Zawody I stopnia

opracowania zadań

Wojciech Guzicki — autor zadania, opracowanie
Tomasz Waleń — program wzorcowy

Gdzie zbudować browar?

Mieszkańcy bajtockiej wyspy Abstynencja bardzo lubią piwo bezalkoholowe. Do tej pory piwo bezalkoholowe sprowadzano z Polski, ale w tym roku w jednym z miast na Abstynencji zostanie wybudowany browar. Wszystkie miasta wyspy leżą na wybrzeżu i są połączone autostradą obiegającą wyspę wzdłuż brzegu. Inwestor budujący browar zebrał informacje o zapotrzebowaniu na piwo, tj. ile cystern piwa trzeba codziennie dostarczyć do każdego z miast. Dysponuje także zestawieniem odległości pomiędzy miastami. Koszt transportu jednej cysterny na odległość 1 km wynosi 1 talar. Dzienny koszt transportu to suma, jaką trzeba wyłożyć, by do każdego miasta przetransportować z browaru tyle cystern, ile wynosi zapotrzebowanie w danym mieście. Jego wielkość zależy od miejsca położenia browaru. Inwestor chce zminimalizować koszty transportu piwa.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `BRO.IN` *liczbę miast, odległości między nimi oraz dzienne zapotrzebowania na piwo,*
- *obliczy minimalny dzienny koszt transportu piwa,*
- *zapisze wynik w pliku tekstowym* `BRO.OUT`.

Wejście

W pierwszym wierszu pliku tekstowego `BRO.IN` *jest zapisana jedna liczba całkowita n równa liczbie miast, $5 \leqslant n \leqslant 10\,000$. W każdym z kolejnych n wierszy zapisana jest para nieujemnych liczb całkowitych oddzielonych pojedynczym odstępem. Liczby z_i, d_i zapisane w $(i+1)$-ym wierszu, to, odpowiednio, zapotrzebowanie na piwo w mieście nr i oraz odległość (w kilometrach) od miasta nr i do następnego miasta na autostradzie. Kolejne miasta na autostradzie mają kolejne numery, po mieście nr n leży miasto nr 1. Całkowita długość autostrady nie przekracza 1 000 000 km. Zapotrzebowanie na piwo w żadnym mieście nie przekracza 1 000 cystern.*

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku tekstowego `BRO.OUT`*, dokładnie jedną liczbę całkowitą równą minimalnemu dziennemu kosztowi transportu piwa.*

Przykład

Dla pliku wejściowego `BRO.IN`*:*

```
6
1 2
2 3
1 2
5 2
1 10
2 3
```

poprawną odpowiedzią jest plik tekstowy `BRO.OUT`*:*

```
41
```

ROZWIĄZANIE

Oczywiste rozwiązanie polega na obliczeniu kosztu transportu z każdego miasta do każdego innego i dodaniu otrzymanych liczb. Dla każdego miasta należy obliczyć w tym celu $n-1$ odległości, co powoduje, że złożoność tego algorytmu wynosi $O(n^2)$. Istnieje algorytm szybszy, działający w czasie $O(n)$.

Warto zauważyć, że jeśli miasta są połączone siecią dróg będącą drzewem, to właściwa lokalizacja browaru nie zależy od odległości między miastami, ale tylko od zapotrzebowań w miastach. Oczywiście łączny koszt transportu zależy od odległości. Przypadek, gdy graf dróg jest drzewem (zresztą bardzo prostym), był już tematem jednego z zadań olimpijskich (Mudstok Bis, II Olimpiada Informatyczna, zobacz [2]). Jeśli graf dróg zawiera cykle, odpowiedź jest znacznie bardziej skomplikowana. Dla jednego cyklu istnieje algorytm liniowy, przy czym nie tylko łączny koszt, ale i lokalizacja browaru zależy teraz również od odległości.

Trudność polega na tym, że w drzewie istnieje tylko jedna droga prowadząca z danego miasta do każdego innego, natomiast w przypadku miast położonych na okręgu zawsze są dwie drogi: można obiegać okrąg w kierunku zgodnym z ruchem wskazówek zegara lub w kierunku przeciwnym. W przypadku każdej pary miast musimy zdecydować, który kierunek jest bardziej opłacalny. Przyjmijmy dla ustalenia uwagi, że miasta są numerowane w kierunku zgodnym z ruchem wskazówek zegara. Wczytane dane umieszczamy w dwóch tablicach: $z[i]$ jest zapotrzebowaniem na piwo w mieście i, $d[i]$ jest odległością od miasta i do następnego.

W algorytmie optymalnym najpierw obliczamy koszt transportu z miasta o numerze 1 (w programie wzorcowym dla wygody rozpoczęto numerację miast od zera; łatwiej wtedy obliczyć numer następnego i poprzedniego miasta na okręgu za pomocą funkcji **mod**). Jednocześnie obliczamy kilka wielkości pomocniczych:

- indeks l najdalszego miasta, do którego dowozimy piwo kierując się „w lewo”, tzn. przeciwnie do ruchu wskazówek zegara;
- indeks r najdalszego miasta, do którego dowozimy piwo kierując się „w prawo”, tzn. zgodnie z ruchem wskazówek zegara;
- łączne zapotrzebowanie z_l na piwo w miastach, do których je dowozimy jadąc „w lewo”;
- łączne zapotrzebowanie z_r na piwo w miastach, do których je dowozimy jadąc „w prawo”;
- odległość d_l z pierwszego miasta do miasta o numerze l, gdy jedziemy „w lewo”;
- odległość d_r z pierwszego miasta do miasta o numerze r, gdy jedziemy „w prawo”;
- łączny koszt transportu c — ten koszt zapamiętujemy jako dotychczas minimalny.

W kolejnych krokach algorytmu koszt transportu piwa z następnych miast będzie obliczany jako korekta obliczonego poprzednio kosztu. Przypuśćmy, że mamy obliczone te wszystkie dane dla miasta o numerze $i-1$. Wtedy dla miasta o numerze i dokonujemy prostej korekty. Najpierw znajdujemy odległość d między miastami o numerach $i-1$ oraz i. Następnie obliczamy, o ile zmieni się łączny koszt transportu piwa, jeśli będziemy je dowozić z miasta i po **tych samych** drogach: a więc do kosztu dodajemy $(z_l + z[i-1]) \cdot d$ i odejmujemy $z_r \cdot d$. Mianowicie wszystkie cysterny wysyłane w lewo przejadą odległość o d większą i dojdą nowe cysterny, które muszą dojechać do miasta o numerze $i-1$, a wszystkie cysterny jadące w prawo (w tym również te, które jechały do miasta i) przejadą o d mniej. Modyfikujemy również z_l (dodając $z[i-1]$), z_r (odejmując $z[i]$), d_l (dodając d) i d_r (odejmując d). Teraz poprawiamy łączny koszt, zwiększając liczbę cystern wysyłanych w prawo i zmniejszając liczbę cystern wysyłanych w lewo.

Patrzymy na odległość $d[r]$ między miastem r i następnym leżącym na prawo (czyli zgodnie z ruchem wskazówek zegara) miastem l. Jeśli $d_l > d_r + d[r]$, to do miasta o numerze l opłaca się teraz jechać w prawo. A więc dokonujemy kolejnej korekty: zwiększamy d_r o $d[r]$ i z_r o $z[r+1]$, zmniejszamy d_l o $d[l]$ i z_l o $z[l]$, zwiększamy o 1 numery miast r i l (pamiętając, że za $n+1$ bierzemy 1) i wreszcie odpowiednio modyfikujemy łączny koszt transportu. Teraz znów sprawdzamy, czy w podobny sposób nie przesunąć granicy między kierunkami w lewo i w prawo do następnego miasta. Postępujemy tak dotąd, aż dalsze zmiany nie przyniosą korzyści. W ten sposób zakończyliśmy korekty i mamy obliczony łączny koszt transportu piwa z miasta o numerze i. Porównujemy go z dotychczasowym minimalnym zapamiętując, jeśli jest mniejszy, i przechodzimy do kolejnego miasta.

Jaka będzie złożoność obliczeniowa tego algorytmu? Dla każdego miasta i dokonywaliśmy najpierw pewnej ustalonej liczby zmian, a następnie w pętli dokonywaliśmy być może wielu korekt. Zauważmy jednak, że każda z tych korekt dotyczyła zawsze nowej granicy między kierunkiem lewym i prawym, przy czym ta granica przesuwała się zawsze tylko w prawo. Zatem łącznie dokonaliśmy tylko n takich korekt. Stąd wynika, że czas działania tego algorytmu będzie proporcjonalny do n.

Dla liczby $n = 10000$ różnica w czasie działania obu algorytmów (pierwszego o złożoności $O(n^2)$ i drugiego o złożoności $O(n)$) jest znaczna i duże testy pozwalały na odrzucenie programów działających w czasie $\Theta(n^2)$. Łączny koszt transportu mógł też przekroczyć zakres liczb typu *longint* i niektóre testy wykrywały ten błąd.

TESTY

Do sprawdzania rozwiązań zawodników użyto 17 testów. Testy 1 i 2, 3 i 4, 5 i 6, 14 i 15 były grupowane.

- BRO0.IN — przykładowy test z treści zadania,
- BRO1.IN — $n = 10$, mały test poprawnościowy,
- BRO2.IN — $n = 15$, mały test poprawnościowy,
- BRO3.IN — $n = 20$, test sprawdzający błąd przekroczenia zakresu,
- BRO4.IN — $n = 20$, pary mają postać (i,i),
- BRO5.IN — $n = 30$, mały test poprawnościowy,
- BRO6.IN — $n = 40$, mały test poprawnościowy,
- BRO7.IN — $n = 500$, na przemian pary (1,1), (100,1),
- BRO8.IN — $n = 1000$, na przemian pary (1,1), (100,1000),
- BRO9.IN — $n = 3000$, trzy grupy zawierające po 1000 miast oddalone o 300000 km,
- BRO10.IN — $n = 4000$, pary postaci ((i+1001) mod 1000, 100),
- BRO11.IN — $n = 5000$, pary postaci ($3000 - i$ mod 1000, 100+(i mod 100)),
- BRO12.IN — $n = 7000$, pary mają postać ((i+1000) mod 1000, 100),
- BRO13.IN — $n = 7500$, dane losowe,
- BRO14.IN — duży prosty test dla $n = 10000$, wszystkie pary mają postać (1,1),
- BRO15.IN — mały test poprawnościowy dla $n = 20$,
- BRO16.IN — $n = 10000$, dane losowe.

Wojciech Rytter autor zadania, opracowanie

Piotr Sankowski program wzorcowy

Wirusy

Komisja Badania Wirusów Binarnych wykryła, że pewne ciągi zer i jedynek są kodami wirusów. Komisja wyodrębniła zbiór wszystkich kodów wirusów. Ciąg zer i jedynek nazywamy **bezpiecznym**, *gdy żaden jego segment (tj. ciąg kolejnych wyrazów) nie jest kodem wirusa. Komisja dąży do ustalenia, czy istnieje nieskończony, bezpieczny ciąg zer i jedynek.*

Przykład

Dla zbioru kodów $\{011, 11, 00000\}$ *nieskończonym, bezpiecznym ciągiem jest 010101 .... Dla zbioru kodów* $\{01, 11, 00000\}$ *nie istnieje nieskończony, bezpieczny ciąg zer i jedynek.*

Zadanie

Napisz program, który:

- *wczyta kody wirusów z pliku tekstowego* `WIR.IN`,
- *stwierdzi, czy istnieje nieskończony, bezpieczny ciąg zer i jedynek,*
- *zapisze wynik do pliku tekstowego* `WIR.OUT`.

Wejście

W pierwszym wierszu pliku wejściowego `WIR.IN` *znajduje się jedna liczba całkowita* n *będąca liczbą wszystkich kodów wirusów.*

W każdym z kolejnych n *wierszy znajduje się jedno niepuste słowo złożone ze znaków* `0` *i* `1` *— kod wirusa. Sumaryczna długość wszystkich słów nie przekracza 30000.*

Wyjście

W pierwszym i jedynym wierszu pliku wyjściowego `WIR.OUT` *powinno znajdować się słowo:*

- `TAK` *— jeżeli istnieje nieskończony, bezpieczny ciąg zer i jedynek,*
- `NIE` *— w przeciwnym przypadku.*

Przykład

Dla pliku wejściowego `WIR.IN`*:*

```
3
01
11
00000
```

poprawną odpowiedzią jest plik tekstowy `WIR.OUT`*:*

```
NIE
```

ROZWIĄZANIE

Algorytm teorio-grafowy

W rozwiązaniu zadania najważniejszą rolę odgrywają sufiksy i prefiksy wirusów. Jeśli słowo α można zapisać jako złożenie trzech słów $\alpha = u \cdot v \cdot w$, to u nazywamy prefiksem, a w sufiksem α. Na przykład prefiksami słowa *rytter* są $\{\epsilon, r, ry, ryt, rytt, rytte, rytter\}$, a sufiksami $\{\epsilon, r, er, ter, tter, ytter, rytter\}$, gdzie ϵ oznacza słowo puste o długości zero.

Dla słowa α i zbioru słów-wirusów W, przez $max\_pref(\alpha)$ oznaczmy najdłuższy sufiks α, który jest jednocześnie prefiksem pewnego wirusa z W. Jeśli chcemy sprawdzić, czy dany ciąg binarny nie zawiera wirusa, to wystarczy przeglądać ten ciąg znak po znaku i pamiętać jedynie ostatni **istotny** fragment. Tym istotnym fragmentem jest maksymalny prefiks wirusa, zatem po wczytaniu ciągu $a_1a_2a_3 \dots a_k$ pamiętamy $max\_pref(a_1a_2a_3 \dots a_k)$. Słowo $a_1a_2a_3 \dots a_n$ jest *bezpieczne*, gdy każdy z pośrednich istotnych fragmentów jest bezpieczny, tzn. gdy

$$max\_pref(a_1),\ max\_pref(a_1a_2),\ max\_pref(a_1a_2a_3),\ \dots,\ max\_pref(a_1a_2a_3 \dots a_n)$$

są bezpieczne (nie są wirusami).

Motywuje to wprowadzenie pomocniczego grafu G, którego ścieżki odpowiadają dokładnie wszystkim możliwym bezpiecznym słowom i którego węzłami są wszystkie bezpieczne prefiksy wirusów. Od węzła x prowadzimy skierowaną krawędź do y, gdy dla pewnej litery a (gdzie $a = 0$ lub $a = 1$) po dopisaniu jej do x otrzymamy słowo takie, że maksymalnym jego sufiksem, który jest prefiksem pewnego wirusa, jest y. Oznaczmy y przez $(\sigma(x, a))$. Można to zapisać bardziej formalnie:

- $\sigma(x, a) \ = \ max\_pref(xa)$, gdy xa jest bezpieczny,
- $\sigma(x, a) \ = \ \mathbf{NIL}$, gdy xa nie jest bezpieczny (**NIL** oznacza brak krawędzi).

Rysunek 2 przedstawia graf G dla przykładowego zbioru wirusów

$$W \ = \ \{000,\ 0110,\ 100101, 1010, 111\}.$$

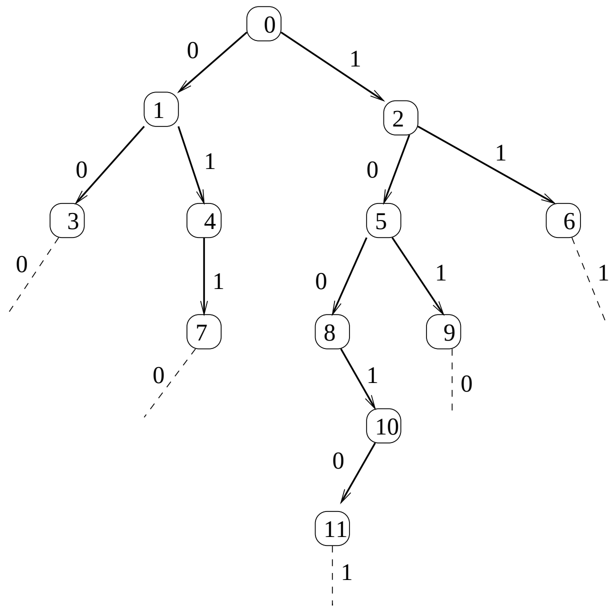
Grafy tego typu odpowiadają w *teorii obliczeń* tzw. *automatom skończonym.* Zbiór W jest bezpieczny wtedy i tylko wtedy, gdy w grafie G istnieje cykl skierowany osiągalny z węzła startowego odpowiadającego słowu pustemu. Każda ścieżka w tym grafie zaczynająca się od węzła startowego wyznacza ciąg bezpieczny.

Konstrukcja drzewa bezpiecznych prefiksów wirusów

Pierwszym krokiem w konstrukcji grafu G jest zbudowanie drzewa właściwych prefiksów wirusów, tzn. takich prefiksów słów ze zbioru W, które nie są równe żadnemu wirusowi. Drzewo dla przykładowego zbioru W jest przedstawione na rysunku 1.

Uwaga: Załóżmy początkowo, że żaden wzorzec nie jest właściwym podsłowem innego wzorca.

Rys. 1 Początkowa "aproksymacja" grafu G: drzewo T bezpiecznych prefiksów wirusów. Linie przerywane oznaczają przejścia zabronione. Brakujące niezabronione przejścia są uzupełnione w grafie G.



Wierzchołki są ponumerowane, jednakże każdy węzeł utożsamiamy z pewnym prefiksem wzorca. Załóżmy, że $syn(x, a)$ oznacza węzeł xa, jeśli xa jest bezpiecznym prefiksem wzorca. Jeśli $xa \in W$, to $syn(x, a) = N$, gdzie N jest specjalną wartością oznaczającą, że krawędzi etykietownej symbolem a z węzła x nigdy być nie może. W pozostałych przypadkach $syn(x, a) =$ **NIL**.

Właściwa konstrukcja grafu G

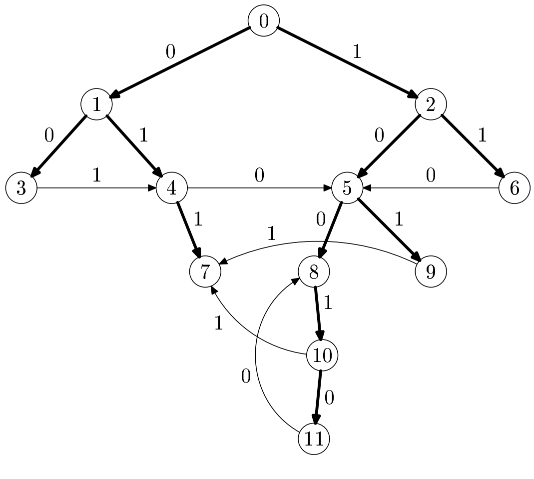
Algorytm korzysta istotnie z drzewa prefiksów T zbudowanego wcześniej i opisanego funkcją syn.

Algorytm
Inicjujemy $\sigma(\epsilon, a) = \epsilon$, $\sigma(\alpha, a) = NIL$, dla a=0,1 i $\alpha \neq \epsilon$ (słowa α odpowiadają węzłom w drzewie).
Przechodząc drzewo T wszerz, dla każdego węzła $x \neq root$ wykonaj:

```
niech y oraz a będą takie, że x = syn(y, a);
if σ(y, a) ≠ NIL then begin
  temp := σ(y, a); σ(y, a) := x;
  for b:=0 to 1 do begin
     if syn(x, b)=N ∨ syn(temp,b)=N then
        σ(x, b):=NIL
     else if syn(temp, b) ≠ NIL then
        σ(x, b):=syn(temp, b)
     else σ(x, b):=σ(temp, b)
  end
end
```

Na zakończenie algorytmu należy ograniczyć graf G do jego podgrafu indukowanego przez wierzchołki osiągalne z wierzchołka startowego.
Uwaga: Założenie, że żaden wirus nie jest podsłowem innego wirusa było użyteczne przy wprowadzeniu pomocniczego drzewa prefiksów. Algorytm konstrukcji grafu automatycznie wykrywa takie sytuacje, tak więc założenie to nie jest istotne.

Rys. 2 Graf G dla przykładowego zbioru wirusów. Pogrubione krawędzie (przejścia) są pozostałością z drzewa prefiksów T, pozostałe krawędzie powstały w wyniku działania algorytmu konstrukcji G.



Sprawdzanie acykliczności

Aby stwierdzić, czy w grafie jest cykl należy sprawdzić, czy wierzchołki grafu można posortować topologicznie. Można zastosować dwie alternatywne metody:

- sortowanie topologiczne przy pomocy DFS,
- sortowanie topologiczne poprzez usuwanie wierzchołków ze stopniem wejściowym równym zero.

Nie opisujemy dokładnie tych metod ponieważ sprawdzanie acykliczności grafu wystąpiło już w wielu poprzednich zadaniach olimpijskich.
Dla naszego przykładowego zbioru wirusów nieskończona ścieżka "idąca" po węzłach grafu G.

$$0,\ 2,\ 5,\ (8,\ 10,\ 11)^{\infty}$$

odpowiada nieskończonemu bezpiecznemu słowu:

$$100(100)^{\infty}$$

Algorytm alternatywny

Wprowadzamy dwie operacje na zbiorze W. Niech $x, y \in W$.

Usuwanie y. Jeśli x jest podsłowem y, to y usuwamy z W.

Skracanie y. Jeśli $y = za$, gdzie $a = 0 \ \vee \ a = 1$, oraz $x \ = \ u\bar{a}$ i u jest sufiksem z, to $y{:=}z$.

Symbol $\bar{a}$ oznacza $1 - a$.
Na przykład jeśli $x = 0010$, $y = 001010011$, to skracając y dostajemy $y = 00101001$.
Operację skracania możemy zastosować dlatego, że jeśli w pewnym *nieskończonym* ciągu wystąpi skrócone y, to albo w tym ciągu wystąpi samo y, albo x.
Niech W' będzie zbiorem powstałym z W za pomocą jednej z tych operacji, wtedy:

$$W \text{ jest bezpieczny} \Leftrightarrow W' \text{ jest bezpieczny.}$$

Łatwo zauważyć, że W jest niebezpieczny wtedy i tylko wtedy, gdy po pewnej liczbie operacji otrzymamy zbiór zawierający oba pojedyncze symbole jako słowa. Natomiast, jeśli w wyniku dowolnego ciągu operacji otrzymamy zbiór zawierający słowo o długości większej niż jeden, to W jest bezpieczny.

Można wówczas skonstruować dowolnie długi bezpieczny ciąg postępując w następujący sposób: zaczynamy od ciągu pustego i dokładamy po jednym symbolu, do chwili gdy otrzymujemy ciąg niebezpieczny (gdy sufiks należy do w); zamieniamy ostatnio dodany symbol na przeciwny i kontynuujemy naszą konstrukcję.

Nasz algorytm polega na stosowaniu w dowolnej kolejności jednej z powyższych operacji, aż otrzymany zbiór zawierający obie litery $0, 1$, albo zbiór zawierający dłuższe słowo, który to zbiór nie da się dalej zmienić za pomocą jednej z naszych operacji. W pierwszym przypadku W jest niebezpieczny. Dla naszego przykładowego zbioru wirusów W z rysunku 1 od razu widać, że jest on bezpieczny, ponieważ żadna z dwóch operacji się do niego nie stosuje.

Przykład

Rozważmy przykład zbioru, który nie jest bezpieczny. W poniższym ciągu przekształceń podkreślamy słowo biorące udział w przekształceniu, słowo to jest usuwane lub skracane o jeden symbol od końca.

$\{000, \underline{010}, 11\} \rightarrow \{\underline{000}, 01, 11\} \rightarrow \{00, \underline{01}, 11\} \rightarrow \{\underline{00}, 0, 11\} \rightarrow \{0, \underline{11}\} \rightarrow \{0, 1\}$.
Zatem $\{000, 010, 11\}$ jest niebezpieczny, gdyż końcowy zbiór $\{0, 1\}$ jest niebezpieczny.

TESTY

Do sprawdzania rozwiązań zawodników użyto 10 testów:

- WIR1-3.IN — małe testy poprawnościowe,
- WIR4.IN — średni test poprawnościowy,
- WIR5.IN — test wydajnościowy sprawdzający metodę wyszukiwania cyklu, tylko bardzo długie cykle,
- WIR6.IN — mały test wydajnościowy, jeden cykl w automacie,
- WIR7.IN — duży test wydajnościowy, duża liczba możliwych nieskończonych słów,
- WIR8.IN — przypadek pesymistyczny dla iloczynu automatów, dużo krótkich słów,
- WIR9.IN — duży test wydajnościowy, mała liczba możliwych nieskończonych słów,
- WIR10.IN — średni test wydajnościowy, tylko jeden cykl w automacie.

Marcin Kubica
autor zadania, opracowanie

Marek Pawlicki
program wzorcowy

Narciarze

Drużyna narciarska organizuje trening na Bajtogórze. Na północnym stoku góry znajduje się jeden wyciąg. Wszystkie trasy prowadzą od górnej do dolnej stacji wyciągu. W trakcie treningu członkowie drużyny będą razem startować z górnej stacji wyciągu i spotykać się przy dolnej stacji. Poza tymi dwoma punktami trasy, zawodników nie mogą się przecinać, ani stykać. Wszystkie trasy muszą cały czas prowadzić w dół.

Mapa tras narciarskich składa się z sieci polan połączonych przecinkami. Każda polana leży na innej wysokości. Dwie polany mogą być bezpośrednio połączone co najwyżej jedną przecinką. Zjeżdżając od górnej do dolnej stacji wyciągu można tak wybrać drogę, żeby odwiedzić dowolną polanę (choć być może nie wszystkie w jednym zjeździe). Trasy narciarskie mogą się przecinać tylko na polanach i nie prowadzą przez tunele, ani estakady.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `NAR.IN` *mapę tras narciarskich,*
- *wyznaczy maksymalną liczbę zawodników, którzy mogą brać udział w treningu,*
- *zapisze wynik do pliku tekstowego* `NAR.OUT`.

Wejście

W pierwszym wierszu pliku wejściowego `NAR.IN` *znajduje się jedna liczba całkowita n równa liczbie polan, $2 \leqslant n \leqslant 5\,000$.*

W każdym z kolejnych $n-1$ wierszy znajduje się ciąg liczb całkowitych pooddzielanych pojedynczymi odstępami. Liczby w $(i+1)$-szym wierszu pliku określają, do których polan prowadzą w dół przecinki od polany nr i. Pierwsza liczba k w wierszu określa liczbę tych polan, a kolejne k liczb to ich numery, które są uporządkowane wg ułożenia prowadzących do nich przecinek w kierunku ze wschodu na zachód. Polany są ponumerowane liczbami od 1 do n. Górna stacja wyciągu znajduje się na polanie numer 1, a dolna na polanie numer n.

Wyjście

W pierwszym i jedynym wierszu pliku wyjściowego `NAR.OUT` *powinna znajdować się dokładnie jedna liczba całkowita — maksymalna liczba narciarzy mogących wziąć udział w treningu.*

Przykład

Dla pliku wejściowego `NAR.IN`:

```
15
5 3 5 9 2 4
1 9
2 7 5
2 6 8
1 7
1 10
2 14 11
2 10 12
2 13 10
3 13 15 12
2 14 15
1 15
1 15
1 15
```

poprawną odpowiedzią jest plik tekstowy `NAR.OUT`*:*

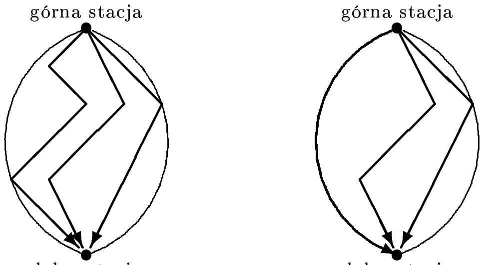
```
3
```

ROZWIĄZANIE

Zadanie to możemy sformułować w języku teorii grafów. Mamy dany acykliczny, planarny graf skierowany — wierzchołki tego grafu to polany i stacje wyciągu, a krawędzie to przecinki. W grafie tym mamy wyróżnione dwa wierzchołki reprezentujące górną i dolną stację wyciągu. Z górnej stacji wyciągu osiągalne są wszystkie wierzchołki w grafie. Podobnie, z każdego wierzchołka w grafie osiągalna jest dolna stacja wyciągu. Ponadto stacje wyciągu (jako najwyżej i najniżej położone wierzchołki) znajdują się na obwodzie grafu. Należy wyznaczyć maksymalną liczbę k takich ścieżek w grafie, które prowadzą od górnej do dolnej stacji wyciągu i poza stacjami wyciągu nie mają (parami) żadnych wspólnych wierzchołków.

Załóżmy chwilowo, że w grafie nie ma krawędzi łączącej bezpośrednio górną i dolną stację wyciągu. Jeżeli taka krawędź jest, to możemy ją usunąć, obliczyć wynik, po czym zwiększyć go o 1.

Zadanie to możemy rozwiązać za pomocą algorytmu zachłannego. Wyobraźmy sobie, że wybraliśmy k ścieżek spełniających warunki zadania. Oznaczmy te ścieżki, od lewej do prawej (ze wschodu na zachód), przez $s_1, \ldots, s_k$. Zauważmy, że bez łamania warunków zadania możemy przesunąć skrajnie lewą (wschodnią) ścieżkę s_1 w lewo tak, aby prowadziła przez wierzchołki leżące na lewym pół-obwodzie grafu.



Rys. 1

Tak więc, bez zmniejszenia ogólności, możemy przyjąć, że s_1 biegnie przez wierzchołki leżące na lewym pół-obwodzie grafu (zobacz rysunek 1).

Podobnie możemy poprzesuwać pozostałe ścieżki. Zauważmy, że wśród wszystkich możliwych ścieżek, które prowadzą od górnej do dolnej stacji wyciągu i nie mają (za wyjątkiem stacji wyciągu) wspólnych wierzchołków z s_1, istnieje skrajnie lewa ścieżka. Bez naruszenia warunków zadania możemy przyjąć, że s_2 jest właśnie taką ścieżką. Analogicznie ustalamy kolejne ścieżki. Przyjmujemy, że s_i jest skrajnie lewą spośród wszystkich możliwych ścieżek, które prowadzą od górnej do dolnej stacji wyciągu i nie mają (za wyjątkiem stacji wyciągu) wspólnych wierzchołków z żadną ze ścieżek $s_1, \ldots, s_{i-1}$.

Wyłania się następujący schemat algorytmu:

(1) $i := 1$;

(2) powtarzaj, dopóki można, konstrukcję kolejnych ścieżek:

  (a) wyznacz s_i jako skrajnie lewą ścieżkę prowadzącą od górnej do dolnej stacji wyciągu i (dla $i > 1$) nie przechodzącą (za wyjątkiem stacji wyciągu) przez wierzchołki należące do ścieżek $s_1, \ldots, s_{i-1}$,

  (b) jeśli konstrukcja ścieżki się udała, to zwiększ i o 1;

(3) jeżeli w zadanym grafie była krawędź prowadząca bezpośrednio z górnej do dolnej stacji wyciągu, to zwiększ i o 1;

(4) $(i - 1)$ jest równe szukanej liczbie ścieżek.

Pozostał do rozwiązania problem jak wyznaczyć skrajnie lewą ścieżkę w odpowiednim podgrafie zadanego grafu. Ścieżkę taką konstruujemy począwszy od górnej stacji wyciągu, korzystając z procedury przeszukiwania grafu w głąb (zobacz [13] lub [10]). Ważne jest, aby w trakcie przeszukiwania przeglądać krawędzie wychodzące z danego wierzchołka w kolejności od lewej do prawej (ze wschodu na zachód), czyli zgodnie z kolejnością, w jakiej zostały one podane w pliku wejściowym. W momencie osiągnięcia dolnej stacji wyciągu przerywamy przeszukiwanie — szukana ścieżka znajduje się wówczas na stosie. W trakcie przeszukiwania grafu kolorujemy odwiedzone wierzchołki. Zauważmy, że pokolorowane zostają wierzchołki tworzące skonstruowaną ścieżkę oraz niektóre wierzchołki leżące na lewo od tej ścieżki. Konstruując kolejne ścieżki omijamy wierzchołki już pokolorowane (za wyjątkiem stacji wyciągu).

Powyższy algorytm znajdowania skrajnie lewej ścieżki można zaimplementować w następujący sposób.

```
const
  MaxN = 5 000; { Maksymalna liczba polan (wierzchołków grafu) }
type
  { Lista sąsiedztwa jednego wierzchołka. }
  { Element 0-wy zawiera liczbę wierzchołków na liście. }
  TListaSasiedztwa = array [0..MaxN] of Integer;
  PListaSasiedztwa = ↑TListaSasiedztwa;
var
  { Graf reprezentowany jako listy sąsiedztwa wierzchołków. }
  T: array [1..MaxN] of PListaSasiedztwa;
  { Tablica do kolorowania odwiedzonych wierzchołków. }
  Odwiedzony: array [1..MaxN] of Boolean;
  N: Integer; { Liczba wierzchołków. }
function Wglab (w: Integer): Boolean;
{ Realizuje przeszukanie grafu T w głąb począwszy od wierzchołka w.
  Zwraca true w przypadku dotarcia do dolnej stacji (sukcesu). }
var
  i: Integer;
  s: Boolean;
begin
  if w = N then
    { Osiągnięto dolną stację wyciągu. }
    Wglab := true
  else begin
    Odwiedzony[w] := true; s := false; i := 1;
    { Przeszukuj aż do pierwszego sukcesu. }
    while not s and (i ⩽ T[w]↑[0]) do begin
      if not Odwiedzony [T[w]↑[i]] then
        s := Wglab (T[w]↑[i]);
      Inc (i)
    end;
    Wglab := s
  end
end;
```

Szukaną liczbę ścieżek możemy wyznaczyć w następujący sposób:

```
function ZnajdzRozwiazanie: Integer;
{ Znajduje rozwiązanie metodą poszukiwania najbardziej
  skrajnej ścieżki. }
var
  i: Integer;
  function UsunTraseBezposrednia: Boolean;
  { Usuwa ewentualną bezpośrednią krawędź 1→N,
    zwraca true, gdy takie połączenie istniało. }
  var
    i, j, lnast: Integer;
```

```
    zn: Boolean;
  begin
    i := 1; zn := false; lnast := T[1]↑[0];
    while (i ⩽ lnast) and not zn do
      if T[1]↑[i] = N then
        zn := true
      else
        Inc (i);
    if zn then begin
      for j := i to lnast - 1 do T[1]↑[j] := T[1]↑[j+1];
      Dec (T[1]↑[0])
    end
    UsunTraseBezposrednia := zn
  end
begin { ZnajdzRozwiazanie. }
  for i := 1 to N do Odwiedzony [i] := false;
  if UsunTraseBezposrednia then i := 1 else i := 0;
  while Wglab (1) do Inc (i);
  ZnajdzRozwiazanie := i
end
```

Z twierdzenia Eulera wiadomo (zobacz [10]), że w każdym spójnym grafie planarnym liczba krawędzi m jest takiego rzędu jak liczba wierzchołków n. Dokładniej, dla $n \geqslant 3$ mamy $n-1 \leqslant m \leqslant 3n-6$. Jeżeli więc nie będziemy alokowali w pamięci całych tablic sąsiedztwa, a jedynie ich wypełnione fragmenty, to uzyskamy złożoność pamięciową algorytmu rzędu n.

Sprawdzenie, czy w grafie jest krawędź łącząca bezpośrednio górną i dolną krawędź wyciągu wymaga przejrzenia wszystkich krawędzi wychodzących z górnej stacji wyciągu. Tak więc złożoność czasowa tej operacji jest rzędu n. Zauważmy, że w trakcie przeglądania grafu po każdej krawędzi przechodzimy co najwyżej raz. Stąd, złożoność czasowa przeszukiwania grafu jest rzędu n. Tak więc, złożoność czasowa całego rozwiązania jest rzędu n.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 8-miu testów. Testy te można podzielić na trzy grupy:

- NAR1–3.IN — proste testy sprawdzające poprawność rozwiązania,
- NAR4–6.IN — bardziej skomplikowane testy sprawdzające poprawność rozwiązania,
- NAR7–8.IN — testy badające złożoność rozwiązania.

W poniższej tabelce podano wielkości i wyniki dla poszczególnych testów.

Test	n	m	i
1	2	1	1
2	15	26	2
3	20	37	4
4	900	1 740	2
5	2 502	2 901	5
6	4 006	9 001	1 001
7	4 600	10 005	3
8	4 913	9 524	16

Adam Borowski autor zadania | **Marcin Sawicki** opracowanie | **Tomasz Waleń** program wzorcowy

Paski

Paski *to gra dwuosobowa. Rekwizytami potrzebnymi do gry są paski w trzech kolorach: czerwonym, zielonym i niebieskim. Wszystkie paski czerwone mają wymiary* $c \times 1$*, zielone* $z \times 1$*, a niebieskie* $n \times 1$*, gdzie* c*,* z *i* n *są liczbami naturalnymi. Gracze dysponują nieograniczoną pulą pasków każdego koloru (jeśli pewnych pasków zabraknie, to zawsze można dodrukować).*

Plansza do gry jest prostokątem o wymiarach $p \times 1$ *i składa się z* p *pól o wymiarach* 1×1*.*

Gracze wykonują ruchy na przemian. Ruch polega na ułożeniu na planszy paska dowolnego koloru. Obowiązują przy tym następujące zasady:

- *pasek nie może wystawać poza planszę,*
- *nie wolno przykrywać (nawet częściowo) pasków ułożonych wcześniej,*
- *końce paska muszą pokrywać się z brzegami pól planszy*

Przegrywa gracz, który jako pierwszy nie może wykonać ruchu zgodnie z zasadami gry. **Pierwszy gracz** *to gracz, który wykonuje pierwszy ruch w grze. Mówimy, że pierwszy gracz ma strategię wygrywającą, jeżeli niezależnie od posunięć drugiego gracza zawsze może wygrać.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `PAS.IN` *wymiary pasków i co najmniej jednej planszy,*
- *dla każdej planszy stwierdzi, czy pierwszy gracz posiada strategię wygrywającą,*
- *zapisze wyniki w pliku tekstowym* `PAS.OUT`*.*

Wejście

Pierwszy wiersz zbioru wejściowego `PAS.IN` *zawiera trzy liczby naturalne* c*,* z *i* n*,* $1 \leqslant c, n, z \leqslant 1000$*, równe długościom pasków, odpowiednio, czerwonych, zielonych i niebieskich. Liczby w wierszu są poodzielane pojedynczymi znakami odstępu.*

Drugi wiersz pliku `PAS.IN` *zawiera jedną liczbę* m*,* $1 \leqslant m \leqslant 1000$*, równą liczbie różnych plansz do rozpatrzenia.*

Wiersze od 3 to $m+2$ *zawierają po jednej liczbie* p*,* $1 \leqslant p \leqslant 1000$*. Liczba w wierszu* $i+2$ *jest długością* i*-tej planszy.*

Wyjście

Plik wyjściowy `PAS.OUT` *powinien zawierać* m *wierszy. W* i*-tym wierszu pliku powinna być zapisana jedna liczba:*

- *1, jeżeli pierwszy gracz ma strategię wygrywającą na* i*-tej planszy,*
- *2, w przeciwnym przypadku.*

Przykład

Dla pliku wejściowego `PAS.IN`*:*

```
1 5 1
3
1
5
6
```

poprawną odpowiedzią jest plik tekstowy `PAS.OUT`*:*

```
1
1
2
```

ROZWIĄZANIE

W teorii gier istnieje pojęcie gry zamkniętej. Gry, które polegają na kolejnych jawnych posunięciach dwu graczy (np. gra w paski, a także szachy, warcaby, kółko i krzyżyk, wilk i owce itd.) są przykładami gier zamkniętych. Znane twierdzenie mówi, że wszystkie takie gry są niesprawiedliwe lub czcze, tzn. że zawsze albo jeden z graczy ma strategię wygrywającą, albo obaj gracze mogą co najwyżej wymusić remis lub nieskończoną rozgrywkę.

Gra w paski jest ponadto grą kategoryczną, czyli zawsze kończy się wygraną jednego z graczy. Jest zatem niesprawiedliwa: któryś z graczy musi mieć strategię prowadzącą do zwycięstwa.

Będąc młodym informatykiem, z pewnością zdajesz sobie sprawę, Szanowny Czytelniku, że teoretycznie można znajdować strategie wygrywające poprzez przeszukiwanie drzewa gry, i że chyba dla żadnej rozsądnej gry nie jest to wykonalne w praktyce. Można polemizować, czy paski są rozsądną grą, w każdym razie przeszukiwanie całego drzewa jest w tym wypadku zbyt pracochłonne.

Ruch w grze w paski można jednak postrzegać jako rozbicie planszy na mniejsze kawałki. Podsuwa nam to pomysł skorzystania z metod programowania dynamicznego. Jeśli umielibyśmy powiedzieć coś o tych „mniejszych kawałkach", to być może umielibyśmy również sklasyfikować jakoś wyjściową planszę?

Chwila namysłu pozwala stwierdzić, że nie wystarczy informacja, który z graczy ma strategię wygrywającą na każdym z kawałków planszy, by ustalić, kto ją ma na całej planszy. Potrzebujemy zatem subtelniejszego rozróżnienia pomiędzy planszami, jednak takiego, z którego łatwo wynika, kto na danej planszy zwycięży. Jakiego? O tym już za chwilę...

Skąd ten XOR?

Szanowny Czytelniku! Być może poznałeś już program wzorcowy i wiesz, że w magiczny sposób korzysta on z właściwości operacji XOR. W tym artykule postaram się przekonać Cię, że w rozwiązaniu tym nie ma aż tak wiele magii. Użyję języka teorii grup, co w wielu miejscach uprości opis. Mimo wszystko, zajmie to jednak trochę czasu, a skoro Ty oglądałeś już rozwiązanie wzorcowe, to pewnie nie należysz do cierpliwych. Dlatego już teraz przedstawię główną myśl rozwiązania. Oto ona:

W grze w paski, jeśli w jakiejś chwili plansza daje się podzielić na dwa identyczne fragmenty, to wiadomo, że gracz, na którego przypada kolejka, musi przegrać: drugi gracz będzie powtarzał jego ruchy, tyle że symetrycznie, na drugim z identycznych fragmentów planszy. Jeśli pierwszy gracz może jeszcze wykonać ruch, to również i drugi może po nim wykonać ruch symetryczny. A jeśli pierwszy nie może, to przegrał!

Ta cecha gry w paski jest w pełni analogiczna do znanej właściwości operacji XOR, polegającej na tym, że dla dowolnej liczby k mamy k XOR $k = 0$. Liczba k odpowiada jednemu z dwu identycznych egzemplarzy planszy, zaś 0 oznacza porażkę pierwszego gracza.

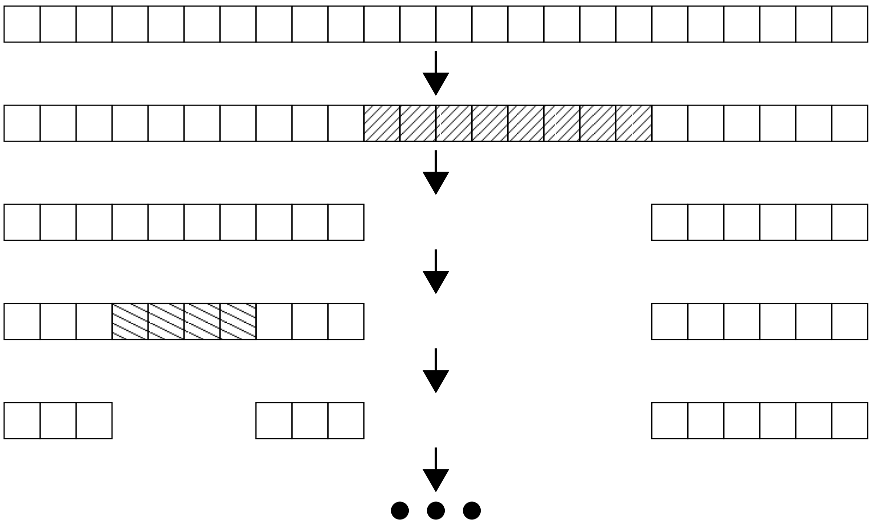
Oczywiście jest to tylko intuicja. Dlatego zachęcam nawet Ciebie, Niecierpliwy Czytelniku, do dalszej lektury.

Terminologia

Na początku gry plansza jest pojedynczym prostokątem zbudowanym z $p \times 1$ kwadratowych pól. Kolejne ruchy polegają na zakrywaniu fragmentów planszy. Pola zakryte nie biorą dalej udziału w grze. Natomiast pola wolne po każdym ruchu tworzą pewien układ spójnych prostokątów. Przykładowo, na rysunku 1 widzimy początek pewnej rozgrywki na planszy długości 24. Po pierwszym ruchu wolne pola tworzą układ dwu prostokątów o długościach 10 i 6, a po następnym mamy trzy prostokąty o długościach 3, 3 i 6.

Definicja. Spójny prostokąt, zbudowany z wolnych pól, do którego skrajów przylegają pola zajęte lub brzegi planszy (czyli nie dający się powiększyć o kolejne wolne pola bez naruszania spójności), nazywać będziemy *obszarem*. Każdy układ obszarów (również początkowy, czyli zbudowany z jednego obszaru $p\times 1$) będziemy nazywać *planszą*.

Rys. 1 W trakcie rozgrywki plansza rozpada się na coraz mniejsze obszary



Zwróćmy uwagę, że z punktu widzenia strategii rozgrywki (w szczególności tego, kto wygra), nie ma znaczenia kolejność, w jakiej na danej planszy występują poszczególne obszary. Czyli plansze 3, 3, 6 oraz 3, 6, 3 możemy uważać za jednakowe. Dlatego odpowiednim pojęciem matematycznym do reprezentowania plansz będzie *multizbiór*, którego elementami będą liczby całkowite dodatnie — długości poszczególnych obszarów.

Definicja. *Multizbiór* jest to zbiór par (x,n), gdzie x jest *elementem* multizbioru, zaś $n = 1,2,3,\ldots$ jest liczbą całkowitą dodatnią, określającą w ilu egzemplarzach dany element należy do multizbioru. Oczywiście dla każdego x do multizbioru może należeć co najwyżej jedna para postaci (x,n). Operacje sumy i części wspólnej są dla multizbiorów zdefiniowane w sposób naturalny. Na przykład, jeśli $A = \{(a,3),(b,1),(c,1)\}$, zaś $B = \{(a,2),(b,3),(d,5)\}$, to $A\cup B = \{(a,5),(b,4),(c,1),(d,5)\}$ oraz $A\cap B = \{(a,2),(b,1)\}$.

Plansze przedstawimy jako multizbiory długości tworzących je obszarów. Przykładowo, przedstawieniem planszy 3, 3, 6, jak również planszy 3, 6, 3, jest ten sam multizbiór $\{(3,2),(6,1)\}$. W dalszych rozważaniach potrzebna nam będzie operacja łączenia dwu plansz, czyli budowania planszy, składającej się ze wszystkich obszarów, wchodzących w skład obu plansz. Operacji łączenia plansz dokładnie odpowiada suma multizbiorów.

Odrobina abstrakcji

Rozważamy grę rozpoczynającą się od planszy $p\times 1$ i prowadzoną przy użyciu pasków o długościach c, z i n. W każdym stanie gry plansza będzie zatem zbudowana z obszarów o długościach $\leqslant p$. Oczywiście, nie każda plansza zbudowana z takich obszarów może się pojawić w naszej rozgrywce, w szczególności suma długości obszarów nigdy nie przekroczy p.

Na nasze potrzeby wygodnie jednak będzie o tym zapomnieć i zbadać zbiór X wszystkich plansz, zbudowanych ze skończonej liczby obszarów, z których każdy ma długość $\leqslant p$. Zatem X jest rodziną skończonych multizbiorów, których elementy są liczbami od 1 do p. Dzięki temu, w zbiorze X wykonalne jest działanie łączenia plansz (czyli dodawania multizbiorów) — to znaczy, że plansza uzyskana przez połączenie dwu elementów zbioru X również należy do tego zbioru. Zwróćmy uwagę, że w zbiorze X znajduje się również plansza pusta, czyli pusty multizbiór (odpowiada ona całkowitemu wypełnieniu planszy paskami). Oznaczmy ją przez $\emptyset$. Dla dowolnego $x \in X$ mamy oczywiście $x \cup \emptyset = x$, czyli $\emptyset$ jest elementem neutralnym dla działania łączenia plansz. Nadto łączenie plansz jest, jak każdy widzi, przemienne i (jak sama nazwa wskazuje) łączne.

Definicja. Zbiór Y z działaniem dwuargumentowym $+$ nazywamy *grupą przemienną* albo *abelową*, jeśli spełnia on następujące warunki:

- Łączność: dla dowolnych $y_1, y_2, y_3 \in Y$ mamy $y_1 + (y_2 + y_3) = (y_1 + y_2) + y_3$.
- Przemienność: dla dowolnych $y_1, y_2 \in Y$ zachodzi $y_1 + y_2 = y_2 + y_1$.
- Istnieje element $e \in Y$ taki, że dla każdego $y \in Y$ zachodzi $y + e = e + y = y$. Element e nazywamy *elementem neutralnym* grupy.
- Dla każdego elementu $y \in Y$ istnieje taki $y' \in Y$, że $y + y' = e$. Przyjmuje się oznaczenie $-y = y'$. Mówimy, że $-y$ jest *elementem odwrotnym* do y.

Pojęcie grupy przemiennej powinno być dobrze znane każdemu: najprostsze przykłady to liczby całkowite (lub wymierne, lub rzeczywiste) z dodawaniem (w tym kontekście mówi się raczej o elemencie przeciwnym, a nie odwrotnym), liczby wymierne (lub rzeczywiste) bez zera z mnożeniem, wektory na płaszczyźnie z dodawaniem wektorów, itd.

Widzimy, że zbiór X z działaniem $\cup$ spełnia wszystkie warunki oprócz odwracania. Przypomnijmy sobie jednak, że na planszy, dającej się podzielić na dwa identyczne fragmenty gracz, na którego przypada kolejka, musi przegrać, tak samo jak na planszy $\emptyset$. Z punktu widzenia strategii można więc uważać, że plansza postaci $x \cup x$ jest równoważna planszy $\emptyset$. Postaramy się zatem tak przerobić zbiór X z działaniem $\cup$, by każdy jego element był odwrotny sam do siebie. Z grubsza biorąc, utożsamimy elementy postaci $x \cup x$ z $\emptyset$. Znów będzie nam potrzebna

Definicja. Relacja dwuargumentowa $\sim$ na zbiorze Y jest *relacją równoważności*, jeśli jest

- *zwrotna*, tzn. dla każdego $y \in Y$ zachodzi $y \sim y$,
- *symetryczna*, tzn. dla każdych $y_1, y_2 \in Y$ jeśli $y_1 \sim y_2$, to również $y_2 \sim y_1$,
- *przechodnia*, czyli jeśli dla $y_1, y_2, y_3 \in Y$ zachodzi $y_1 \sim y_2$ oraz $y_2 \sim y_3$, to także $y_1 \sim y_3$.

Relacje równoważności są matematycznym sposobem na utożsamienie pewnych elementów zbioru: każda relacja równoważności na zbiorze Y jednoznacznie zadaje jego *podział*, czyli rodzinę $\mathcal{A}$ podzbiorów zbioru Y taką, że $y_1 \sim y_2$ wtedy i tylko wtedy, gdy dla pewnego elementu B rodziny $\mathcal{A}$ zachodzi jednocześnie $y_1 \in B$ oraz $y_2 \in B$. Taką rodzinę oznacza się przez ${}^{Y}\!/_{\sim} = \mathcal{A}$. Każdy element $y \in Y$ należy do dokładnie jednego elementu $B \in {}^{Y}\!/_{\sim}$; ten jedyny element oznaczamy $[y] = B$ i nazywamy *klasą abstrakcji* elementu y. Oczywiście $[y_1] = [y_2]$ wtedy i tylko wtedy, gdy $y_1 \sim y_2$. Czyli $[y] \subseteq Y$ jest zbiorem tych i tylko tych elementów zbioru Y, które właśnie utożsamiliśmy z y za pomocą relacji $\sim$.

Dobrym pomysłem okazuje się następujące określenie naszej relacji: $x_1 \equiv x_2$ wtedy i tylko wtedy, gdy na planszy $x_1 \cup x_2$ gracz rozpoczynający musi przegrać (tzn. jego przeciwnik posiada strategię wygrywającą). Wiemy zatem, że zawsze $x \equiv x$. Poza tym jest oczywiście wiele innych par x_1, x_2 takich, że $x_1 \equiv x_2$ — jak się niedługo okaże, działa to na naszą korzyść. Przypominam, że rozmiary pasków c, z, n są cały czas ustalone. Dla innych rozmiarów pasków możemy otrzymać zupełnie inną relację!

Z drugiej strony, uważny Czytelnik niedługo już zorientuje się, że utożsamianie plansz, na których gracz rozpoczynający ma strategię wygrywającą, byłoby dużo gorszym pomysłem. Poza bardzo prostymi przypadkami, musielibyśmy wówczas w trosce o przechodniość naszej relacji konsekwentnie posklejać ze sobą wszystko.

Zadanie dla czytelnika. Proszę sprawdzić, że $\equiv$ jest istotnie relacją równoważności (spełnia definicję). Wskazówka: aby udowodnić przechodniość, należy, znając strategię wygrywającą drugiego gracza dla plansz $x_1 \cup x_2$ oraz $x_2 \cup x_3$, opisać sposób (algorytm) gry dla drugiego gracza, który zagwarantuje mu zwycięstwo na planszy $x_1 \cup x_3$.

Zagadka tylko dla wtajemniczonych: czy jeśli zdefiniowalibyśmy $\simeq$ jako najmniejszą relację równoważności na X, która utożsamia wszystkie plansze postaci $x \cup x$ z $\emptyset$, to czy $\equiv = \simeq$? A gdyby dodatkowo zarządać, by $\simeq$ było najmniejszą kongruencją na X ze względu na $\cup$?

No dobrze, powiecie, utożsamiliśmy jakieś elementy, ale przecież działanie $\cup$ było określone na X, a my mamy teraz inny zbiór ${}^{X}\!/_{\equiv}$. A jednak wszystko jest pod kontrolą! Przypomnijmy tylko jedno pojęcie, z którym każdy spotkał się (lub spotka) w szkole:

Definicja. Relacja równoważności $\sim$ jest *kongruencją* na zbiorze Y ze względu na działanie $+$, jeśli dla dowolnych $y_1, y_1', y_2, y_2' \in Y$ jeśli tylko $y_1 \sim y_1'$ oraz $y_2 \sim y_2'$, to zachodzi również $y_1 + y_2 \sim y_1' + y_2'$.

Przypadek „szkolny" to relacja przystawania modulo n, która utożsamia liczby całkowite $a, b \in \mathbf{Z}$ wtedy i tylko wtedy, gdy $a-b$ dzieli się przez n. Jest to relacja równoważności, co więcej jest to kongruencja ze względu na dodawanie, odejmowanie i mnożenie.

Kongruencja pozwala wykonywać działania w zbiorze klas abstrakcji ${}^{Y}/_{\sim}$: aby np. dodać dwie klasy, bierzemy dowolną parę ich elementów i dodajemy, a następnie znajdujemy klasę abstrakcji sumy. To, że relacja jest kongruencją, nie oznacza nic więcej, niż że tak zdefiniowane działanie jest dobrze określone w zbiorze ${}^{Y}/_{\sim}$, tzn. że różne wybory reprezentantów prowadzą w wyniku do tej samej klasy abstrakcji.

Na całe szczęście, $\equiv$ jest kongruencją w X ze względu na $\cup$. Proszę to sprawdzić wprost z definicji, dowód będzie podobny do dowodu faktu, że relacja $\equiv$ jest przechodnia.

Mamy zatem

Fakt. ${}^{X}/_{\equiv}$ z działaniem zadanym przez $\cup$ jest grupą przemienną, z elementem neutralnym $[\emptyset]$ (klasa abstrakcji zbioru pustego). Działanie w tej grupie również będziemy oznaczać $\cup$, zaś element neutralny po prostu przez $0 = [\emptyset]$.

Istotnie, podzielenie X przez $\equiv$ nie mogło popsuć łączności, ani przemienności $\cup$, ani neutralności $\emptyset$ (która przeniosła się na całą klasę abstrakcji $[\emptyset]$), i oczywiście uzyskaliśmy elementy odwrotne: każdy element jest odwrotny sam do siebie.

Do czego była nam potrzebna ta „odrobina" abstrakcji?

Potrzeba wyjaśnić, czemu miał służyć ten długi ciąg definicji. Celów było kilka. Po pierwsze, możemy teraz zwięźle sformułować nasz cel: algorytm ma stwierdzać, czy dla danej planszy x, składającej się z jednego obszaru długości p, zachodzi $x \equiv \emptyset$, czyli czy $[x] = 0$ (gdyż zgodnie z naszą definicją, $x \equiv \emptyset$ wtedy i tylko wtedy, gdy drugi gracz ma strategię wygrywającą na planszy $x \cup \emptyset = x$).

Po drugie, bez określenia naszej grupy $\langle {}^{X}/_{\equiv}, \cup, 0 \rangle$ wiele faktów, które w tej chwili są dla nas najzupełniej oczywiste, należałoby za każdym razem dowodzić „ręcznie" w sposób podobny do dowodu przechodniości relacji $\equiv$ (a więc byłyby one pozostawione jako zadania dla Ciebie, Szanowny Czytelniku). Przykład takiego faktu: jeśli na planszy x strategię wygrywającą ma drugi gracz (w szczególności, jeśli np. $x = x' \cup x'$), to na planszy $x_1 \cup x$ strategię wygrywającą ma ten sam gracz, co na planszy x_1. Uzasadnienie: $[x] = 0$, więc $[x_1] \cup [x] = [x_1]$. Własności grup pozwalają nam wycisnąć wszystko co się da z przeprowadzonych już przez Ciebie dowodów, że $\equiv$ jest relacją równoważności i kongruencją ze względu na $\cup$.

Po trzecie i najważniejsze, jak się za chwilę okaże, klasy abstrakcji naszej relacji dają się efektywnie obliczać metodą programowania dynamicznego.

Trochę porządków

Na drodze do rozwiązania czeka nas jeszcze jedno niespodziewane odkrycie: klasy plansz (czyli klasy abstrakcji relacji $\equiv$) dają się uporządkować!

Zauważmy najpierw, że dla dowolnej planszy x, jeśli $[x] \neq 0$, to z planszy x można jednym ruchem dojść do pewnej planszy $x_0 \in 0 = [\emptyset]$. Innymi słowy, z każdej planszy x wygrywającej dla gracza, na którego przypada kolejka, można zawsze jednym ruchem przejść do jakiejś planszy x_0 przegrywającej dla gracza, którego kolejka wypadnie po tym ruchu (a więc który będzie musiał położyć pierwszy pasek na planszy x_0). Jest to ten właśnie ruch, który graczowi rozpoczynającemu grę na planszy x gwarantuje zwycięstwo. Oczywiście, na planszy x być może da się wykonać wiele innych ruchów, które mogą prowadzić do plansz z innych klas. Ważne jest jednak istnienie co najmniej jednego wygrywającego posunięcia dla każdej spośród plansz z klasy $[x] \neq 0$.

Wykorzystajmy konsekwentnie tę obserwację i określmy relację dwuargumentową $\succ$ na zbiorze ${}^{X}/_{\equiv}$ jak następuje: dla klas $B_1, B_2 \in {}^{X}/_{\equiv}$ zachodzi $B_1 \succ B_2$ wtedy i tylko wtedy, gdy z każdej planszy $x_1 \in B_1$ można jednym ruchem dojść do pewnej planszy $x_2 \in B_2$.

Na razie wiemy tylko, że dla $[x] = B \neq 0$ jest $B \succ 0$. Nie wiemy ani czy może zajść $0 \succ B$ dla jakiegoś B, ani czy może być $B_1 \succ B_2$ dla jakichkolwiek $B_1, B_2 \neq 0$.

Fakt 1. (najważniejszy) Jeśli $B_1, B_2 \in {}^X/_\equiv$ i $B_1 \neq B_2$, to albo $B_1 \succ B_2$, albo $B_2 \succ B_1$.

Dowód. Niech nie zachodzi ani $B_1 \succ B_2$, ani $B_2 \succ B_1$. Czyli istnieje plansza $x_1 \in B_1$, z której jednym ruchem nie da się dojść do żadnej spośród plansz z klasy B_2. Analogicznie, istnieje pewna plansza $x_2 \in B_2$, na której żadne posunięcie nie prowadzi do klasy B_1. Rozważmy planszę $x_1 \cup x_2$. Jeśli gracz, na którego przypada kolejność, wykona na tej połączonej planszy ruch w obrębie części x_1, to przejdzie do pewnej planszy $x_1' \cup x_2$, gdzie $x_1' \notin B_2$. A zatem $[x_1'] \neq [x_2]$. Do tej nierówności dodajmy stronami $[x_2]$, otrzymamy $[x_1'] \cup [x_2] \neq [x_2] \cup [x_2]$, czyli $[x_1' \cup x_2] \neq 0$. Jest to zatem ruch prowadzący do porażki. Gracz przegra również, jeśli położy pasek w obrębie planszy x_2. Czyli plansza $x_1 \cup x_2$ jest dla niego przegrywająca, to znaczy $[x_1 \cup x_2] = 0$, skąd $B_1 = B_2$, co kończy dowód.

Fakt 2. Dla żadnego $B \in {}^X/_\equiv$ nie zachodzi $B \succ B$.

Dowód. Jeśli by tak było, to startując od dowolnej planszy $x_1 \in B$ moglibyśmy wykonać ruch, prowadzący do pewnej planszy $x_2 \in B$, następnie jednym ruchem przeszlibyśmy do planszy $x_3 \in B$ i tak w nieskończoność. A nie jest to możliwe, bo po każdym ruchu liczba wolnych pól zmniejsza się co najmniej o długość najkrótszego z pasków c, z, n!

Fakt 3. Dla żadnych $B_1, B_2 \in {}^X/_\equiv$ nie zachodzi jednocześnie $B_1 \succ B_2$ i $B_2 \succ B_1$.

Dowód. Podobnie jak poprzednio, prowadziłoby to do istnienia nieskończonej rozgrywki, w której na zmianę przechodzilibyśmy z planszy z klasy B_1 do planszy z klasy B_2 i z powrotem.

Fakt 4. Jeśli $B_1 \succ B_2$ oraz $B_2 \succ B_3$, to $B_1 \succ B_3$.

Dowód. Ten fakt jest chyba dość zaskakujący, gdyż oznacza, że z planszy z klasy B_1 do planszy z klasy B_3 można zawsze przejść *jednym* ruchem, a nie dwoma (choć oczywiście dwoma też się da). Uzasadnienie jest jednak proste: nie może oczywiście być $B_1 = B_3$ (wykluczyliśmy to przed chwilą). Musi być zatem $B_3 \succ B_1$ lub $B_1 \succ B_3$. Jednak w tym pierwszym przypadku znów znaleźlibyśmy nieskończoną rozgrywkę, prowadzącą kolejno z klasy B_1 do B_2, z B_2 do B_3, a z B_3 na powrót do B_1 i tak w kółko. Pozostaje jedyna możliwość $B_1 \succ B_3$.

Fakt 5. Klas plansz w X jest skończenie wiele.

Dowód. (Pamiętamy, że sam zbiór X był nieskończony.) Niech x_m oznacza planszę, zbudowaną z jednego obszaru długości m (czyli $x_m = \{(m,1)\}$), gdzie $1 \leqslant m \leqslant p$. Wówczas oczywiście każda plansza $x \in X$ daje się przedstawić jako suma

$$x = \underbrace{x_1 \cup x_1 \cup \ldots \cup x_1}_{n_1} \cup \underbrace{x_2 \cup \ldots \cup x_2}_{n_2} \cup \ldots \cup \underbrace{x_p \cup \ldots \cup x_p}_{n_p}$$

Oczywiście dopuszczamy, by dla pewnych k było $n_k = 0$. Planszę x reprezentuje multizbiór $\{(1, n_1), \ldots, (p, n_p)\}$, trzeba tylko pominąć te pary (k, n_k), dla których $n_k = 0$.

Aby znaleźć klasę planszy x zauważmy, że $\underbrace{[x_m] \cup \ldots \cup [x_m]}_{n_m}$ jest równe $[x_m]$ dla n_m nieparzystego, zaś 0 dla n_m parzystego. Suma ta upraszcza się zatem do co najwyżej jednego składnika. W związku z tym, mamy

$$[x] = B_1 \cup B_2 \cup \ldots \cup B_p$$

gdzie odpowiednio $B_m = [x_m]$ lub $B_m = 0$, zależnie od parzystości liczby n_m. Sum tej postaci jest zaś tylko skończona liczba, dokładnie 2^p.

Zwróćmy uwagę, że klas plansz może nie być dokładnie 2^p, tylko mniej, gdyż może się np. zdarzyć, że dla pewnych $m' \neq m''$ jest $[x_{m'}] = [x_{m''}]$. Np. jeśli $c, z, n \geqslant 3$, to oczywiście $0 = [x_1] = [x_2]$.

Wniosek z faktów 1–5: Klasy plansz można ponumerować kolejnymi liczbami naturalnymi tak, by klasa 0 miała numer 0 i by dla dowolnych dwóch klas B', B'' numer klasy B' był większy od numeru klasy B'' wtedy i tylko wtedy, gdy $B' \succ B''$.

Na podstawie udowodnionych faktów wniosek ten powinien wydawać się oczywisty. Takie własności może po prostu mieć tylko naturalny porządek na liczbach naturalnych $0 < 1 < 2 \ldots < l-1 < l$. Jako zadanie proponuję uważnie uzasadnić ten wniosek np. przez indukcję lub posiłkując się pojęciami "dagu" (acyklicznego grafu skierowanego) i sortowania topologicznego. Klasę o numerze j będziemy od teraz oznaczać B^j. W szczególności, $0 = B^0$.

Programowanie dynamiczne

Najwyższa pora przejść do opisu algorytmu. Przypominam, że naszym celem jest obliczanie klas poszczególnych plansz.

Na podstawie wyciągniętego przed chwilą wniosku wiemy, że z każdej planszy z klasy B^j można jednym ruchem przejść do każdej z klas $B^0, B^1, \ldots, B^{j-1}$.

Wiemy też, że nie zachodzi $B_j \succ B_j$, czyli z pewnej planszy z klasy B_j nie da się wykonać ruchu tak, by pozostać w klasie B_j. Prawdą jest jednak mocniejsze stwierdzenie: nigdy nie da się wykonać ruchu tak, by pozostać w tej samej klasie. Dowód jest bardzo prosty: gdyby tak było dla pewnej planszy $x \in B$, to na planszy $x \cup x$ gracz pierwszy wykonałby to posunięcie na jednej z połówek, doprowadzając do sytuacji $x' \cup x$, gdzie $x, x' \in B$, czyli $[x] = [x'] = B$, skąd $[x' \cup x] = 0$. Z sytuacji dla siebie przegrywającej $x \cup x$ przeszedłby zatem do sytuacji, w której przegrać musi jego oponent, co oczywiście jest sprzeczne z naszymi definicjami i ze zdrowym rozsądkiem.

Widzimy zatem co następuje: numer planszy x to jedyna taka liczba j, że z planszy x da się jednym posunięciem przejść do każdej z klas $B^0, B^1, \ldots, B^{j-1}$, i nie da się żadnym posunięciem przejść do żadnej spośród plansz, należących do klasy B^j.

Ta obserwacja poprowadzi nas już prosto do algorytmu, który indukcyjnie wyznacza numery klas plansz. Dla porządku przytoczę jeszcze tylko jedną definicję:

Definicja. Jeśli Y z działaniem $+$ jest grupą, a $Y' \subseteq Y$ takim podzbiorem, który również jest grupą ze względu na to samo działanie, to mówimy, że Y' jest *podgrupą* Y.

Przykładowo, liczby całkowite podzielne przez 3 stanowią podgrupę grupy liczb całkowitych z dodawaniem.

Oznaczmy przez $X_m \subseteq X$ zbiór wszystkich plansz, zbudowanych z obszarów, z których żaden nie jest dłuższy niż m. Oczywiście $X = X_p$.

Niech $\mathcal{X}_m$ oznacza zbiór tych wszystkich klas plansz (elementów grupy ${}^X/_{\equiv}$), które mają reprezentanta z X_m. Jest on zamknięty ze względu na działanie $\cup$ i oczywiście jest podgrupą grupy ${}^X/_{\equiv}$. Mamy zatem $0 = \mathcal{X}_0 \subseteq \mathcal{X}_1 \subseteq \cdots \subseteq \mathcal{X}_p = {}^X/_{\equiv}$. Zauważmy też, że każdy ze zbiorów $\mathcal{X}_m$ składa się z klas o kolejnych numerach, od B^0 do pewnego B^N. Jest tak, gdyż jeśli N jest największą liczbą taką, że $B^N \in \mathcal{X}_m$, to pewna plansza $x \in B^N$ składa się z obszarów nie dłuższych niż m. Można z niej jednym ruchem dojść do każdej z klas $B^0, B^1, \ldots, B^{N-1}$, a ruch w grze w paski nie może spowodować, by którykolwiek z obszarów wydłużył się. To dowodzi, że każda z tych klas ma reprezentanta w X_m, czyli należy do $\mathcal{X}_m$.

Podam teraz algorytm, który po każdym kroku dla kolejnych liczb m potrafi

- wyznaczyć numer klasy każdej planszy ze zbioru X_m
- wykonywać działanie $\cup$ w grupie $\mathcal{X}_m$ (to znaczy, wyznaczać numer klasy $B^i \cup B^j$ dla $B^i, B^j \in \mathcal{X}_m$)

Pierwszy krok algorytmu jest trywialny: wiemy, że plansza pusta ma numer 0, a grupa $\mathcal{X}_0 = 0$ jest grupą trywialną.

Załóżmy więc, że potrafimy wyznaczać klasy dla plansz z X_{m-1} oraz że dla takich klas potrafimy wykonywać działanie $\cup$. Weźmy planszę x_m zbudowaną z jednego obszaru długości m. Przejrzyjmy wszystkie możliwe ruchy, jakie możemy wykonać na tej planszy. Każdy z tych ruchów prowadzi do planszy zbudowanej z 1 lub 2 obszarów, z których każdy ma długość $\leqslant m-1$. Zatem zgodnie z założeniem potrafimy wyznaczyć numery klas wszystkich tych plansz (jeśli jakiś ruch rozspaja naszą planszę, to numer klasy wyznaczymy, obliczając $\cup$ klas obu obszarów, na które rozpadła się nasza plansza). Mamy wówczas pewność, że klasa, do której należy x_m, jest najmniejszą liczbą naturalną, jaka nie pojawiła się wśród otrzymanych numerów klas plansz. Jeśli np. możliwe ruchy prowadzą z x_m do plansz z klas o numerach $0, 1, 2, 3, 6, 9, 13$, to znaczy, że $[x_m] = B^4$.

Są tu dwie możliwości. Albo $[x_m] \in \mathcal{X}_{m-1}$, skąd wynika, że $\mathcal{X}_m = \mathcal{X}_{m-1}$ i nic więcej w tym kroku algorytmu nie musimy robić, albo z planszy x_m można dojść do każdej z klas, należących do $\mathcal{X}_{m-1}$, czyli klasa $[x_m]$ nie jest nam jeszcze znana. Ten drugi przypadek musimy rozważyć dokładniej.

Wiemy, że $\mathcal{X}_{m-1} = \{B^0, \ldots, B^{M-1}\}$ dla pewnego M. Stąd natychmiastowy wniosek, że $[x_m] = B^M \in \mathcal{X}_m \setminus \mathcal{X}_{m-1}$.

Rozważmy teraz klasy postaci $[x_m] \cup B^j$, gdzie $0 \leqslant j \leqslant M-1$.

- Wszystkie one są elementami $\mathcal{X}_m$.

- Żadna z nich nie należy do $\mathcal{X}_{m-1}$, gdyż jeśli $[x_m] \cup B^j = B^k \in \mathcal{X}_{m-1}$, to oczywiście mielibyśmy również $[x_m] = [x_m] \cup B^j \cup B^j = B^k \cup B^j \in \mathcal{X}^{m-1}$ (suma elementów B^j i B^k grupy $\mathcal{X}^{m-1}$ również należy do tej grupy).

- Klasa każdej planszy $x \in X_m$, jeśli nie należy do grupy $\mathcal{X}_{m-1}$, to może być przedstawiona w tej postaci. Plansza $x \in X_m$ jest bowiem połączeniem pewnej (parzystej bądź nieparzystej) liczby obszarów długości m oraz planszy, należącej do X_{m-1}, czyli $x = x_m \cup x_m \cup \ldots \cup x_m \cup x'$, gdzie $x' \in X_{m-1}$. Stąd istotnie $[x] = [x_m] \cup [x']$ lub $[x] = [x'] \in \mathcal{X}_{m-1}$, gdzie $[x'] = B^j$ dla pewnego $j < M$.

- Dla różnych wartości j klasy $[x_m] \cup B^j$ są między sobą różne, bo do nierówności $B^j \neq B^{j'}$ można stronami dodać $[x_m]$.

W takim razie mamy

$$\mathcal{X}_m = \{B^0, B^1, \ldots, B^{M-1},\ [x_m] \cup B^0, [x_m] \cup B^1, \ldots, [x_m] \cup B^{M-1}\}$$

Co więcej, skoro umieliśmy wykonywać działanie $\cup$ w grupie $\mathcal{X}_{m-1}$, to pamiętając, że $[x_m] \cup [x_m] = 0$, umiemy je wykonać także w $\mathcal{X}_m$!

Teraz już nietrudno zgadnąć, że numerem klasy $[x_m] \cup B^j$ będzie $M + j$. Aby się o tym przekonać, wybierzmy dowolnego reprezentanta $x \in B^j$ i rozważmy planszę $x_m \cup x$. Na tej planszy możemy wykonać ruch albo w obrębie części x_m, albo x.

Wykonując ruch w części x możemy przejść do dowolnie wybranej spośród klas $[x_m] \cup B^i$, gdzie $0 \leqslant i < j$. Być może uda nam się również przejść do którejś z klas $[x_m] \cup B^k$ dla $k > j$, ale dla każdego takiego k znajdziemy reprezentanta $x \in B^j$, dla którego nie będzie to możliwe. Wszystko odbywa się analogicznie jak na samej planszy $x \in B^j$, bez dołączonej części x_m.

Natomiast kładąc pasek w części x_m, możemy przejść do dowolnej klasy $B' \in \mathcal{X}_{m-1}$. Wystarczy wziąć $B'' = B' \cup B^j$. Wiemy, że $B'' \in \mathcal{X}_{m-1}$, więc istnieje taki ruch na planszy x_m, który prowadzi do klasy B''. Wykonując ten ruch na planszy $x_m \cup x$, istotnie otrzymamy w wyniku planszę klasy $B'' \cup B^j = (B' \cup B^j) \cup B^j = B'$. Widzimy też, że ruch w obrębie x_m nie doprowadzi nas do żadnej klasy spoza $\mathcal{X}_{m-1}$.

Podsumowując, widzimy wyraźnie, że:

$B^{M-1} \prec [x_m] = [x_m] \cup B^0 \prec [x_m] \cup B^1 \prec [x_m] \cup B^2 \prec \cdots \prec [x_m] \cup B^{M-1}$.

Ponieważ wiemy skądinąd, że $\mathcal{X}_m = \{B^0, \ldots, B^{2M-1}\}$, to musi rzeczywiście zachodzić $[x_m] \cup B^j = B^{M+j}$, dla każdego $0 \leqslant j \leqslant M - 1$.

W ten sposób z nawiązką wypełniamy wszystkie warunki indukcyjne, dotyczące naszego algorytmu.

Pozostaje jeszcze jeden drobiazg: widzimy, że po pierwszym kroku znamy tylko jedną planszę (pustą), a po każdym następnym liczba znanych nam plansz (tzn. liczba elementów grupy $\mathcal{X}_m$) albo nie zmienia się, albo się podwaja. Będzie zatem zawsze potęgą dwójki. W szczególności, w całym powyższym rozumowaniu M było potęgą dwójki. Po tej uwadze nie powinno być dla Czytelnika najmniejszym problemem udowodnienie przez indukcję, że działaniu $\cup$ w grupie klas plansz odpowiada dokładnie operacja XOR na numerach tych klas.

W ten sposób wszystkie tajemnice gry w paski zostały ujawnione.

TESTY

Na potrzeby testowania rozwiązań przygotowano 15 testów. Każdy z nich zawierał p przypadków plansz do rozpatrzenia — było to zawsze kolejno p plansz o długościach od 1 do p.

W poniższej tabeli zestawiono wartości c, z, n oraz p dla poszczególnych testów. Ponadto, k oznacza największą liczbę taką, że co najmniej jedna z plansz x_m przy $m \leqslant p$ należy do klasy B^k. Dalej, W_1 oznacza liczbę tych plansz spośród $x_1, \ldots, x_p$, które są wygrywające dla gracza rozpoczynającego (czyli należą do klasy B^j dla $j > 0$), zaś W_2 — liczbę plansz przegrywających dla rozpoczynającego.

Nr	**c**	**z**	**n**	**p**	**k**	$\mathbf{W_1}$	$\mathbf{W_2}$
0	1	5	1	6	1	3	3
1	2	2	2	10	3	7	3
2	3	3	3	20	4	16	4
3	4	4	4	50	5	43	7
4	2	18	26	70	15	63	7
5	1	5	6	905	256	903	2
6	2	3	4	100	16	99	1
7	3	5	5	100	12	94	6
8	2	2	111	200	17	181	19
9	1000	1000	1000	1000	1	1	999
10	2	38	226	1000	150	984	16
11	3	221	233	1000	103	977	23
12	2	16	60	1000	65	985	15
13	3	29	91	1000	139	987	13
14	3	51	51	1000	64	980	20

Testy 4 i 5 oraz 6, 7, 8 i 9 zostały zgrupowane.

PODSUMOWANIE

Zadanie „Paski” było prawdopodobnie jednym z najtrudniejszych zadań w historii Olimpiady. Mimo że było zadaniem z pierwszego etapu, a więc można było poświęcić na nie wiele czasu i posiłkować się przy pracy literaturą, to jednak poprawnie rozwiązał je tylko jeden zawodnik.

Przedstawione w niniejszym artykule rozumowanie jest oparte na dowodzie poprawności programu wzorcowego. Autorem tego programu i dowodu jest Tomasz Waleń.

Podstawowe pojęcia teorii gier są przystępnie objaśnione w pierwszym rozdziale znakomitej książki [HS]. Serdecznie zachęcam do lektury!

Czytelnikom zainteresowanym tematem polecam również pozycję [JC], a w szczególności zawarte tam informacje o grze Nim. Gra ta okazuje się mieć wiele wspólnego z grą w paski. W niej także do znalezienia strategii wygrywającej przydaje się operacja XOR.

BIBLIOGRAFIA

[HS] Hugo Steinhaus *„Kalejdoskop matematyczny”*, WSiP Warszawa 1989
[JC] John Conway *„On Numbers and Games”*, Academic Press, Inc. New York 1976