany $G = (V, E)$ o zbiorze wierzchołków V i multizbiorze krawędzi E, w którym wierzchołki odpowiadają lampkom ($V = \{1, \ldots, n\}$), a krawędzie odpowiadają wymaganiom (dla każdego wymagania $(u, v, 1)$ w multizbiorze E umieszczamy

krawędź uv). Przypomnijmy, że multigraf to po prostu graf, w którym dopuszczamy powtórzenia krawędzi. Multigraf G nazwiemy *grafem lampek*.



Rys. 1: Graf lampek G skonstruowany dla pierwszego przykładu z treści zadania. Ma on trzy spójne składowe: pierwsza z nich zawiera wierzchołki $\{1, 3, 6, 8, 10\}$, druga wierzchołki $\{2, 5, 7, 9\}$, a trzecia tylko jeden wierzchołek 4.

Jeśli wierzchołki odpowiadające dwóm lampkom są połączone ścieżką w grafie lampek, to lampki te muszą mieć ten sam kolor. Podzielmy więc graf lampek na spójne składowe. Lampki odpowiadające wierzchołkom z tej samej spójnej składowej muszą mieć ten sam kolor, natomiast lampki odpowiadające wierzchołkom z różnych spójnych składowych mogą mieć różne kolory. Ponieważ zależy nam na wykorzystaniu jak największej liczby kolorów, najbardziej opłaca nam się każdej spójnej składowej przypisać inny kolor lampek. Wynikiem będzie więc liczba spójnych składowych w grafie lampek. Na rysunku 1 zobrazowaliśmy to na przykładzie z treści zadania.

Graf lampek ma $|V| = n$ wierzchołków oraz $|E| = O(nm)$ krawędzi. Spójne składowe w (multi)grafie możemy wyznaczyć w czasie $O(|V| + |E|)$ za pomocą przeszukiwania w głąb (DFS) lub – co w praktyce jest równie szybkie – w czasie $O((|V| + |E|) \log^* |V|)$ z wykorzystaniem struktury danych dla zbiorów rozłącznych (Find-Union)[1]. W ten sposób otrzymujemy rozwiązanie o złożoności czasowej $O(nm)$ (lub $O(nm \log^* n)$), które przechodzi pierwsze podzadanie. Można zauważyć, że rozwiązanie to jest wystarczające również dla drugiego podzadania, jako że warunek $l_i = 1$ gwarantuje, że graf lampek ma tylko m krawędzi.

Lepsze rozwiązanie: graf wymagań

Do zaliczenia kolejnego podzadania wystarczało rozwiązanie dzielące wymagania na fragmenty długości $\Theta(\sqrt{n})$. Można zaproponować kilka takich algorytmów; omówimy tu jeden z nich, który potem będziemy mogli jeszcze usprawnić.

Niech $p = \lfloor \sqrt{n} \rfloor$. *Długością* wymagania (a, b, l) nazwijmy liczbę l. Jeśli wszystkie wymagania mają długość mniejszą niż p, to możemy każde z nich przekształcić na wymagania długości 1 tak jak w poprzednim rozwiązaniu. Jeśli natomiast jakieś wymaganie ma długość $l \geqslant p$, to możemy je podzielić na pewną liczbę wymagań o długości p oraz jedno wymaganie o długości $l \bmod p < p$; tych pierwszych będzie co najwyżej $\frac{n}{p}$, a to ostatnie możemy rozbić na mniej niż p wymagań o długości 1. W ten sposób otrzymujemy łącznie co najwyżej:

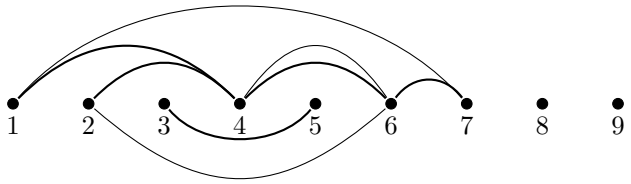
(a) mp wymagań o długości 1 oraz

[1] Więcej o obu sposobach podziału grafu na spójne składowe można przeczytać np. w książce [6].

(b) $m \cdot \frac{n}{p}$ wymagań o długości p.

Z wymaganiami typu (a) poradzimy sobie dokładnie tak jak w poprzednim rozwiązaniu. Skoncentrujmy się teraz na wymaganiach typu (b). Okazuje się, że jeśli jest ich dostatecznie dużo, to muszą one być redundantne, tzn. możemy je wówczas zastąpić niewielką liczbą równoważnych wymagań. W tym celu wprowadzimy pomocniczy multigraf $G' = (V', E')$, który nazwiemy *grafem wymagań*. Graf wymagań ma n wierzchołków: $V' = \{1, \ldots, n\}$, a krawędź $ab \in E'$ odpowiada wymaganiu (a, b, p). Mamy więc $|E'| = O(\frac{nm}{p}) = O(m\sqrt{n})$.

Jeśli dwa wierzchołki $a, b \in V'$ są połączone krawędzią w grafie wymagań, to fragmenty ciągu lampek o długości p zaczynające się na pozycjach a i b są równe. A zatem taka sama własność zachodzi także wtedy, gdy dwa wierzchołki są połączone ścieżką w grafie wymagań, czyli jeśli znajdują się w tej samej spójnej składowej. To oznacza, że do reprezentacji wszystkich wymagań znajdujących się w grafie G' wystarczy wziąć wymagania z dowolnego lasu rozpinającego grafu G'; patrz rys. 2. Taki las rozpinający ma oczywiście co najwyżej $n - 1$ krawędzi, a można go znaleźć chociażby za pomocą wspomnianego już algorytmu DFS w czasie $O(|V'| + |E'|) = O(n + m\sqrt{n})$.



Rys. 2: Graf wymagań G' odpowiadający wymaganiom: $(1, 4, 3)$, $(1, 7, 3)$, $(2, 4, 3)$, $(2, 6, 3)$, $(3, 5, 3)$, $(4, 6, 3)$ (dwukrotnie), $(6, 7, 3)$. Pogrubieniem zaznaczono las rozpinający tego grafu (wierzchołki izolowane możemy pominąć).

W ten sposób uzyskujemy co najwyżej $n - 1$ wymagań o długości p. Wszystkie te wymagania wraz z wymaganiami typu (a) rozbijamy na wymagania o długości 1, otrzymując graf lampek rozmiaru $O((n + m)p) = O((n + m)\sqrt{n})$. Następnie stosujemy do tego grafu poprzednie rozwiązanie. Całość – przetwarzanie grafu wymagań, a następnie grafu lampek – działa w czasie $O((n + m)\sqrt{n})$.

Rozwiązanie wzorcowe

Często bywa, że podział ciągu na fragmenty o długości $\sqrt{n}$ można zastąpić odpowiednią strukturą fragmentów o długościach będących potęgami dwójki, co pozwala zredukować w złożoności czasowej czynnik $\sqrt{n}$ do czynnika $\log n$. Tak samo możemy zrobić także i w tym zadaniu. Kluczowe spostrzeżenie jest takie, że metoda redukcji liczby wymagań za pomocą grafu wymagań działa tak samo dobrze dla dowolnej, ustalonej długości wymagania.

W tym rozwiązaniu będziemy konstruować grafy wymagań dla długości wymagań będących malejącymi potęgami dwójki: 2^k dla $k = \lfloor \log n \rfloor, \ldots, 0$. Przez W_k oznaczymy

zbiór wymagań, jakie pozostały nam do rozważenia przed krokiem k. Dla każdego k spełniony będzie niezmiennik, że wszystkie wymagania ze zbioru W_k są nie dłuższe niż 2^{k+1}. Zauważmy, że dla $k = \lfloor \log n \rfloor$ niezmiennik ten jest spełniony w sposób trywialny.

W kroku odpowiadającym danemu k wydzielimy wszystkie wymagania o długościach co najmniej 2^k. Spośród nich, wymagania dłuższe niż 2^k przekształcimy na pary wymagań o długości 2^k według wzoru:

$$(a, b, l) \quad \rightarrow \quad (a, b, 2^k),\ (a + l - 2^k, b + l - 2^k, 2^k).$$

Następnie z wszystkich wymagań o długości 2^k zbudujemy graf wymagań G'_k. Tak jak w poprzednim rozwiązaniu, zbiór krawędzi grafu G'_k możemy zastąpić lasem rozpinającym, zawierającym co najwyżej $n-1$ krawędzi. W ten sposób przekształcamy zbiór W_k w zbiór W_{k-1}, w którym wszystkie wymagania są długości co najwyżej 2^k. Kontynuujemy to postępowanie dla kolejnych k, a ostateczny wynik uzyskujemy jako liczbę spójnych składowych grafu G'_0.

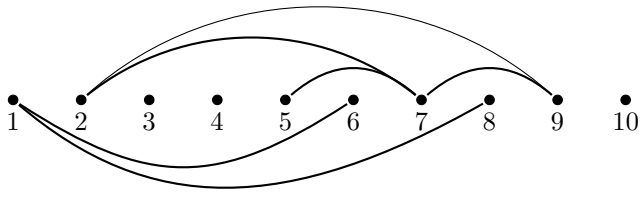
Z powyższej konstrukcji wynika, że w każdym zbiorze W_k jest co najwyżej $n+m-1$ wymagań, z czego co najwyżej $n-1$ wymagań o długości 2^{k+1} oraz co najwyżej m wymagań krótszych. Stąd rozmiar każdego z grafów G'_k to $O(n+m)$. Całe rozwiązanie działa więc w czasie $O((n+m)\log n)$ i w pamięci $O(n+m)$ (nie musimy przechowywać grafów i zapytań dla wcześniej rozważonych k).

Przykład 1. Rozważmy $n = 10$ i zbiór wymagań z pierwszego przykładu z treści zadania z dodanym wymaganiem $(1, 8, 3)$ (które nie zmienia ostatecznego wyniku).

- $k = 3$, $W_3 = \{(1,6,3), (5,7,4), (3,8,1), (1,8,3)\}$.
  Brak wymagań o długości co najmniej 8.
- $k = 2$, $W_2 = \{(1,6,3), (5,7,4), (3,8,1), (1,8,3)\}$.
  Jest tylko jedno wymaganie o długości co najmniej 4. Redukcja za pomocą grafu G'_2 nie przynosi żadnych zmian.
- $k = 1$, $W_1 = \{(1,6,3), (5,7,4), (3,8,1), (1,8,3)\}$.
  Rozbijamy wymagania o długości co najmniej 2:

  $$(1,6,3) \rightarrow (1,6,2),\ (2,7,2), \quad (5,7,4) \rightarrow (5,7,2),\ (7,9,2),$$
  $$(1,8,3) \rightarrow (1,8,2),\ (2,9,2).$$

  Konstruujemy graf G'_1. W lesie rozpinającym nie znajdzie się krawędź $(2, 9)$.

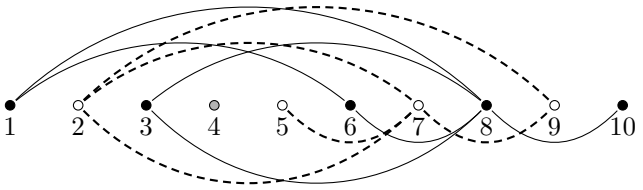


Rozmiar zbioru wymagań wzrasta, ale nie może przekroczyć $n + m - 1$.

- $k = 0$, $W_0 = \{(1,6,2),(1,8,2),(2,7,2),(5,7,2),(7,9,2),(3,8,1)\}$.
  Rozbijamy wymagania ze zbioru W_0 na wymagania o długości 1:

$$(1,6,2) \to (1,6,1),\ (2,7,1), \quad (1,8,2) \to (1,8,1),\ (2,9,1),$$
$$(2,7,2) \to (2,7,1),\ (3,8,1), \quad (5,7,2) \to (5,7,1),\ (6,8,1),$$
$$(7,9,2) \to (7,9,1),\ (8,10,1).$$

  Konstruujemy graf G_0 i dzielimy go na spójne składowe:



Posłowie: Układ równań na słowie

Aby osadzić to zadanie w szerszym kontekście, warto pokusić się o jego interpretację w dziedzinie algorytmów tekstowych. Wówczas jego treść można sformułować równoważnie tak: o pewnym nieznanym słowie (tj. ciągu symboli) długości n wiemy, że określone pary podsłów tego słowa są równe. Chcemy odtworzyć szukane słowo, a jeśli jest wiele możliwości, wyznaczyć taką, w której występuje najwięcej różnych symboli.

Zorientowanemu Czytelnikowi taka interpretacja pozwoli zauważyć pewne związki między tym zadaniem (i jego pierwszym rozwiązaniem) a zadaniem *Równanie na słowach* z V Olimpiady Informatycznej [1]. To jednak nie wszystko. Przykładowo, rozważmy inny problem, w którym mamy odtworzyć nieznane słowo, znając jedynie zbiór długości jego prefikso-sufiksów[2]. Aby go rozwiązać, możemy zastosować nasz algorytm, zadając mu na wejściu wymagania odpowiadające poszczególnym prefikso-sufiksom. Algorytm skonstruuje słowo, które ma te wszystkie długości prefikso-sufiksów. Na koniec musimy jeszcze sprawdzić, czy skonstruowane słowo nie zawiera innej długości prefikso-sufiksów. To jednak możemy łatwo uczynić w czasie $O(n)$, wyznaczając dla tego słowa funkcję prefiksową P i obliczając $P[n], P[P[n]], \ldots$ (patrz książka [6]). Jeśli słowo okaże się nie mieć żadnych innych prefikso-sufiksów, to mamy wynik. W przeciwnym razie zaś możemy od razu stwierdzić, że słowo o żądanym zbiorze prefikso-sufiksów *nie istnieje*, gdyż nasz algorytm konstruuje „najbardziej różnorodne" słowo spełniające podany zbiór równości (czyli jeśli jakieś dwie litery słowa spełniającego zestaw równości mogą być różne, to nasz algorytm rzeczywiście wykorzysta w tym miejscu różne litery), więc jeśli jakieś dwa jego podsłowa są równe, to muszą być także równe w dowolnym słowie spełniającym zadane wymagania.

Podobnie możemy zastosować nasz algorytm do odtworzenia słowa (nad największym alfabetem symboli) o zadanej funkcji prefiksowej, funkcji PREF czy np. o zadanych długościach maksymalnych palindromów. Więcej zastosowań tego algorytmu oraz jego sprytniejszą wersję działającą w czasie $O(n+m)$ można znaleźć w pracy [24].

[2] *Prefikso-sufiksem* słowa nazywamy początkowy fragment słowa równy jego końcowemu fragmentowi tej samej długości.

Marek Cygan
Treść zadania, Opracowanie

Bartosz Kostka
Program

Dostępna pamięć: 512 MB. *OI, etap II, dzień pierwszy, 10.02.2016*

Drogi zmiennokierunkowe

Bajtazar zastanawia się nad przeprowadzką do Bajtowa i chce wynająć tam mieszkanie. Bajtów jest pięknym miastem o licznych zaletach, choć niestety nie należy do nich komunikacja. W mieście jest n skrzyżowań połączonych mniej lub bardziej chaotyczną siecią m dróg. Drogi są bardzo wąskie, więc z przyczyn obiektywnych wszystkie są jednokierunkowe. Jakiś czas temu miejscy specjaliści od komunikacji wpadli na pomysłowe rozwiązanie, które bez konieczności poszerzania dróg umożliwia poruszanie się po nich w różnych kierunkach. A mianowicie, codziennie na wszystkich ulicach zmienia się kierunek poruszania. Innymi słowy, w dni nieparzyste ruch odbywa się zgodnie z oryginalnym skierowaniem ulic, natomiast w dni parzyste ruch na wszystkich ulicach odbywa się w przeciwnych kierunkach.

Bajtazar chce wynająć mieszkanie w takim miejscu, z którego będzie mógł wszędzie łatwo dojechać. Konkretnie, interesuje go mieszkanie przy takim skrzyżowaniu, z którego da się dojechać do każdego innego skrzyżowania **w ciągu jednego dnia** *– w przypadku niektórych skrzyżowań może to być tylko nieparzysty dzień, a w przypadku innych tylko parzysty. Drogą powrotną nie trzeba się przejmować, Bajtazar może wrócić do siebie następnego dnia.*

Mając daną sieć drogową w Bajtowie, wyznacz wszystkie skrzyżowania, które spełniają wymagania Bajtazara.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n i m ($n \geqslant 2$, $m \geqslant 1$) oddzielone pojedynczym odstępem, oznaczające liczbę skrzyżowań i liczbę dróg w Bajtowie. Skrzyżowania numerujemy od 1 do n. W kolejnych m wierszach zawarte są opisy dróg: i-ty z tych wierszy zawiera dwie liczby całkowite a_i i b_i ($1 \leqslant a_i, b_i \leqslant n$, $a_i \neq b_i$) oddzielone pojedynczym odstępem, oznaczające, że istnieje jednokierunkowa droga ze skrzyżowania o numerze a_i do skrzyżowania o numerze b_i (tzn. w dni nieparzyste można przejechać tą drogą z a_i do b_i, natomiast w dni parzyste można nią przejechać z b_i do a_i). Każda uporządkowana para (a_i, b_i) wystąpi na wejściu co najwyżej raz.

Wyjście

W pierwszym wierszu standardowego wyjścia należy zapisać jedną liczbę całkowitą k oznaczającą liczbę skrzyżowań spełniających wymagania Bajtazara. W drugim wierszu należy zapisać rosnący ciąg k liczb pooddzielanych pojedynczymi odstępami, oznaczających numery tych skrzyżowań. Jeśli $k = 0$, drugi wiersz powinien pozostać pusty (tj. program może wypisać pusty wiersz albo po prostu go nie wypisywać).

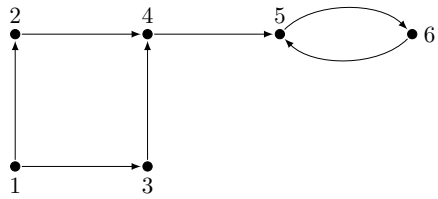
Przykład

Dla danych wejściowych:

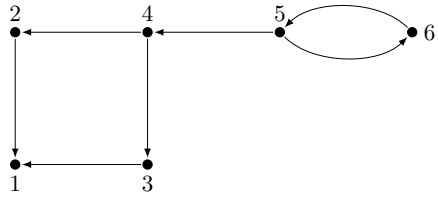
```
6 7
1 2
1 3
2 4
3 4
4 5
5 6
6 5
```

poprawnym wynikiem jest:

```
4
1 4 5 6
```



Sieć dróg w dni nieparzyste.



Sieć dróg w dni parzyste.

Wyjaśnienie do przykładu: *Ze skrzyżowania numer 1 można dojechać do wszystkich innych skrzyżowań w dni nieparzyste. Ze skrzyżowań numer 5 i 6 można dojechać do wszystkich innych skrzyżowań w dni parzyste. Ze skrzyżowania numer 4 do skrzyżowań numer 5 i 6 można dojechać w dni nieparzyste, a do skrzyżowań numer 1, 2 i 3 – w dni parzyste.*

Testy „ocen":

`1ocen`: $n = 10$, $m = 9$; *„ścieżka", w której co druga droga jest skierowana w lewo, a co druga w prawo. Żadne skrzyżowanie nie spełnia wymagań Bajtazara.*

`2ocen`: $n = 100\,000$, $m = 100\,000$, *w dni nieparzyste można przejechać bezpośrednią drogą ze skrzyżowania numer 1 do każdego innego skrzyżowania. Dodatkowo, w dni nieparzyste można przejechać bezpośrednią drogą ze skrzyżowania numer n do skrzyżowania numer 1. Tylko skrzyżowania numer 1 i n spełniają wymagania Bajtazara.*

`3ocen`: $n = 500\,000$, $m = 499\,999$, *„ścieżka"; wszystkie skrzyżowania spełniają wymagania Bajtazara.*

Ocenianie

Zestaw testów dzieli się na podzadania spełniające poniższe warunki. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n, m \leqslant 5000$	*28*
2	$n \leqslant 300\,000$, $m \leqslant 1\,000\,000$; *ze wszystkich skrzyżowań spełniających wymagania Bajtazara można dojechać do każdego innego w dzień nieparzysty*	*29*
3	$n \leqslant 500\,000$, $m \leqslant 1\,000\,000$	*43*

Rozwiązanie

Sieć dróg w Bajtowie możemy przedstawić jako graf skierowany G. Każdy wierzchołek odpowiada skrzyżowaniu, natomiast skierowana krawędź reprezentuje drogę jednokierunkową. Skrzyżowanie x znajduje się w kręgu zainteresowań Bajtazara, jeśli dla każdego innego skrzyżowania y w ciągu jednego dnia da się dojechać ze skrzyżowania x do skrzyżowania y. Innymi słowy, skoro w kolejnych dniach drogi (krawędzie grafu) zmieniają swoją orientację, dla każdego skrzyżowania y musi być spełniony *co najmniej* jeden z warunków:

- w grafie G istnieje ścieżka z wierzchołka x do wierzchołka y,
- w grafie G istnieje ścieżka z wierzchołka y do wierzchołka x.

W problemach dotyczących ścieżek w grafach skierowanych bardzo często przydaje się pojęcie silnie spójnych składowych. Podobnie jest w przypadku naszego zadania. Przypomnijmy, że dwa wierzchołki x i y znajdują się w jednej silnie spójnej składowej wtedy i tylko wtedy, gdy istnieje zarówno ścieżka z wierzchołka x do wierzchołka y, jak i ścieżka z wierzchołka y do wierzchołka x.

Zauważmy, że w przypadku pojedynczej silnie spójnej składowej albo wszystkie jej wierzchołki spełniają wymagania postawione przez Bajtazara, albo też żaden wierzchołek ich nie spełnia. Możemy zatem ściągnąć każdą silnie spójną składową podanego grafu do wierzchołka, tym samym otrzymując *graf silnie spójnych składowych*. Ten krok możemy wykonać w czasie liniowym od rozmiaru grafu za pomocą standardowych metod (więcej o silnie spójnych składowych można znaleźć np. w książkach [4, 6]).

Graf silnie spójnych składowych z natury jest acykliczny, dlatego dla uproszczenia w dalszych rozważaniach będziemy zakładać, że rozważany graf nie ma cykli. Przyjmijmy dodatkowo, że wierzchołki grafu są podane w porządku topologicznym $(v_1, \ldots, v_n)$, tj. każda krawędź prowadzi od wierzchołka o mniejszym indeksie do wierzchołka o większym indeksie. Wygodnie będzie nam wyobrazić sobie, że wierzchołki są uszeregowane na prostej, a krawędzie skierowane są wyłącznie w prawą stronę (rys. 1). Na tej podstawie możemy wywnioskować, że aby wierzchołek v_i spełniał wymagania Bajtazara, muszą być spełnione dwa następujące warunki:

1. dla każdego $j < i$ z wierzchołka v_j istnieje ścieżka do wierzchołka v_i,
2. dla każdego $j > i$ z wierzchołka v_i istnieje ścieżka do wierzchołka v_j.

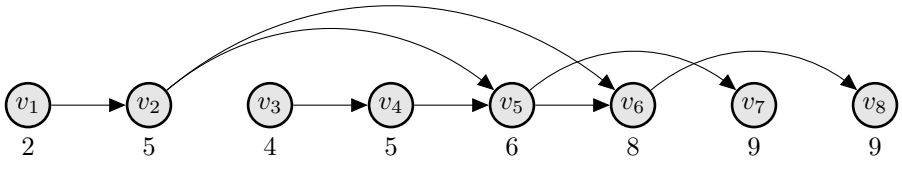
Ze względu na symetrię, jeśli będziemy umieli wyznaczyć wszystkie wierzchołki spełniające pierwszy warunek, to będziemy także umieli wyznaczyć wszystkie wierzchołki spełniające drugi warunek, gdyż drugi warunek jest równoważny pierwszemu w odwrotnie zorientowanym grafie. Aby rozwiązać zadanie wystarczy zatem wyznaczyć wszystkie wierzchołki v_i, do których da się dojść z każdego wierzchołka na lewo od v_i (tj. z każdego wierzchołka o mniejszym indeksie). Wierzchołki takie będziemy nazywać wierzchołkami *dobrymi*, natomiast pozostałe – wierzchołkami *złymi*.

Rozważmy wierzchołek v_i. Jeśli z wierzchołka v_i nie wychodzi krawędź prowadząca do wierzchołka v_{i+1}, to z całą pewnością wierzchołek v_{i+1} jest zły. Aby uogólnić

tę prostą obserwację zdefiniujmy funkcję $r(i)$, której wartość to najmniejszy indeks wierzchołka, do którego prowadzi bezpośrednia krawędź z v_i. Formalnie,

$$r(i) = \min_{(v_i,v_j)\in A} j\,,$$

gdzie A to zbiór krawędzi naszego skierowanego grafu acyklicznego. W przypadku, gdy z wierzchołka v_i nie wychodzi żadna krawędź, ustalamy $r(i) = n + 1$.



Rys. 1: Ilustracja wartości $r(i)$.

Ustalmy pewien wierzchołek v_j. Jeśli istnieje wierzchołek v_i, dla którego $i < j < r(i)$, to wierzchołek v_j jest zły – nie jest on osiągalny z wierzchołka v_i, gdyż wszystkie krawędzie wychodzące z v_i „przeskakują" v_j. Okazuje się, że ten warunek pozwala wykryć *wszystkie* złe wierzchołki.

Lemat 1. Jeśli wierzchołek v_j jest zły, to istnieje wierzchołek v_i, taki że $i < j < r(i)$.

Dowód: Skoro wierzchołek v_j jest zły, istnieje pewien wierzchołek v_a, taki że $a < j$ i z v_a nie da się dojść do v_j. Naszym celem jest znalezienie wierzchołka v_i spełniającego $i < j$ oraz $r(i) > j$. Wykonajmy następujący spacer w grafie, rozpoczynając w wierzchołku v_a. Dopóki znajdujemy się w wierzchołku na lewo od v_j, wykonujemy przejście z aktualnego wierzchołka v_a do $v_{r(a)}$, czyli do najbliższego wierzchołka, do którego możemy wykonać ruch. Spacer może zakończyć się z jednego z dwóch powodów: (i) z pewnego wierzchołka v_i na lewo od v_j przeszliśmy na prawo od v_j, czyli $i < j < r(i)$, lub też (ii) z pewnego wierzchołka v_i, gdzie $i < j$, nie wychodzi żadna krawędź, co oznacza, że $r(i) = n + 1$. W obydwóch przypadkach otrzymujemy $i < j < r(i)$. ■

Powyższy lemat oraz poprzedzająca go obserwacja prowadzą nas do rozwiązania.

Na początku obliczamy graf silnie spójnych składowych i sortujemy jego wierzchołki w porządku topologicznym. Obydwa te kroki możemy wykonać w czasie $O(n+m)$. Następnie w czasie $O(n+m)$ obliczamy wartości $r(i)$, po czym znajdujemy wierzchołki, których indeksy należą do sumy przedziałów $(i, r(i))$. Ten ostatni krok zrealizować możemy w czasie $O(n)$, przeglądając przedziały dla rosnących wartości i i pamiętając największą dotychczas napotkaną wartość $r(i)$. Tym samym znajdziemy wszystkie złe wierzchołki w grafie silnie spójnych składowych, co pozwoli nam zidentyfikować złe wierzchołki w oryginalnym grafie. Jeśli powtórzymy wszystkie kroki dla grafu z odwróconą orientacją krawędzi, otrzymamy rozwiązanie problemu Bajtazara.

Tomasz Syposz	Jakub Radoszewski	Aleksander Łukasiewicz
Treść zadania	Opracowanie	Program

Dostępna pamięć: 32 MB. *OI, etap II, dzień pierwszy, 10.02.2016*

Zająknięcia

Bitek zapadł ostatnio na dziwną chorobę: strasznie się jąka, a przy tym jedyne słowa, które wypowiada, to liczby. Jego starszy brat, Bajtek, zauważył jednak dziwną powtarzalność w zająknięciach Bitka. Podejrzewa, że Bitek tak naprawdę udaje, żeby nie chodzić do szkoły i móc więcej grać na komputerze. Bajtek nie może przez to uczyć się programowania i jest z tego powodu bardzo smutny. Postanowił więc zdemaskować młodszego brata i liczy, że w nagrodę będzie miał tyle czasu na programowanie, ile dusza zapragnie.

Opiszmy formalnie podejrzenia Bajtka. Załóżmy, że mamy dany ciąg liczb A.

- **Podciągiem** *A nazywamy ciąg powstały przez wyrzucenie z A dowolnych wyrazów, np. $1, 1, 7, 5$ jest podciągiem ciągu $1, 3, 1, 7, 6, 6, 5, 5$.*
- **Zająknięciem** *A nazywamy podciąg A, który składa się z ustawionych po kolei par takich samych wyrazów, np. $1, 1, 1, 1, 3, 3$ jest zająknięciem ciągu $1, 2, 1, 2, 1, 2, 1, 3, 3$.*

Mając dane dwie wypowiedzi Bitka jako ciągi liczb, pomóż Bajtkowi stwierdzić, jaka jest długość najdłuższego zająknięcia, które występuje w każdym z tych ciągów, a nagroda Cię nie ominie.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n oraz m ($n, m \geqslant 2$) oddzielone pojedynczym odstępem, oznaczające długości ciągów A i B, które reprezentują wypowiedzi Bitka. W drugim wierszu wejścia znajduje się n liczb całkowitych $a_1, a_2, \ldots, a_n$ oddzielonych pojedynczymi odstępami, czyli kolejne wyrazy ciągu A ($1 \leqslant a_i \leqslant 10^9$). W trzecim wierszu wejścia znajduje się m liczb całkowitych $b_1, b_2, \ldots, b_m$ oddzielonych pojedynczymi odstępami, czyli kolejne wyrazy ciągu B ($1 \leqslant b_i \leqslant 10^9$).

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną nieujemną liczbę całkowitą oznaczającą długość najdłuższego wspólnego zająknięcia ciągów A i B. Jeśli ciągi nie mają żadnego wspólnego zająknięcia, poprawnym wynikiem jest 0.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`7 9`
`1 2 2 3 1 1 1`
`2 4 2 3 1 2 4 1 1`
	`4`

Wyjaśnienie do przykładu: *Szukanym ciągiem jest 2, 2, 1, 1.*

Testy „ocen”:

`1ocen`: *$n = 5$, $m = 4$, wszystkie liczby to 42,*

`2ocen`: *$n = 9$, $m = 13$, ciągi to słowa* `OLIMPIADA` *i* `INFORMATYCZNA` *zapisane w kodzie ASCII,*

`3ocen`: *$n = 15\,000$, $m = 15\,000$, ciąg A składa się z par rosnących liczb $(1, 1, 2, 2, 3, 3, \ldots, 7500, 7500)$, natomiast B powstał w wyniku odwrócenia A,*

`4ocen`: *$n = 10\,000$, $m = 5000$, oba ciągi składają się z par naprzemiennych liczb 13 oraz 37 $(13, 37, 13, 37, \ldots)$.*

Ocenianie

Zestaw testów dzieli się na podzadania spełniające poniższe warunki. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n, m \leqslant 2000$	*30*
2	$n, m \leqslant 15\,000$ *i każda liczba w każdym ciągu występuje co najwyżej dwa razy*	*28*
3	$n, m \leqslant 15\,000$	*42*

Rozwiązanie

W zadaniu dane są dwa ciągi liczb $A = (a_1, \ldots, a_n)$ i $B = (b_1, \ldots, b_m)$. W opisie rozwiązania zamiast o wartościach elementów ciągów wygodniej nam będzie mówić o ich kolorach, tak więc np. a_i oznaczać będzie kolor i-tego elementu ciągu A. Naszym celem jest znaleźć najdłuższe wspólne zająknięcie ciągów A i B, czyli najdłuższy ciąg kolorów złożony z parami powtarzających się elementów, który jest podciągiem każdego z ciągów A i B. Co istotne, w odpowiedzi wystarczy podać długość najdłuższego wspólnego zająknięcia.

Najdłuższy wspólny podciąg

Nasze zadanie ewidentnie ma związek z problemem znajdowania najdłuższego wspólnego podciągu dwóch ciągów. Rozważania zacznijmy więc od przypomnienia klasycznego rozwiązania tego problemu za pomocą programowania dynamicznego (patrz np. książka [6]). Wyznacza się w nim dwuwymiarową tablicę NWP rozmiaru $(n+1)\times(m+1)$, taką że $NWP[i, j]$ oznacza długość najdłuższego wspólnego podciągu ciągów $a_1, \ldots, a_i$ oraz $b_1, \ldots, b_j$. Szukanym wynikiem jest oczywiście $NWP[n, m]$. Mamy następującą zależność rekurencyjną:

$$NWP[i,j] = \begin{cases} 0 & \text{jeżeli } i = 0 \text{ lub } j = 0 \\ NWP[i-1, j-1] + 1 & \text{jeżeli } a_i = b_j \\ \max(NWP[i-1, j], NWP[i, j-1]) & \text{w przeciwnym przypadku.} \end{cases}$$

Z zależności tej wynika następujący algorytm o złożoności $O(nm)$, w którym wypełniamy kolejne pola tablicy $NWP[i,j]$ dla $i = 0,\ldots,n$ oraz $j = 0,\ldots,m$.

```
1:  procedure ObliczNWP
2:  begin
3:    for j := 0 to m do NWP[0,j] := 0;
4:    for i := 1 to n do begin
5:      NWP[i,0] := 0;
6:      for j := 1 to m do
7:        if a[i] = b[j] then
8:          NWP[i,j] := NWP[i-1,j-1] + 1
9:        else
10:         NWP[i,j] := max(NWP[i-1,j], NWP[i,j-1]);
11:   end
12:   return NWP[n,m];
13: end
```

Warto dodać, że choć złożoność pamięciowa powyższego algorytmu to także $O(nm)$, to można ją łatwo zredukować do $O(n+m)$. Wystarczy mianowicie pamiętać tylko dwa ostatnie wiersze tablicy, tj. $NWP[i-1,\star]$ oraz $NWP[i,\star]$. W tym celu można po prostu we wszystkich odwołaniach do pól tablicy NWP w powyższym pseudokodzie na pierwszej współrzędnej brać resztę z dzielenia przez 2.

Pierwsze rozwiązanie

Nasze pierwsze rozwiązanie będzie naśladować opisaną wyżej metodę. Niech $NWZ[i,j]$ oznacza długość najdłuższego wspólnego zająknięcia ciągów $a_1,\ldots,a_i$ oraz $b_1,\ldots,b_j$.

Jeśli do wyznaczenia tablicy $NWZ[i,j]$ chcielibyśmy zastosować metodę z powyższego pseudokodu, to przypadki brzegowe oraz przypadek $a_i \neq b_j$ pozostaną bez zmian. Natomiast w sytuacji, gdy $a_i = b_j$, powinniśmy do zająknięcia dołożyć jeszcze jeden element koloru a_i. Odpowiada to wybraniu pary elementów: $a_{i'} = a_i$ dla $i' < i$ oraz $b_{j'} = b_j$ dla $j' < j$, co zwiększa długość zająknięcia o dwa elementy. Może się też okazać, że w tym przypadku któryś z szukanych elementów $a_{i'}$ oraz $b_{j'}$ nie istnieje lub znajduje się bardzo wcześnie w ciągu; wówczas lepiej jest wybrać, podobnie jak w przypadku $a_i \neq b_j$, większą z wartości NWZ dla krótszych fragmentów ciągów.

Sprecyzujmy, że jako indeks i' – jeśli istnieje – najlepiej wybrać indeks wskazujący najbliższy wcześniejszy element o tym samym kolorze co a_i. Rzeczywiście, gdyby w najdłuższym wspólnym zająknięciu w ciągu A występowała jako para kolejnych jednokolorowych elementów para $a_{i'}$ oraz a_i, a istniałby indeks i'' taki że $i' < i'' < i$ oraz $a_{i''} = a_i$, to moglibyśmy równie dobrze zamiast $a_{i'}$ wziąć do zająknięcia element $a_{i''}$. Podobnie rzecz ma się w przypadku ciągu B.

Dla danego indeksu $i \in \{1,\ldots,n\}$ przez $prev_A[i]$ oznaczmy indeks najbliższego wcześniejszego elementu o kolorze a_i w ciągu A. Jeśli element $prev_A[i]$ nie istnieje, przyjmujemy, że $prev_A[i] = 0$. Wprowadźmy też analogiczne oznaczenie $prev_B[j]$ dla elementu b_j w ciągu B. Pozwala nam to zapisać następujący pseudokod wyznaczania tablicy $NWZ[i,j]$.

```
 1: procedure ObliczNWZ
 2: begin
 3:   for j := 0 to m do NWZ[0, j] := 0;
 4:   for i := 1 to n do begin
 5:     NWZ[i, 0] := 0;
 6:     for j := 1 to m do begin
 7:       if a[i] = b[j] and prev_A[i] > 0 and prev_B[j] > 0 then
 8:         NWZ[i, j] := NWZ[prev_A[i] − 1, prev_B[j] − 1] + 2
 9:       else
10:         NWZ[i, j] := 0;
11:       NWZ[i, j] := max(NWZ[i, j], NWZ[i − 1, j], NWZ[i, j − 1]);
12:     end
13:   end
14:   return NWZ[n, m];
15: end
```

Algorytm ten ma złożoność czasową i pamięciową $O(nm)$, pod warunkiem, że będziemy mieli do dyspozycji tablice $prev_A[i]$ oraz $prev_B[j]$. Tablice te można wyznaczyć zupełnie siłowo w czasie $O(n^2 + m^2)$, co było wystarczające w tym zadaniu.

Całe rozwiązanie ma zatem złożoność czasową $O((n + m)^2)$ i pamięciową $O(nm)$. Przykładowe implementacje można znaleźć w plikach `zajb2.cpp`, `zajb3.cpp` i `zajb7.pas`. Tego typu rozwiązania przechodziły pierwsze podzadanie, natomiast w pozostałych podzadaniach przekraczały limit pamięciowy. Rzeczywiście, dla maksymalnych wartości n i m z zadania (tj. 15 000) tablica NWZ musiałaby mieć $(15\,000)^2 = 225\,000\,000$ komórek, co nie ma możliwości zmieścić się w 32 MB pamięci.

Dodajmy jeszcze, że implementację rozwiązania o złożoności czasowej $O(n^2m^2)$, które nie zapamiętuje wartości tablic $prev_A[i]$ oraz $prev_B[j]$, tylko każdorazowo sprawdza wszystkich kandydatów na elementy $a_{i'}$ i $b_{j'}$, można znaleźć w pliku `zajb4.cpp`.

Rozwiązanie wzorcowe

W naszym zadaniu nie jest niestety tak łatwo zmniejszyć złożoność pamięciową rozwiązania jak w przypadku problemu najdłuższego wspólnego podciągu. Moglibyśmy zastosować sztuczkę z traktowaniem pierwszej współrzędnej modulo 2, gdyby nie konieczność odwoływania się do wartości $NWZ[prev_A[i] - 1, prev_B[j] - 1]$, która teoretycznie może znajdować się w zupełnie dowolnym miejscu tablicy NWZ.

Przyjrzyjmy się jednak dokładniej, które komórki tablicy występują w tych kłopotliwych odwołaniach w poszczególnych momentach obliczeń. Gdy w zewnętrznej pętli rozpatrujemy konkretny indeks i, wartość $prev_A[i]$ jest oczywiście ustalona i wskazuje na wcześniejszy element tego samego koloru co i; niech będzie to kolor c. W tym obrocie pętli interesują nas tylko indeksy $prev_B[j]$ dla j takich, że $b_j = c$. Elementy o tych indeksach w ciągu B mają także kolor c.

Gdy w algorytmie przechodzimy do kolejnych indeksów i, takich że $a_i \neq c$, to interesujące nas indeksy $prev_B[j]$ są zatem zupełnie inne. Natomiast kiedy napotkamy pierwszy indeks $i' > i$, taki że $a_{i'} = c$, będziemy mieli $prev_A[i'] = i$, a interesujące nas wartości $prev_B[j]$ będą znów odpowiadały elementom koloru c.

Wprowadźmy do rozwiązania pomocniczą tablicę jednowymiarową *memo* indeksowaną parametrem j. Zauważmy, że gdybyśmy podczas rozpatrywania indeksu i zapamiętali, jako $memo[j]$, wartości $NWZ[i-1, j-1]$ dla wszystkich indeksów j takich że $b_j = c$, to wówczas, rozpatrując indeks i', moglibyśmy jako $NWZ[prev_A[i']-1, prev_B[j]-1]$ wziąć dokładnie wartość $memo[prev_B[j]]$. Mamy wówczas gwarancję, że obliczenia dla indeksów pomiędzy i a i' nie nadpiszą pól $memo[j]$ dla indeksów j, na których w ciągu B znajdują się elementy koloru c.

Ostatecznie musimy wprowadzić w pseudokodzie stosunkowo niewielkie zmiany.

```
 1: procedure ObliczNWZ2
 2: begin
 3:   for j := 0 to m do NWZ[0, j] := memo[j] := 0;
 4:   for i := 1 to n do begin
 5:     NWZ[i mod 2, 0] := 0;
 6:     for j := 1 to m do begin
 7:       if a[i] = b[j] and prev_A[i] > 0 and prev_B[j] > 0 then
 8:         NWZ[i mod 2, j] := memo[prev_B[j]] + 2
 9:       else
10:         NWZ[i mod 2, j] := 0;
11:       NWZ[i mod 2, j] := max(NWZ[i mod 2, j], NWZ[(i - 1) mod 2, j],
12:                              NWZ[i mod 2, j - 1]);
13:     end
14:     for j := 1 to m do
15:       if a[i] = b[j] then
16:         memo[j] := NWZ[(i - 1) mod 2, j - 1];
17:   end
18:   return NWZ[n mod 2, m];
19: end
```

Otrzymane rozwiązanie ma ewidentnie złożoność pamięciową $O(n + m)$, a jego złożoność czasowa nie uległa zmianie. Implementację tego typu rozwiązania można znaleźć w plikach `zaj.cpp`, `zaj3.pas`, `zaj4.cpp` i `zaj6.cpp`.

Dodatkowe optymalizacje

W naszych rozwiązaniach tablice $prev_A$ i $prev_B$ wyznaczaliśmy, odpowiednio, w czasie $O(n^2)$ i $O(m^2)$. Programujący w języku C++ mogli obliczyć je efektywniej np. z użyciem kontenera `map`. Faktycznie, przeglądając ciąg A za pomocą indeksu i, dla każdego koloru wystarczy pamiętać w strukturze danych ostatnio napotkany indeks elementu tego koloru. Wówczas $prev_A[i]$ wyznaczamy jako indeks zapamiętany w strukturze danych pod kolorem a_i, a odtąd w strukturze pamiętamy dla tego koloru indeks i. Złożoność pojedynczej operacji na kontenerze `map` to $O(\log n)$, więc cały proces zajmuje czas $O(n \log n)$. Tak samo w czasie $O(m \log m)$ można wyznaczyć elementy tablicy $prev_B[j]$. W identycznej złożoności tablice te można wypełnić także bez użycia wspomnianego kontenera – w przypadku tablicy $prev_A[i]$ wystarczy przejrzeć wszystkie pary postaci (a_i, i), posortowawszy je niemalejąco po współrzędnych.

Warto też wspomnieć o pewnym prostym usprawnieniu, które można było zastosować na początku każdego z omawianych rozwiązań. Otóż jeśli elementy jakiegoś koloru występują w którymś z ciągów A, B mniej niż dwukrotnie, to możemy usunąć wszystkie elementy tego koloru z obydwu ciągów. Optymalizację tę łatwo przeprowadzić w czasie $O((n+m)\log(n+m))$. Choć nie zmniejsza ona pesymistycznej złożoności czasowej rozwiązania, w przypadku wielu typów testów pozwala istotnie zmniejszyć długość ciągów.

Rozwiązanie drugiego podzadania

W drugim podzadaniu mieliśmy gwarancję, że w obu ciągach każdy z kolorów występuje co najwyżej dwukrotnie. Przy tym założeniu zadanie można rozwiązać w inny sposób, stosując metodę programowania dynamicznego z liniową liczbą stanów.

Dla każdego koloru elementu, który w każdym z ciągów występuje dwukrotnie (pozostałe kolory możemy w ogóle odrzucić na podstawie opisanej powyżej optymalizacji), znajdujemy indeks i późniejszego wystąpienia elementu tego koloru w ciągu A i zapamiętujemy dla niego indeks $odp[i]$ późniejszego wystąpienia elementu tego koloru w ciągu B. Pozostałe pola tablicy odp inicjujemy zerami. Dla każdego indeksu i w ciągu A wyznaczymy, w tablicy $NWZ'[i]$, długość najdłuższego wspólnego zająknięcia ciągów A i B, które kończy się w ciągu A elementem a_i. Widzimy, że $NWZ'[i] > 0$ tylko dla indeksów i takich że $odp[i] > 0$.

Aby obliczyć $NWZ'[i]$, wystarczy przejrzeć wszystkie wcześniejsze pozycje j i sprawdzić, czy kończące się na nich najdłuższe wspólne zająknięcia (jeśli istnieją) można przedłużyć o elementy znajdujące się pod indeksami $prev_A[i]$, i w ciągu A oraz te pod indeksami $prev_B[odp[i]]$, $odp[i]$ w ciągu B. W ten sposób uzyskujemy poniższy pseudokod.

```
1:  procedure ObliczNWZ'
2:  begin
3:    for i := 0 to n do begin
4:      NWZ'[i] := 0;
5:      if odp[i] > 0 then
6:        for j := 0 to prev_A[i] - 1 do
7:          if odp[j] < prev_B[odp[i]] then
8:            NWZ'[i] := max(NWZ'[i], NWZ'[j] + 2);
9:    end
10:   return max{NWZ'[1], ..., NWZ'[n]};
11: end
```

Indeksy $odp[i]$ można wyznaczyć siłowo; nie będziemy się na ten temat szczegółowo rozpisywać. Otrzymane rozwiązanie ma złożoność czasową $O((n+m)^2)$ i pamięciową $O(n+m)$. Przykładową implementację można znaleźć w pliku `zajb1.cpp`.

Dodajmy na koniec, że podzadanie 2 można także rozwiązać efektywniej, bo w czasie $O((n+m)\log(n+m))$. Rozwiązanie to wykorzystuje drzewo przedziałowe i jest podobne do rozwiązania problemu najdłuższego wspólnego podciągu w przypadku, gdy każdy kolor występuje w każdym z ciągów co najwyżej raz. Dopracowanie jego szczegółów pozostawiamy Czytelnikowi.