Tomasz Idziaszek
Treść zadania, Opracowanie

Jakub Radoszewski, Piotr Smulewicz
Program

Dostępna pamięć: 512 MB. *OI, etap II, dzień drugi, 11.02.2016*

Arkanoid

Arkanoid *jest grą komputerową, w której za pomocą ruchomej paletki odbija się poruszającą się po planszy piłeczkę. Piłeczka ta zbija znajdujące się na planszy klocki, a celem gry jest zbicie ich wszystkich. Ci, którzy grali w tę grę, wiedzą, jak frustrujące i czasochłonne może być zbicie kilku ostatnich klocków. Warto mieć zatem program, który dla początkowego ustawienia planszy obliczy czas potrzebny na wygranie gry. Na potrzeby tego zadania zakładamy dla uproszczenia, że gracz gra bezbłędnie, tzn. że zawsze odbije piłeczkę i uczyni to środkiem paletki.*

Plansza ma długość m i wysokość n, przy czym m jest nieparzyste, a m i n są względnie pierwsze[1]*. Wprowadzamy na niej prostokątny układ współrzędnych: lewy dolny róg planszy ma współrzędne $(0, 0)$, a prawy górny współrzędne (m, n). Dla uproszczenia zakładamy, że piłeczka ma pomijalny rozmiar, a paletka pomijalną grubość. Paletka porusza się po prostej $y = 0$, natomiast początkowo piłeczka znajduje się w punkcie $(\frac{m}{2}, 0)$ i jej początkowy wektor prędkości to $(-\frac{1}{2}, \frac{1}{2})$.*

W momencie, w którym piłeczka dotknie paletki, brzegu planszy lub dowolnego klocka na planszy, odbija się idealnie sprężyście. Dodatkowo, dotknięty klocek zostaje zbity i znika z planszy. Po ilu jednostkach czasu wszystkie klocki zostaną zbite?

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite m, n i k ($m, n, k \geqslant 1$, $k \leqslant nm - 1$) oddzielone pojedynczymi odstępami, oznaczające wymiary planszy oraz początkową liczbę klocków na planszy. W kolejnych k wierszach znajdują się opisy klocków: i-ty z tych wierszy zawiera dwie liczby całkowite x_i i y_i ($1 \leqslant x_i \leqslant m$, $1 \leqslant y_i \leqslant n$) oddzielone pojedynczym odstępem, oznaczające, że na planszy znajduje się klocek, który jest prostokątem o przeciwległych wierzchołkach $(x_i - 1, y_i - 1)$ oraz (x_i, y_i). Możesz założyć, że na polu opisanym przez $x_i = \frac{m+1}{2}$, $y_i = 1$ nie znajduje się żaden klocek.

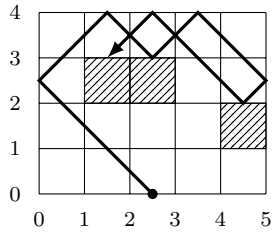
Wyjście

W jedynym wierszu standardowego wyjścia należy wypisać jedną liczbę całkowitą oznaczającą liczbę jednostek czasu, po których wszystkie klocki na planszy zostaną zbite.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
5 4 3 2 3 5 2 3 3	22

[1] Dwie liczby całkowite dodatnie są względnie pierwsze, jeśli ich największym wspólnym dzielnikiem jest 1.



Testy „ocen”:

`1ocen`: $m = 5$, $n = 4$, $k = 2$, *całkiem duży wynik,*

`2ocen`: $m = 11$, $n = 10$, *klocki tworzą szachownicę niedotykającą brzegów planszy,*

`3ocen`: $m = 99\,999$, $n = 100\,000$, *klocki na polach* $(\frac{m-1}{2}, 2)$, $(\frac{m-5}{2}, 2)$, $(\frac{m-9}{2}, 2), \ldots$,

`4ocen`: $m = 99\,999$, $n = 100\,000$, *jeden klocek na polu* $(1, 1)$, *duży wynik.*

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	$m, n \leqslant 100$, $k \leqslant 1000$	*25*
2	$n, m \leqslant 100\,000$, $k \leqslant 50$	*25*
3	$m, n, k \leqslant 100\,000$, *żaden klocek nie styka się bokiem z innymi klockami ani z brzegiem planszy (klocki mogą się stykać wierzchołkami)*	*25*
4	$m, n, k \leqslant 100\,000$	*25*

Rozwiązanie

Zadanie *Arkanoid* jest dość dobrym przykładem, jak można, wychodząc od prostego rozwiązania, aczkolwiek działającego wolno, stopniowo je poprawiać, aż do uzyskania rozwiązania efektywnego czasowo. W związku z tym w poniższym opracowaniu całkiem dokładnie prześledzimy proces konstrukcji rozwiązania. Zaczniemy od zaimplementowania symulacji procesu opisanego w zadaniu, a następnie będziemy dokładać kolejne obserwacje. Ponieważ dużo trudności w tym zadaniu było natury implementacyjnej, pozwolimy sobie na sporą ilość pseudokodu w opisie.

Symulujemy ruch piłeczki w najprostszy sposób

Wygodniej nam będzie operować na liczbach całkowitych, więc na początek wszystkie współrzędne pomnożymy przez 2. Po tej operacji lewy dolny róg planszy ma współrzędne $(0, 0)$, prawy górny współrzędne $(2m, 2n)$, a piłeczka zaczyna w punkcie o współrzędnych $(m, 0)$ i porusza się z początkowym wektorem prędkości $(-1, 1)$.

Piłeczka odbija się idealnie sprężyście, zatem jej tor jest łamaną, której każdy odcinek jest nachylony pod kątem 45° do obu brzegów planszy, a wektor prędkości jest równy $(\pm 1, \pm 1)$. Współrzędne (x, y) punktów całkowitych, w których może znaleźć się piłeczka, są takie, że jedna z liczb x, y jest parzysta, a druga nieparzysta. Istotnie: zaczynamy z punktu $(m, 0)$ dla nieparzystego m i w każdym ruchu każda z współrzędnych zmienia się o jeden (w górę lub w dół), co zmienia jej parzystość. Co więcej, wszystkie punkty odbicia piłeczki mają współrzędne całkowite.

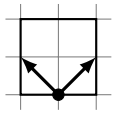
Najprostsze rozwiązanie polega na bezpośredniej symulacji ruchu piłeczki w kolejnych jednostkach czasu. Na początek spróbujmy zrobić to dla pustej planszy.

Tor piłeczki na planszy jest zdeterminowany przez jej położenie (x, y) oraz wektor prędkości (dx, dy). W dalszej części opisu czwórkę (x, y, dx, dy) będziemy nazywać *pozycją* piłeczki. W każdej jednostce czasu położenie piłeczki zostaje uaktualnione o wektor prędkości i zmienia się na $(nx, ny) = (x+dx, y+dy)$. Po tej operacji musimy sprawdzić, czy piłeczka odbije się od ściany (brzegu planszy). Jeśli po pojedynczym kroku dotyka ona pionowego brzegu planszy (czyli nx jest równe 0 lub $2m$), to jej składowa dx prędkości zmienia się na przeciwną. Analogicznie, gdy piłeczka dotyka poziomego brzegu (dla ny równego 0 lub $2n$), jej składowa dy się zmienia. Zauważmy, że zawsze zmienia się co najwyżej jedna składowa, jako że piłeczka nigdy nie znajdzie się w rogu planszy (gdyż obie współrzędne każdego z rogów są parzyste). Na zmiennej *czas* będziemy przechowywać liczbę jednostek czasu, które upłynęły od początku ruchu piłeczki. Poniższa funkcja Ruszaj implementuje ruch piłeczki w pojedynczej jednostce czasu.

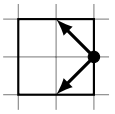
```
1:  function Ruszaj
2:  begin
3:    x := x + dx;
4:    y := y + dy;
5:    czas := czas + 1;
6:    if x = 0 or x = 2m then { odbicie od pionowego brzegu }
7:      dx := −dx
8:    else if y = 0 or y = 2n then { odbicie od poziomego brzegu }
9:      dy := −dy;
10: end
```

Teraz dodajmy obsługę klocków. Zakładamy tak jak w treści zadania, że plansza jest podzielona na $m \times n$ pól, a niektóre z nich mogą zawierać klocki. Jeśli po pojedynczym kroku piłeczka nie dotyka brzegu planszy, to musimy sprawdzić, czy dotyka boku któregoś z klocków. Innymi słowy, czy piłeczka w trakcie następnego ruchu wjechałaby na pole, na którym znajduje się klocek. Pole to będziemy reprezentować przez współrzędne (sx, sy) jego środka.

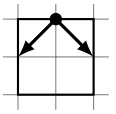
Każde pole ma 4 punkty, w których piłeczka może dotknąć jego brzegu (co, uwzględniając kierunki, daje 8 możliwych pozycji; patrz też rys. 1). Punkty, w których



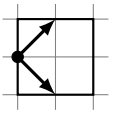
$(sx, sy-1, 1, 1)$
$(sx, sy-1, -1, 1)$



$(sx+1, sy, -1, 1)$
$(sx+1, sy, -1, -1)$



$(sx, sy+1, -1, -1)$
$(sx, sy+1, 1, -1)$



$(sx-1, sy, 1, -1)$
$(sx-1, sy, 1, 1)$

Rys. 1: Osiem pozycji, w których może znaleźć się piłeczka wjeżdżająca na pole o środku (sx, sy).

piłeczka dotyka pionowego boku, mają współrzędną x parzystą, a punkty, w których dotyka poziomego boku pola, mają współrzędną x nieparzystą. Funkcja SrodekPola wyznacza środek pola, którego dotyka piłeczka o położeniu w punkcie (x, y) i wektorze prędkości (dx, dy) skierowanym do wewnątrz tego pola:

```
1:  function SrodekPola(x, y, dx, dy)
2:  begin
3:    if x mod 2 = 0 then begin { piłeczka dotyka pionowego boku pola }
4:      sx := x + dx;
5:      sy := y;
6:    end else begin { piłeczka dotyka poziomego boku pola }
7:      sx := x;
8:      sy := y + dy;
9:    end
10:   return (sx, sy);
11: end
```

Na podstawie powyższych rozważań możemy też napisać ogólną funkcję Odbij, która uaktualnia prędkość piłeczki przy odbiciu zarówno od ścian, jak i klocków:

```
1: function Odbij
2: begin
3:   if x mod 2 = 0 then { pionowy bok }
4:     dx := −dx
5:   else { poziomy bok }
6:     dy := −dy;
7: end
```

Jeśli piłeczka właśnie ma odbić się od ściany, to funkcja SrodekPola zwróci współrzędne punktu leżącego poza planszą. Poniższa funkcja sprawdza, czy mamy do czynienia z takim przypadkiem:

```
1: function Sciana(x, y, dx, dy)
2: begin
3:   (sx, sy) := SrodekPola(x, y, dx, dy);
4:   return sx < 0 or sx > 2m or sy < 0 or sy > 2n;
5: end
```

Aby móc szybko sprawdzać, gdzie na planszy są klocki, użyjemy najprostszej metody i będziemy stan całej planszy trzymać w dwuwymiarowej tablicy. Przyjmujemy, że jeśli na polu o środku (sx, sy) znajduje się klocek, to $klocek[sx, sy] =$ **true**:

```
1: function InicjujPlansze
2: begin
3:   for sx := 1 to 2m − 1 step 2 do
4:     for sy := 1 to 2n − 1 step 2 do
5:       klocek[sx, sy] := false;
6:   for i := 1 to k do
7:     klocek[2x_i − 1, 2y_i − 1] := true;
8: end
```

Jesteśmy już gotowi do napisania funkcji Ruszaj, uwzględniającej klocki:

```
1: function Ruszaj
2: begin
3:   x := x + dx;
4:   y := y + dy;
5:   czas := czas + 1;
6:   (sx, sy) := SrodekPola(x, y, dx, dy);
7:   if Sciana(x, y, dx, dy) then { odbicie od ściany }
8:     Odbij
9:   else if klocek[sx, sy] then begin { odbicie od klocka }
10:    klocek[sx, sy] := false;
11:    k := k − 1;
12:    Odbij;
13:  end
14: end
```

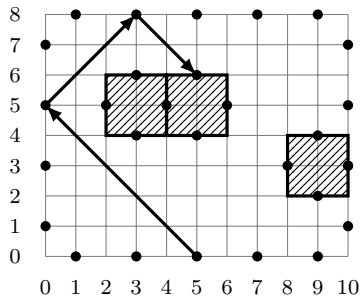
Główna pętla programu symulującego ruch piłeczki jest bardzo prosta. Inicjujemy $czas := 0$ oraz $(x, y, dx, dy) := (m, 0, -1, 1)$, a następnie wywołujemy funkcję Ruszaj dopóty, dopóki liczba niezbitych klocków k jest dodatnia.

Przeanalizujmy czas działania tego algorytmu. Inicjowanie planszy kosztuje czas $O(mn + k)$. Przy niesprzyjającym układzie klocków, potencjalnie możemy odwiedzić większość planszy, by zbić kolejny klocek, więc liczbę jednostek czasu pomiędzy kolejnymi odbiciami od klocków oszacujemy przez $O(mn)$. Zatem cały algorytm działa w czasie $O(mnk)$. Rozwiązanie to zostało zapisane w pliku `arks1.cpp`; przechodzi ono pierwsze podzadanie.

Przyspieszamy ruch piłeczki bez odbić

Zauważmy, że jeśli na dużej planszy jest mało klocków, to często wielokrotnie będziemy wywoływali funkcję Ruszaj bez wywoływania funkcji Odbij. Spróbujmy przyspieszyć to tak, aby funkcja Ruszaj poruszała piłeczką aż do następnego miejsca, w którym nastąpi odbicie (od ściany lub od klocka).

Przypomnijmy, że pozycja piłeczki jest czwórką (x, y, dx, dy). O zbiorze pozycji, które może przyjąć piłeczka, możemy też myśleć jak o zbiorze wierzchołków pewnego



Rys. 2: Plansza z zaznaczonymi wierzchołkami grafu: każdy pogrubiony punkt (x, y) oznacza cztery wierzchołki-pozycje $(x, y, \pm 1, \pm 1)$. Trzy zaznaczone krawędzie to $graf[5, 0, -1, 1] = (0, 5, 5)$, $graf[0, 5, 1, 1] = (3, 8, 3)$ i $graf[3, 8, 1, -1] = (5, 6, 2)$.

grafu. Na potrzeby poprzedniego algorytmu dwa wierzchołki-pozycje połączone były (skierowaną) krawędzią, jeśli piłeczka mogła przejść z jednej pozycji do drugiej w pojedynczej jednostce czasu. Taki graf był duży, bo zawierał aż $O(mn)$ wierzchołków i krawędzi. Ponieważ ruch piłeczki jest zdeterminowany przez jej pozycję, to z każdego wierzchołka wychodzi co najwyżej jedna krawędź. (Piszemy „co najwyżej jedna", a nie „dokładnie jedna", gdyż z niektórych pozycji kontynuowanie ruchu piłeczki nie jest możliwe; są to pozycje odbić, w których pozycja piłeczki zmienia się za sprawą funkcji Odbij).

Teraz zbudujemy ważony graf, w którym dwa wierzchołki-pozycje połączone będą krawędzią o wadze ℓ, jeśli piłeczka może przejść z jednej pozycji do drugiej w ℓ jednostkach czasu, nie wykonując przy tym żadnego odbicia. Interesować nas będą jedynie wierzchołki-pozycje, w których może dojść do odbicia, czyli które leżą na brzegach planszy i na bokach klocków. Dzięki temu nasz graf będzie miał jedynie $O(m+n+k)$ wierzchołków i tyle samo krawędzi (rys. 2).

Dla ustalenia notacji przyjmijmy, że $graf[x, y, dx, dy] = (nx, ny, \ell)$ oznacza, że piłeczka startująca z pozycji (x, y, dx, dy) musi przebyć ℓ kroków, żeby wykonać następne odbicie; będzie ono na pozycji (nx, ny, dx, dy).

Symulacja ruchu piłeczki do następnego odbicia może wyglądać następująco:

```
 1: function Ruszaj
 2: begin
 3:   (x, y, ℓ) := graf[x, y, dx, dy];
 4:   czas := czas + ℓ;
 5:   if Sciana(x, y, dx, dy) then { odbicie od ściany }
 6:     Odbij
 7:   else begin { odbicie od klocka }
 8:     UsunKlocek(x, y, dx, dy);
 9:     k := k − 1;
10:     Odbij;
11:   end
12: end
```

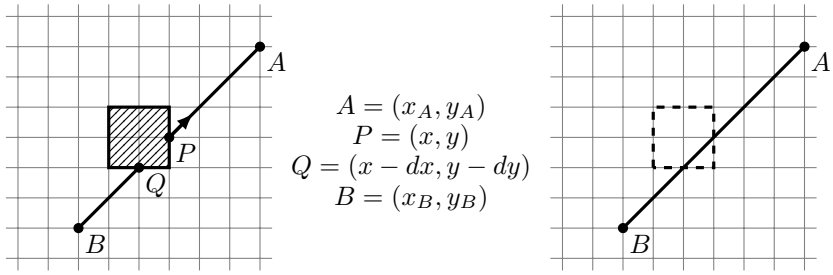
Przejdźmy teraz do kwestii budowania grafu, czyli wyznaczenia wartości *graf*. Załóżmy, że chcemy obliczyć wartość dla wierzchołka (x, y, dx, dy). Poruszamy się zatem z punktu (x, y) w kierunku (dx, dy), aż nie natrafimy na taką pozycję (nx, ny, dx, dy), za którą jest już ściana lub pole z klockiem. Liczba wykonanych ruchów to oczywiście wartość bezwzględna różnicy $nx - x$. Zauważmy, że skoro wierzchołek (x, y, dx, dy) też leżał na brzegu planszy lub przy klocku (bo tylko takie nas interesują), to wyznaczyliśmy jednocześnie wartość dla wierzchołka $(nx, ny, -dx, -dy)$, bo wystarczy rozważyć ruch piłeczki do tyłu.

```
1:  function DodajKrawedz(x, y, dx, dy)
2:  begin
3:    nx := x;
4:    ny := y;
5:    while not Sciana(nx, ny, dx, dy) and
6:              not klocek[SrodekPola(nx, ny, dx, dy)] do begin
7:      nx := nx + dx;
8:      ny := ny + dy;
9:    end
10:   ℓ := abs(nx − x);
11:   graf[x, y, dx, dy] := (nx, ny, ℓ);
12:   graf[nx, ny, −dx, −dy] := (x, y, ℓ);
13: end
```

Aby zbudować cały graf, należy wywołać powyższą funkcję dla wszystkich wierzchołków. Z tego co powiedzieliśmy o symetrii, wystarczy to zrobić dla punktów, z których idziemy przekątną w górę.

```
1:  function InicjujGraf
2:  begin
3:    for x := 1 to 2m − 1 step 2 do begin
4:      DodajKrawedz(x, 0, 1, 1);
5:      DodajKrawedz(x, 0, −1, 1);
6:    end
7:    for y := 1 to 2n − 1 step 2 do begin
8:      DodajKrawedz(0, y, 1, 1);
9:      DodajKrawedz(2m, y, −1, 1);
10:   end
11:   for i := 1 to k do begin
12:     sx := 2x_i − 1;
13:     sy := 2y_i − 1;
14:     DodajKrawedz(sx, sy + 1, 1, 1);
15:     DodajKrawedz(sx, sy + 1, −1, 1);
16:     DodajKrawedz(sx − 1, sy, −1, 1);
17:     DodajKrawedz(sx + 1, sy, 1, 1);
18:   end
19: end
```

Zauważmy, że w sytuacji, gdy jakiś klocek dotyka bokiem ściany lub dwa klocki sąsiadują bokiem, w naszym grafie będą istnieć krawędzie o wadze 0. Co prawda



Rys. 3: Scalanie pary krawędzi przechodzących przez usunięty klocek.

nie używamy ich w funkcji Ruszaj, ale przy usuwaniu klocków będziemy korzystali z faktu, że te krawędzie istnieją. Często dość łatwo przeoczyć tego rodzaju przypadki szczególne; z tego też powodu w testach w jednym z podzadań mieliśmy gwarancję, że takie przypadki nie występują.

Uaktualnianie grafu po usunięciu klocka

Po każdym usunięciu klocka struktura naszego grafu przestaje opisywać sytuację na planszy. Musimy zatem odpowiednio zaktualizować graf. Dla ustalonego klocka mamy osiem pozycji na jego brzegu, które były wierzchołkami, natomiast po usunięciu klocka już nimi nie będą.

Zobaczmy to na przykładzie: ustalmy pewien klocek oraz pozycję (x, y, dx, dy) na jego brzegu; oznaczmy $P = (x, y)$ (rys. 3). Niech $graf[x, y, dx, dy] = (x_A, y_A, \ell_A)$, zatem kolejne odbicie nastąpi na pozycji (x_A, y_A, dx, dy) po ℓ_A jednostkach czasu. Dopóki klocek leży na planszy, to oczywiście $graf[x_A, y_A, -dx, -dy] = (x, y, \ell_A)$. Jeśli jednak klocek zostanie usunięty, to startując z wierzchołka $(x_A, y_A, -dx, -dy)$, piłeczka nie zatrzyma się w wierzchołku $(x, y, -dx, -dy)$, ale przeleci dalej do wierzchołka $(x - dx, y - dy, -dx, -dy)$, a ponieważ jest to wierzchołek, który znajdował się na zbitym klocku, wobec tego piłeczka poruszy się z niego aż do $graf[x - dx, y - dy, -dx, -dy] = (x_B, y_B, \ell_B)$. Wierzchołki $(x, y, -dx, -dy)$ oraz $(x - dx, y - dy, -dx, -dy)$ mogą zostać usunięte z grafu (bo w nich już nie będzie odbić), natomiast wierzchołkom $(x_A, y_A, -dx, -dy)$ oraz (x_B, y_B, dx, dy) należy uaktualnić wychodzące z nich krawędzie, tak jak w poniższej funkcji.

```
1: function ScalKrawedz(x, y, dx, dy)
2: begin
3:     (x_A, y_A, ℓ_A) := graf[x, y, dx, dy];
4:     (x_B, y_B, ℓ_B) := graf[x − dx, y − dy, −dx, −dy];
5:     ℓ := ℓ_A + 1 + ℓ_B;
6:     graf[x_A, y_A, −dx, −dy] := (x_B, y_B, ℓ);
7:     graf[x_B, y_B, dx, dy] := (x_A, y_A, ℓ);
8: end
```

Przez każdy klocek przechodzą cztery pary krawędzi, które muszą być scalone, wobec tego przy usuwaniu klocka należy uwzględnić je wszystkie:

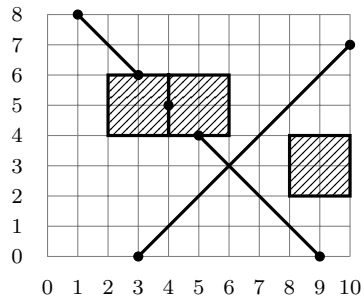
```
1: function UsunKlocek(x, y, dx, dy)
2: begin
3:     (sx, sy) := SrodekPola(x, y, dx, dy);
4:     klocek[sx, sy] := false;
5:     ScalKrawedz(sx + 1, sy, 1, 1);
6:     ScalKrawedz(sx, sy + 1, −1, 1);
7:     ScalKrawedz(sx − 1, sy, −1, −1);
8:     ScalKrawedz(sx, sy − 1, 1, −1);
9: end
```

W powyższym kodzie jest jeszcze jedna subtelność, na którą należy zwrócić uwagę. Powiedzieliśmy, że wierzchołki na brzegach klocka można usunąć z grafu, ale jest jeden wyjątek. Bowiem tuż po usunięciu klocka piłeczka znajduje się w jednym z wierzchołków, które były na brzegu klocka, więc potrzebujemy pamiętać jego wartość *graf*, gdyż będziemy z niej korzystać w następnym wywołaniu funkcji Ruszaj. Jednak to nie jest problem, gdyż nie musimy fizycznie usuwać tej wartości z tablicy.

Choć nowy graf jest dużo mniejszy, to nadal tworzymy go, przechodząc przez całą planszę, co zabiera czas $O(mn + k)$. Dużo szybsze jest jednak wyznaczanie kolejnego klocka do zbicia – jako że nadal potencjalnie możemy się odbić od wielu ścian, czas działania programu pomiędzy kolejnymi odbiciami od klocków oszacujemy przez $O(m+n)$. Zatem cały algorytm działa w czasie $O(mn+(m+n)k)$. Jego kod jest w pliku `arks2.cpp`. Pomimo poprawy czasu działania, program wciąż zalicza tylko pierwsze podzadanie. Widać, że faza, którą musimy przyspieszyć, to tworzenie grafu.

Przyspieszamy tworzenie grafu

Wyznaczenie zbioru wszystkich wierzchołków grafu jest proste – problem leży w szybkim wyznaczeniu krawędzi pomiędzy nimi. Zauważmy, że każda krawędź leży na prostej nachylonej pod kątem 45°, zatem jeśli dwa wierzchołki o współrzędnych (x_1, y_1),



Rys. 4: Plansza i pogrubione dwie przykładowe przekątne: dla $x - y = 3$ zawierająca punkty $(3, 0)$ i $(10, 7)$ oraz dla $x + y = 9$ zawierająca punkty $(1, 8)$, $(3, 6)$, $(4, 5)$, $(4, 5)$, $(5, 4)$ i $(9, 0)$. Utworzone z nich krawędzie to $graf[3, 0, 1, 1] = (10, 7, 7)$, $graf[1, 8, 1, -1] = (3, 6, 2)$, $graf[4, 5, 1, -1] = (4, 5, 0)$, $graf[5, 4, 1, -1] = (9, 0, 4)$ oraz ich symetryczne odpowiedniki.

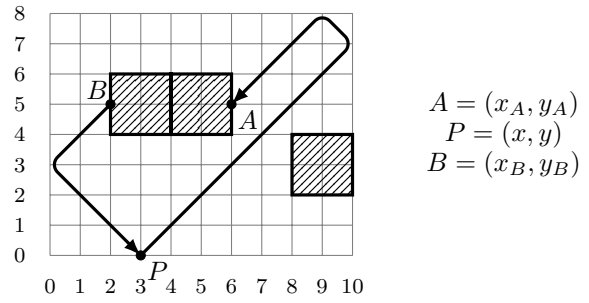
(x_2, y_2) są połączone krawędzią, to $x_1 - y_1 = x_2 - y_2$ (dla prostych nachylonych w prawo) lub $x_1 + y_1 = x_2 + y_2$ (dla prostych nachylonych w lewo). Ustalmy zatem pewną prostą, np. przechodzącą przez punkty (x, y), dla których $x + y = \text{const}$. Aby wyznaczyć krawędzie dla wierzchołków na tej prostej, wystarczy posortować je w kolejności występowania na prostej (czyli innymi słowy w kolejności współrzędnych x), a następnie połączyć krawędzią kolejne pary wierzchołków (bo pierwszy z nich będzie na dolnym lub lewym brzegu planszy, kolejne pary na bokach klocków, a ostatni na górnym lub prawym brzegu planszy). Przykład jest na rys. 4.

Zatem stworzenie grafu wymaga posortowania $O(m + n + k)$ par liczb, co możemy zrobić w czasie $O((m + n + k) \log(m + n))$. Warto też nadmienić, że już nie potrzebujemy kosztownej tablicy *klocek*; położenia klocków są jedynie wykorzystywane przy generowaniu wierzchołków. Zatem cały algorytm będzie działał w czasie $O((m + n + k) \log(m + n) + (m + n)k)$. Przykładowa implementacja jest w pliku `arks3.cpp`. Przechodzi ona dwa podzadania i uzyskuje połowę punktów za zadanie.

Przyspieszamy odbicia od brzegów planszy

Teraz w czasie stałym wyznaczamy położenie piłeczki w momencie kolejnego odbicia. Ale czasem możemy długo odbijać się od brzegów planszy, aby dostać się do klocka. Naturalna jest więc próba modyfikacji algorytmu, żeby w czasie stałym znajdować kolejne odbicie od klocka.

Rozważmy pewien wierzchołek naszego grafu, który odpowiada pozycji (x, y, dx, dy) na brzegu planszy. Istnieje dokładnie jedna krawędź wchodząca do tej pozycji oraz jedna krawędź wychodząca z pozycji otrzymanej po zastosowaniu funkcji Odbij (rys. 5). Zatem za każdym razem, gdy piłeczka będzie przechodzić przez tę pozycję, będzie to robić tymi dwiema krawędziami. Nic nie stoi zatem na przeszkodzie, abyśmy skleili te dwie krawędzie w jedną (oczywiście o wadze będącej sumą ich wag). Jeśli tak zrobimy dla wszystkich pozycji na brzegu planszy (poza pozycją początkową), to uzyskamy graf o $O(k)$ wierzchołkach i tylu krawędziach. Co



Rys. 5: Kompresja krawędzi przy brzegu planszy. Do pozycji $(3, 0, 1, -1)$ wchodzi krawędź z $(2, 5, -1, -1, 5)$, a z pozycji $(3, 0, 1, 1)$ wychodzi krawędź do $(6, 5, -1, -1, 11)$. Po kompresji będziemy mieli $graf[2, 5, -1, -1] = (6, 5, -1, -1, 16)$.

więcej, jeśli będziemy umieli poprawiać ten graf po usunięciu klocka, to znajdziemy rozwiązanie zadania po przejściu zaledwie k krawędzi.

Praktyczna realizacja tego pomysłu wymaga jednak pewnej uwagi. Po pierwsze, musimy dokonać drobnej zmiany w definicji grafu: ponieważ teraz pojedyncza krawędź może opisywać trasę piłeczki, która zawiera odbicia od ścian, nie mamy gwarancji, że wypadkowa pozycja piłeczki będzie miała ten sam kierunek. Zatem musimy go pamiętać w strukturze: $graf[x, y, dx, dy] = (nx, ny, ndx, ndy, \ell)$ będzie oznaczać, że piłeczka startująca z pozycji (x, y, dx, dy) przebędzie ℓ kroków, aby wykonać następne odbicie od klocka na pozycji (nx, ny, ndx, ndy). Stosowne zmiany w wartościach *graf* należy wprowadzić w funkcji generującej początkowy graf oraz w funkcji ScalKrawedz.

Poniższa funkcja do kompresji krawędzi jest bardzo podobna do funkcji ScalKrawedz:

```
1:  function KompresujKrawedz(x, y, dx, dy)
2:  begin
3:    (x_A, y_A, dx_A, dy_A, ℓ_A) := graf[x, y, dx, dy];
4:    (x_B, y_B, dx_B, dy_B, ℓ_B) := graf[x, y, −dy, dx];
5:    ℓ := ℓ_A + ℓ_B;
6:    graf[x_A, y_A, −dx_A, −dy_A] := (x_B, y_B, dx_B, dy_B, ℓ);
7:    graf[x_B, y_B, −dx_B, −dy_B] := (x_A, y_A, dx_A, dy_A, ℓ);
8:  end

9:  function KompresujGraf
10: begin
11:   for x := 1 to 2m − 1 step 2 do begin
12:     if x ≠ m then
13:       KompresujKrawedz(x, 0, 1, 1);
14:     KompresujKrawedz(x, 2n, −1, −1);
15:   end
16:   for y := 1 to 2n − 1 step 2 do begin
17:     KompresujKrawedz(0, y, 1, −1);
18:     KompresujKrawedz(2m, y, −1, 1);
19:   end
20: end
```

Niestety, dość łatwo przegapić jeszcze jedną zmianę, którą należy wprowadzić do funkcji UsunKlocek. Do tej pory cztery pary krawędzi przecinające klocek mogliśmy scalać niezależnie od siebie, gdyż na pewno były rozłączne. Teraz jednak, gdy w grafie nie ma wierzchołków na brzegach planszy, może wystąpić krawędź, która ma początek na jednym boku klocka, a koniec na innym (lub nawet tym samym) boku tego samego klocka. Wobec tego może się zdarzyć tak, że następny ruch będzie się odbywał po scalonej krawędzi, która przecina aktualnie usuwany klocek.

To rodzi potencjalny problem z usuwanym wierzchołkiem, którego będziemy używać w kolejnym wywołaniu funkcji Ruszaj. Musimy zagwarantować, że wartość *graf* dla tego wierzchołka nie zostanie zmieniona po scaleniu jego krawędzi. Na szczęście istnieje prosty sposób na poradzenie sobie z tym kłopotem: wystarczy że para krawędzi przecinająca klocek, do której należy ten wierzchołek, będzie uaktualniana *jako ostatnia*. Po szczegóły odsyłamy do implementacji.

Procedura zmniejszania grafu poprzez usuwanie wierzchołków na brzegach planszy i kompresję przechodzących przez nie krawędzi działa w czasie $O(m+n)$ i jest zdominowana przez procedurę tworzenia grafu w czasie $O((m+n+k)\log(m+n))$. Z kolei wyznaczanie odbić od kolejnych klocków działa w końcu w czasie stałym, zatem cała symulacja wykona się w czasie $O(k)$. Daje to ostateczną złożoność $O((m+n+k)\log(m+n))$. Takie rozwiązanie zapisano w pliku `ark.cpp` i zdobywa ono komplet punktów. Dla pełności zauważmy, że stosując sortowanie przez zliczanie, można jeszcze zmniejszyć tę złożoność do $O(m+n+k)$.

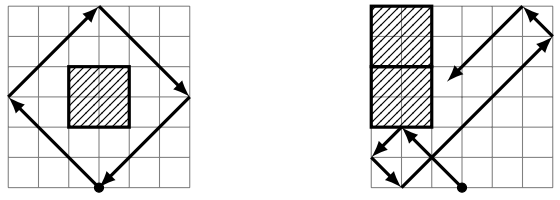
Dlaczego piłeczka zbije wszystkie klocki?

Pozostała do rozważenia jeszcze jedna kwestia. Otóż, do tej pory przyjmowaliśmy założenie, że piłce uda się zbić wszystkie klocki, tzn. że algorytm w pewnym momencie się zatrzyma. Nie jest jednak zupełnie oczywiste, dlaczego tak się stanie. Dla przykładu, dla planszy rozmiarów 3×3 z jednym klockiem na środku, piłeczka nigdy nie zbije tego klocka. Mogą też zdarzyć się układy klocków, w których jedynie część klocków zostanie zbita (rys. 6).

Jednakże wymiary tej planszy są niezgodne z założeniem z treści zadania, które mówi, że wymiary te muszą być liczbami względnie pierwszymi. W istocie dla planszy zgodnych z tym założeniem, piłeczka zawsze zbije wszystkie klocki. W tym rozdziale poczynimy pewne obserwacje, które zbliżą nas do odpowiedzi na pytanie, dlaczego tak musi być, ale do pełnego dowodu będziemy potrzebowali pewnego faktu, który udowodnimy na końcu opisu.

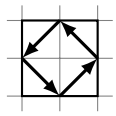
Rozważmy ruch piłeczki po pustej planszy. Piłeczka może poruszać się w nieskończoność, ale ponieważ plansza zawiera skończoną liczbę pozycji, to w pewnym momencie piłeczka drugi raz pojawi się w uprzednio odwiedzonej pozycji i dalej jej ruch będzie się powtarzał w cyklu.

Przeanalizujmy, które z pozycji (x, y, dx, dy) mogą się pojawić na trasie piłeczki. Ustalmy w tym celu pewne pole o środku (sx, sy) i załóżmy, że piłeczka dotyka boku tego pola od wewnętrznej strony. Z tego, co powiedzieliśmy już wcześniej, parzystości liczb x i y są różne, zatem są cztery możliwe położenia piłeczki na bokach tego pola: $(sx \pm 1, sy)$ oraz $(sx, sy \pm 1)$. Uwzględniając możliwe wartości dx i dy, otrzymujemy osiem pozycji piłeczki w obrębie pola. W zależności od ich skierowania podzielmy te pozycje na cztery lewoskrętne i cztery prawoskrętne (rys. 7).

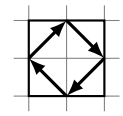


Rys. 6: Przykłady plansz 3×3, dla których piłeczka nie zbije wszystkich klocków.

$(sx, sy-1, 1, 1)$
$(sx+1, sy, -1, 1)$
$(sx, sy+1, -1, -1)$
$(sx-1, sy, 1, -1)$



$(sx, sy-1, -1, 1)$
$(sx+1, sy, -1, -1)$
$(sx, sy+1, 1, -1)$
$(sx-1, sy, 1, 1)$



Rys. 7: Lewo- i prawoskrętne pozycje dla pola (sx, sy).

Zauważmy, że jeśli po pojedynczym kroku z danej pozycji następuje odbicie (zatem pole sąsiaduje ze ścianą lub z klockiem), to nowa pozycja również znajduje się na tym samym polu, a jej skrętność zostaje zachowana. Z kolei, gdy odbicie nie następuje, czyli piłeczka przechodzi na pole sąsiadujące bokiem, to nowa pozycja (na nowym polu) ma przeciwną skrętność.

Pomalujmy teraz pola planszy w szachownicę (na czarno i na biało, przy czym każde dwa pola mające wspólny bok są pomalowane przeciwnymi kolorami). Załóżmy, że pole, z którego startuje piłeczka, jest białe. Wtedy widać, że pozycje piłeczki na każdym polu białym muszą być prawoskrętne, natomiast pozycje piłeczki na każdym polu czarnym muszą być lewoskrętne.

Wynika z tego ograniczenie górne na liczbę pozycji, które może przyjąć piłeczka podczas swojego ruchu: mamy mn pól, na każdym są cztery możliwe pozycje o odpowiedniej skrętności, zatem w sumie mamy co najwyżej $4mn$ pozycji.

Dla zupełnie dowolnych plansz nie wszystkie z tych $4mn$ pozycji będą mogły być osiągnięte. Dla lewej planszy z rys. 6 spośród 36 pozycji piłeczka przechodzi przez 12, wracając do pola startowego, natomiast dla prawej planszy po zbiciu klocka po dwóch ruchach piłeczka wpada w cykl składający się z 12 innych pozycji, nie powracając już nigdy do pozycji startowej.

W ogólności jest tak, że zbiór możliwych pozycji piłeczki rozpada się na kilka niezależnych cykli (między którymi możliwe są przeskoki w momencie odbicia piłeczki od klocków). Jednakże (jak udowodnimy nieco później), gdy liczby m i n są względnie pierwsze, to cykl jest zawsze jeden. Innymi słowy, w tym przypadku piłeczka na pustej planszy odwiedzi wszystkie $4mn$ pozycji, po czym wróci do punktu startowego. Z kolei w przypadku odbicia od klocka, jej pozycja zmienia się, ale nadal na którąś z pozycji z tego cyklu.

Alternatywny sposób znajdowania kolejnego odbicia od klocka

Obserwację, że wszystkie możliwe pozycje piłeczki leżą na jednym cyklu, wykorzystamy przy alternatywnym podejściu do rozwiązania naszego zadania. Pomysł jest taki, że każdej możliwej pozycji piłeczki przyporządkujemy unikalny numer, oznaczający liczbę jednostek czasu, po których pierwszy raz ta pozycja pojawia się na cyklu. Oznaczmy ten numer przez $\mathrm{Numer}(x, y, dx, dy)$.

Przykładowo dla planszy rozmiarów 5×4 (rys. 2) mamy $\mathrm{Numer}(5, 0, -1, 1) = 0$, $\mathrm{Numer}(4, 1, -1, 1) = 1$, $\mathrm{Numer}(0, 5, 1, 1) = 5$, $\mathrm{Numer}(7, 4, 1, -1) = 12$ i ostatecznie $\mathrm{Numer}(6, 1, -1, -1) = 79$.

Przyda nam się też funkcja odwrotna, która na podstawie numeru poda nam pozycję, tzn. $\mathrm{Pozycja}(t) = (x, y, dx, dy)$ jeśli $\mathrm{Numer}(x, y, dx, dy) = t$.

Najprostsza implementacja funkcji Numer i Pozycja będzie bezpośrednio korzystać z tablic *numery* i *pozycje*, które wyznaczymy, jednokrotnie przechodząc piłeczką po wszystkich pozycjach planszy:

```
1:  function InicjujNumery
2:  begin
3:    (x, y, dx, dy) := (m, 0, −1, 1);
4:    for t := 0 to 4·m·n − 1 do begin
5:      numery[x, y, dx, dy] := t;
6:      pozycje[t] := (x, y, dx, dy);
7:      x := x + dx;
8:      y := y + dy;
9:      if Sciana(x, y, dx, dy) then
10:       Odbij;
11:   end
12: end
```

Najbardziej nas będą interesować numery tych pozycji, które znajdują się na bokach klocków (czyli opisanych na rys. 7). Niech T oznacza zbiór tych numerów dla wszystkich klocków (zatem zbiór ten ma $4k$ elementów).

Będziemy znajdować kolejne odbicia od klocków. Załóżmy, że piłeczka znajduje się na pewnej pozycji (x, y, dx, dy), czyli że jest w odległości $t = \text{Numer}(x, y, dx, dy)$ od początku cyklu. Będzie teraz odwiedzać pozycje o kolejnych numerach, aż znajdzie pierwszy numer większy od t, należący do zbioru T (czyli w którym jest odbicie od klocka); oznaczmy ten numer przez t'. Jeśli zbiór T będziemy reprezentowali jako strukturę `set` z biblioteki standardowej C++, to taki element łatwo możemy znaleźć w czasie $O(\log k)$. Ponadto należy uwzględnić przypadek, gdy w T nie ma żadnego elementu większego od t. To oznacza, że piłeczka przejdzie przez pozycje początkową i cykl zacznie się od nowa; wtedy za t' przyjmujemy najmniejszy element zbioru T.

Na podstawie pozycji piłeczki Pozycja(t') wyznaczamy klocek, od którego nastąpi odbicie, i je wykonujemy. W końcu usuwamy ze zbioru T wszystkie numery, które odpowiadały temu klockowi.

```
1:  function Ruszaj
2:  begin
3:    t := Numer(x, y, dx, dy);
4:    if w zbiorze T istnieje element większy od t then begin
5:      t' := najmniejszy element z T większy od t;
6:      czas := czas + t' − t;
7:    end else begin
8:      t' := najmniejszy element z T;
9:      czas := czas + 4·m·n − (t − t');
10:   end
11:   (x, y, dx, dy) := Pozycja(t');
12:   UsunKlocek(x, y, dx, dy); { usuwa z T numery odpowiadające klockowi }
13:   k := k − 1;
14:   Odbij;
15: end
```

Czas działania tego algorytmu to $O(mn+k\log k)$, przy czym $O(mn)$ to koszt obliczeń wstępnych służących wyznaczaniu tablic *numery* i *pozycje*, natomiast $O(k\log k)$ to koszt symulacji ruchu piłeczki. Algorytm jest zapisany w pliku `arks4.cpp` i zalicza pierwsze podzadanie.

Przyspieszamy generowanie numerów

Dotychczas funkcje Numer i Pozycja były bardzo proste, ale kosztem tego, że podczas obliczeń wstępnych generowaliśmy numery po kolei dla wszystkich możliwych pozycji, pomimo tego, że podczas właściwej fazy algorytmu potrzebowaliśmy jedynie $O(k)$ pozycji, odpowiadających miejscom odbić od klocków.

Pomysł na przyspieszenie jest następujący: podczas obliczeń wstępnych przejdziemy cały cykl, ale będziemy zapamiętywać jedynie wartości funkcji Numer dla pozycji, w których piłeczka odbija się od ściany. Dzięki temu będziemy mogli przeskakiwać całe przekątne naraz.

W tym celu potrzebujemy funkcji, która dla danej pozycji piłeczki (x, y, dx, dy) obliczy, po ilu jednostkach czasu piłeczka odbije się od ściany. Można to zrobić następująco. Załóżmy, że chcemy wyznaczyć moment odbicia od prawego brzegu planszy. Kandydatem będzie taka wartość ℓ, dla której piłeczka dotknie prostej $x = 2m$:

$$x + \ell \cdot dx = 2m, \qquad \text{czyli} \qquad \ell = (2m - x)/dx.$$

Oczywiście, kandydat ten jest poprawny, jeśli $\ell \geqslant 0$ oraz nie ma żadnego mniejszego. Ponadto, w przypadku, gdy dostaniemy $\ell = 0$, musimy sprawdzić, czy piłeczka rzeczywiście porusza się w kierunku ściany (a nie oddala się od niej).

```
1: function IdzDoSciany(x, y, dx, dy)
2: begin
3:   minℓ := ∞;
4:   foreach ℓ in {−x/dx, −y/dy, (2m − x)/dx, (2n − y)/dy} do
5:     if ℓ ⩾ 0 and Sciana(x + ℓ · dx, y + ℓ · dy, dx, dy) then
6:       minℓ := min(minℓ, ℓ);
7:   return minℓ;
8: end
```

Mając taką funkcję, możemy wyznaczyć kolejność, w jakiej piłeczka będzie odwiedzać przekątne na cyklu. Dla każdej z tych przekątnych zapamiętujemy pierwszą pozycję na przekątnej oraz numer tej pozycji (służą do tego tablice *przekpoz* i *przeknum*). Ponadto dla pozycji, które są początkami przekątnych, zapamiętujemy numery tych przekątnych w tablicy *przek*.

```
1:  function InicjujNumery
2:  begin
3:    (x, y, dx, dy) := (m, 0, −1, 1);
4:    t := 0;
5:    for i := 0 to 2(m + n) − 1 do begin
6:      przek[x, y, dx, dy] := i;
7:      przekpoz[i] := (x, y, dx, dy);
8:      przeknum[i] := t;
9:      ℓ := IdzDoSciany(x, y, dx, dy);
10:     t := t + ℓ;
11:     x := x + ℓ · dx;
12:     y := y + ℓ · dy;
13:     Odbij;
14:   end
15: end
```

Teraz napisanie funkcji Numer jest już nietrudne. Na początek znajdujemy początek przekątnej, na której jest pozycja (x, y, dx, dy). W tym celu z punktu (x, y) cofamy się w kierunku $(-dx, -dy)$ do pierwszego odbicia ze ścianą. Wynik to suma numeru przypisanego początkowi przekątnej i liczby kroków, o które musieliśmy się cofnąć.

```
1: function Numer(x, y, dx, dy)
2: begin
3:   ℓ := IdzDoSciany(x, y, −dx, −dy);
4:   i := przek[x − ℓ · dx, y − ℓ · dy, dx, dy];
5:   return przeknum[i] + ℓ;
6: end
```

Funkcja odwrotna jest równie prosta. Znajdujemy wyszukiwaniem binarnym przekątną o największym numerze początku nie większym od t (niech to będzie i-ta przekątna o numerze początku t'). Łatwo to zrobić przy pomocy funkcji `upper_bound` z biblioteki standardowej C++ wywołanej na tablicy *przeknum*. Idziemy tą przekątną $t - t'$ kroków:

```
1: function Pozycja(t);
2: begin
3:   i := największa liczba, że przeknum[i] ⩽ t;
4:   (x, y, dx, dy) := przekpoz[i];
5:   ℓ := t − przeknum[i];
6:   return (x + ℓ · dx, y + ℓ · dy, dx, dy);
7: end
```

Ponieważ przekątnych jest $2(m+n)$, więc obliczenia wstępne zajmą czas $O(m+n)$. Funkcja Pozycja działa w czasie $O(\log(m + n))$ z uwagi na wyszukiwanie binarne. Ostatecznie algorytm zadziała w czasie $O(m + n + k \log(m + n + k))$. Jest zapisany w pliku `ark1.cpp`. Wystarczył do zaliczenia kompletu testów.

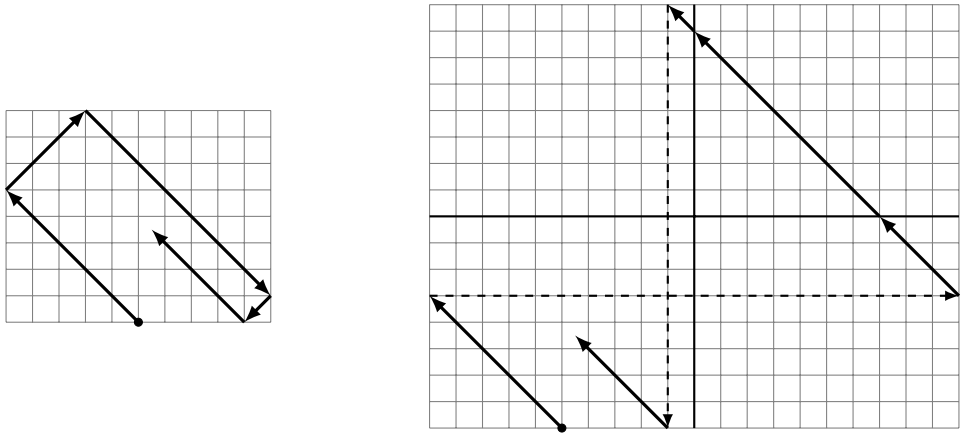
Jeszcze szybsza funkcja do numerów i dowód istnienia dokładnie jednego cyklu

Okazuje się, że istnieje jeszcze szybsze rozwiązanie zadania, które działa podliniowo od rozmiarów planszy. Wykorzystuje się w nim trik implementacyjny, który ma zastosowanie w przypadku wielu zadań, w których mamy piłeczkę odbijającą się od ścian prostokątnej planszy. Jedną z trudności w tego typu zadaniach jest konieczność rozważania różnych przypadków w zależności od kierunku poruszania się piłeczki. Trik jest następujący: zamiast odbijać piłeczkę, odbijamy planszę i zakładamy, że piłeczka porusza się zawsze w jednym kierunku. A konkretniej: zamiast planszy $2m \times 2n$ rozważamy czterokrotnie większą planszę $4m \times 4n$, na której piłeczka porusza się jak po torusie (tzn. dochodząc do brzegu planszy, pojawia się z przeciwległego brzegu; rys. 8).

Ruchowi piłeczki na oryginalnej planszy w kierunku $(-1, 1)$ odpowiada ruch piłeczki w dolnej lewej ćwiartce torusa. Po dotarciu do lewej ściany na oryginalnej planszy następuje odbicie i zmiana kierunku na $(1, 1)$, natomiast na torusie piłeczka przenosi się na prawą ścianę i kontynuuje ruch w kierunku $(-1, 1)$ w dolnej prawej ćwiartce. Analogicznie ruchom w kierunkach $(1, -1)$ i $(-1, -1)$ na oryginalnej planszy odpowiadają ruchy w górnej prawej i górnej lewej ćwiartce.

Poniższy wzór ustala odpowiedniość między pozycjami (x, y, dx, dy) na oryginalnej planszy a położeniami (X, Y) na torusie. Nietrudno napisać wzór, który będzie przeliczał współrzędne w drugą stronę.

$$(X, Y) = \begin{cases} (x, y) & \text{dla } (dx, dy) = (-1, 1), \\ (4m - x, y) & \text{dla } (dx, dy) = (1, 1), \\ (4m - x, 4n - y) & \text{dla } (dx, dy) = (1, -1), \\ (x, 4n - y) & \text{dla } (dx, dy) = (-1, -1). \end{cases}$$



Rys. 8: Oryginalna plansza i czterokrotnie większa plansza, na której piłeczka porusza się jak po torusie.

Ponieważ na torusie ruch odbywa się cały czas w jednym kierunku, a jego brzegi są „zawinięte", możemy w łatwy sposób opisać położenie piłeczki (X, Y) w dowolnej chwili t przy użyciu układu równań modularnych. A mianowicie:

$$X = (m - t) \bmod 4m, \qquad Y = t \bmod 4n. \tag{1}$$

Zauważmy, że z tego bardzo łatwo wynika sposób obliczania funkcji Pozycja(t). Wystarczy wyznaczyć położenie (X, Y) z równania (1) i zobaczyć, jaka pozycja odpowiada mu na oryginalnej planszy.

Wyznaczenie funkcji Numer(x, y, dx, dy) jest już trudniejsze, bo wymaga (po zamianie na współrzędne na torusie) rozwiązania układu kongruencji

$$t \equiv m - X \pmod{4m}, \qquad t \equiv Y \pmod{4n}. \tag{2}$$

W tym celu wykorzystamy Chińskie Twierdzenie o Resztach. Musimy jeszcze przezwyciężyć drobny kłopot polegający na tym, że twierdzenie to działa dla układów o względnie pierwszych modułach, a w naszym przypadku liczby $4m$ i $4n$ nie są względnie pierwsze. Jednakże w założeniach zadania jest, że liczby m i n są względnie pierwsze oraz m jest nieparzysta, zatem względnie pierwsze są liczby m i $4n$. Możemy zatem najpierw rozwiązać układ kongruencji

$$t' \equiv m - X \pmod{m}, \qquad t' \equiv Y \pmod{4n}, \tag{3}$$

w którym wyznaczymy t', takie że $t' \equiv t \pmod{4mn}$. Zatem t' jest też rozwiązaniem oryginalnego układu (2).

Układ kongruencji (3) możemy rozwiązać w czasie stałym, jeśli znamy dwie liczby całkowite α i β spełniające równanie

$$\alpha \cdot m + \beta \cdot 4n = 1.$$

Ponieważ m i $4n$ są względnie pierwsze, to liczby te istnieją i możemy je wyznaczyć w czasie $O(\log(m+n))$ za pomocą rozszerzonego algorytmu Euklidesa. Rozwiązaniem układu kongruencji (3) jest wtedy

$$t' \equiv (m - X \bmod m) \cdot \beta \cdot 4n + Y \cdot \alpha \cdot m \pmod{4mn}.$$

Zatem cały algorytm będzie działał w czasie $O(\log(m+n) + k \log k)$. Jego implementacja znajduje się w pliku `ark2.cpp`.

Pozostała nam do wykonania ostatnia obserwacja: z Chińskiego Twierdzenia o Resztach wiemy, że w układzie kongruencji (3) różnym wartościom $0 \leqslant t' < m \cdot 4n$ odpowiadają różne pary (X, Y). Pociąga to za sobą różnowartościowość rozwiązań układu (2) dla $0 \leqslant t < 4mn$. Wynika z tego, że każda z $4mn$ początkowych pozycji piłeczki będzie różna. To kończy dowód twierdzenia, że wszystkie $4mn$ pozycje piłeczki leżą na jednym cyklu.

Na koniec jeszcze jedna uwaga natury implementacyjnej. Otóż trik z odbijaniem planszy zamiast piłeczki można było zrobić na samym początku i przez całe rozwiązanie operować na współrzędnych (X, Y) i ich numerach. Niektóre z powyższych rozwiązań mogą stać się dzięki temu prostsze w implementacji (np. wszystkie przekątne będą miały ten sam kierunek), ale trzeba uważać na nowe komplikacje (np. funkcja Odbij będzie musiała „teleportować" piłeczkę na odpowiednią ćwiartkę torusa).

Tomasz Idziaszek	Tomasz Idziaszek, Paweł Parys	Karol Pokorski
Treść zadania	Opracowanie	Program

Dostępna pamięć: 128 MB. *OI, etap II, dzień drugi, 11.02.2016*

Wcale nie Nim

Bajtoni i jego młodszy braciszek Bajtuś często grają w grę Nim. Bajtoni objaśnił braciszkowi, jaka jest strategia wygrywająca w tej grze, ale Bajtuś jeszcze nie radzi sobie z jej stosowaniem, i często przegrywa. Z tego powodu co rusz proponuje zmiany w regułach gry, mając nadzieję, że rozgrywka będzie łatwiejsza.

Właśnie zaproponował nową wersję: mamy n par stosów, przy czym stosy z i-tej pary zawierają początkowo po a_i kamieni. Gracze wykonują ruchy na przemian. Bajtuś w swoim ruchu zabiera niezerową liczbę kamieni z dowolnego wybranego przez siebie stosu. Z kolei Bajtoni w swoim ruchu przekłada niezerową liczbę kamieni pomiędzy stosami w wybranej przez niego parze. Bajtuś wykonuje ruch jako pierwszy. Przegrywa ten, kto nie może wykonać już żadnego ruchu.

Bajtoni od razu zauważył, że przy takich zasadach nie ma żadnych szans na zwycięstwo, ale nie chcąc robić przykrości braciszkowi, zgodził się zagrać. Postawił sobie jednak za punkt honoru jak najdłużej odwlekać nieuchronną przegraną. Pomóż mu i napisz program, który stwierdzi, jak długo może potrwać rozgrywka, jeśli obaj bracia grają optymalnie (Bajtuś dąży do zwycięstwa w najmniejszej liczbie ruchów, zaś Bajtoni dąży do maksymalnego wydłużenia gry).

Wejście

W pierwszym wierszu standardowego wejścia znajduje się dodatnia liczba całkowita n oznaczająca liczbę par stosów. W drugim wierszu znajduje się ciąg n dodatnich liczb całkowitych $a_1, a_2, \ldots, a_n$ pooddzielanych pojedynczymi odstępami, oznaczających liczności kolejnych par stosów.

Wyjście

W jedynym wierszu standardowego wyjścia należy zapisać jedną liczbę całkowitą oznaczającą liczbę ruchów, po których nastąpi koniec gry, jeśli obaj bracia grają optymalnie.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
2 1 2	7

Wyjaśnienie do przykładu: *Optymalna rozgrywka może wyglądać następująco:*

1 1 2 2 → 1 1 2 0 → 1 1 1 1 → 1 1 1 0 → 1 1 0 1 → 1 1 0 0 → 2 0 0 0 → 0 0 0 0

Testy „ocen”:

`1ocen`: $n = 1$, $a_1 = 100$, *wynik 15*,

`2ocen`: $n = 5$, *wszystkie stosy po 2 kamienie, wynik 21*,

`3ocen`: $n = 3$, $a_1 = 10^7$, $a_2 = 10^8$, $a_3 = 10^9$, *wynik 163*,

`4ocen`: $n = 3000$, $a_i = i$, *wynik 65 197*,

`5ocen`: $n = 100\,000$, *wszystkie stosy po 1 kamieniu, wynik 200 001*.

Ocenianie

Zestaw testów dzieli się na podzadania spełniające następujące warunki. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów. We wszystkich podzadaniach zachodzi $a_i \leqslant 1\,000\,000\,000$.

Podzadanie	Warunki	Liczba punktów
1	$n = 1$	*10*
2	*suma wartości* $a_i \leqslant 10$	*10*
3	$n \leqslant 3$	*20*
4	$n \leqslant 3\,000$	*20*
5	$n \leqslant 500\,000$	*40*

Rozwiązanie

W zadaniu mamy do czynienia z grą dla dwóch graczy, a naszym celem jest ustalenie, jaka jest optymalna strategia gry dla każdego z nich. Jak zobaczymy później, samo zaprogramowanie wyliczania liczby ruchów, które zostaną wykonane przez graczy przy zastosowaniu tych strategii, będzie już proste.

Na początku opiszemy, mniej lub bardziej intuicyjny, sposób dochodzenia do rozwiązania; formalny dowód poprawności zaprezentowanych strategii zostanie przedstawiony w dalszej części.

Jak grać w grę?

Dla łatwiejszego rozróżniania graczy, nazwijmy gracza pierwszego A, a drugiego B. Gracz A chce jak najszybciej zakończyć grę, zatem chce minimalizować liczbę kamieni na stosach. Dość łatwo więc zgodzić się z następującą obserwacją:

Obserwacja 1. *Gdy gracz A ustali, z której pary stosów zabierze kamienie w następnym ruchu, to najbardziej opłaca mu się zabrać wszystkie kamienie z większego ze stosów z pary.*

Skoro wiemy już, że ruchy A powodują opróżnienie większego ze stosów z pary, to celem B powinno być zmaksymalizowanie liczby kamieni na mniejszym ze stosów w parze. Czynimy więc kolejną obserwację:

Obserwacja 2. *Gdy gracz B ustali, że w następnym ruchu będzie przekładał kamienie w pewnej parze stosów zawierającej a i b kamieni ($a + b = k$), to najbardziej opłaca mu się:*

- *wyrównać zawartość stosów do $(\frac{k}{2}, \frac{k}{2})$, gdy k jest parzyste i $a \neq b$,*
- *lekko odejść od równowagi, do $(\frac{k}{2} - 1, \frac{k}{2} + 1)$, gdy k jest parzyste i $a = b = \frac{k}{2}$,*
- *„prawie" wyrównać do $(\frac{k-1}{2}, \frac{k+1}{2})$, gdy k jest nieparzyste.*

Początkowo dla wszystkich par stosów łączna ilość kamieni na obu stosach z pary jest parzysta i kamienie te są równo rozłożone. Zatem nie ma powodu, aby B psuł na nich równowagę, bo może tylko stracić. Natomiast gdy A usuwa wszystkie kamienie z jakiegoś stosu, to B zapewne będzie chciał wyrównać układ na parze, do której należał ten stos, bo inaczej A w następnym ruchu zdejmie od razu wszystkie kamienie z drugiego stosu, zamiast robić to w wielu ruchach. Możemy więc przyjąć za prawdziwą (bez ścisłego dowodu) następującą obserwację:

Obserwacja 3. *O ile w parze stosów, z której ostatnio usuwał gracz A, są jeszcze jakieś kamienie, to graczowi B najbardziej opłaca się ruszać w tej parze.*

Zatem jeśli para, z której ostatnio usuwał gracz A, jest postaci $(k, 0)$ dla $k \neq 0$, to B wyrówna do $(\frac{k}{2}, \frac{k}{2})$ dla k parzystego lub „prawie" wyrówna do $(\frac{k-1}{2}, \frac{k+1}{2})$ dla k nieparzystego.

W przypadku, gdy A usunie ostatni kamień w parze stosów, gracz B nie może wykonać swojego ruchu w tej parze i musi popsuć równowagę w jakiejś innej parze. Jak ma wybrać tę parę, opiszemy nieco dalej.

Z powyższych obserwacji wynika też, że przed ruchem gracza A wszystkie pary stosów będą jednego z trzech rodzajów wymienionych w obserwacji 2. Istotnie, na początku gry wszystkie pary są wyrównane, a potem po każdym ruchu A, który psuje którąś parę, gracz B poprawia tę parę, a po każdym ruchu A opróżniającym parę, gracz B lekko odchodzi od równowagi w innej parze (jeśli wszystkie są wyrównane) lub robi ruch zamieniający kolejność stosów w pewnej „niewyrównanej" parze.

Przyjrzyjmy się bliżej, jakie ruchy A ma do dyspozycji. Załóżmy, że przed ruchem gracza A w parze stosów znajduje się $a + b = k$ kamieni. Zapiszmy w systemie dwójkowym łączną liczbę kamieni k i zobaczmy, jak zapis ten zmieni się po ruchu gracza A. Otóż, jeśli k było parzyste i była równowaga lub jeśli k było nieparzyste i była „prawie"-równowaga, to ruch A usuwa ostatnią cyfrę z liczby k, np.:

$$k = 44 = 101100_2 \quad \rightarrow \quad \tfrac{k}{2} = 22 = 10110_2,$$
$$k = 45 = 101101_2 \quad \rightarrow \quad \tfrac{k-1}{2} = 22 = 10110_2.$$

Natomiast jeśli k było parzyste i równowaga była zaburzona, to ruch A zamienia k na $\frac{k}{2} - 1$, czyli najpierw usuwa ostatnią cyfrę, po czym ostatnią jedynkę w zapisie zamienia na zero, a zera za nią zamienia na jedynki:

$$k = 44 = 101\underline{10}0_2 \quad \rightarrow \quad \tfrac{k}{2} - 1 = 21 = 101\underline{01}_2.$$

Nazwijmy ten pierwszy rodzaj ruchu gracza A ruchem *zwykłym*, a ten drugi ruchem *ulepszonym*. Z tego, co powiedzieliśmy o tym, jak wyglądają stosy przed ruchem

gracza A, wynika, że są to jedyne rodzaje ruchów, które może wykonać A. Ponadto, jeśli ruchy te nie czyszczą pary stosów, to następny ruch gracza B na tej parze przywróci na niej równowagę lub „prawie"-równowagę.

Zastanówmy się teraz, ile ruchów będzie potrzebował A, żeby wyczyścić ustaloną parę stosów, na której znajduje się $a+b=k$ kamieni. Jeśli A będzie wykonywał same ruchy zwykłe, to będzie musiał wykonać tyle ruchów, ile cyfr ma zapis dwójkowy liczby k. Oznaczmy tę wartość przez $\ell(k) = \lfloor \log_2 k \rfloor + 1$.

Przeanalizujmy, czy używanie ruchów ulepszonych umożliwia graczowi A szybsze wyczyszczenie pary stosów (pomińmy na razie fakt, że wykonanie ruchu ulepszonego nie zawsze jest możliwe). Zauważmy, że dla większości liczb k ruch ulepszony również powoduje usunięcie jednej cyfry z zapisu dwójkowego. Wyjątkiem jest na przykład $k = 2 = 10_2$, gdzie pojedynczy ruch ulepszony powoduje opróżnienie pary stosów, czym skraca zapis dwójkowy o *dwie* cyfry.

Wynika z tego, że dla dowolnej liczby kamieni k, która w zapisie binarnym zaczyna się od jedynki i zera, możliwe jest wyczyszczenie stosów, używając $\ell(k) - 1$ ruchów. Najpierw wykonujemy $\ell(k) - 2$ ruchów zwykłych, doprowadzając do 10_2, a potem jeden ruch ulepszony.

W przypadku liczb k, których zapis binarny zaczyna się od większej liczby jedynek, jest nieco trudniej. Dla przykładu dla $k = 6 = 110_2$ wykonanie ruchu zwykłego prowadzi do liczby nieparzystej $3 = 11_2$, co uniemożliwia użycie ruchu ulepszonego. W tym przypadku lepiej jest od razu zacząć od ruchu ulepszonego, który prowadzi do 10_2, a następnie wykonać drugi ruch ulepszony. W ogólności, jeśli zapis dwójkowy liczby k zaczyna się od j jedynek, po których następuje zero, a potem $\ell(k) - 1 - j$ dowolnych cyfr, to A może wyczyścić taką parę, wykonując najpierw $\ell(k)-1-j$ ruchów zwykłych, a następnie j ruchów ulepszonych; zatem w sumie wykonując $\ell(k) - 1$ ruchów.

Zatem A, chcąc opróżnić ustaloną parę stosów, mógłby poprawić swój wynik o jeden ruch, jeśli tylko mógłby zapewnić sobie wykonanie tylu ruchów ulepszonych, ile jest jedynek na początku zapisu dwójkowego liczby kamieni w tej parze. Nietrudna analiza pokazuje, że lepiej się nie da (nie można poprawić wyniku o więcej niż jeden ruch, a aby poprawić o jeden ruch, nie wystarczy wykonanie mniejszej liczby ruchów ulepszonych).

Musimy jednak uwzględnić fakt, że wykonanie przez A na parze stosów ruchu ulepszonego możliwe jest tylko, gdy para ta spełnia następujące dwa warunki:

- liczba kamieni w parze k w zapisie binarnym składa się z pewnej liczby jedynek, po której następuje zero, czyli $k = 2^m - 2$ dla $m \geqslant 2$; oznaczmy zbiór wszystkich takich liczb przez M;
- para stosów ma zaburzoną równowagę przez gracza B.

Gracz A łatwo może zapewnić, że zaburzanie równowagi przez gracza B będzie odbywać się tylko na takich parach stosów, dla których liczba kamieni jest już w zbiorze M. Wystarczy, że początkowo, wykonując tylko ruchy zwykłe (nie czyszcząc w międzyczasie żadnej pary stosów do końca) dla każdej pary stosów usunie z zapisu dwójkowego liczby kamieni wszystkie cyfry występujące za pierwszym zerem z lewej strony. Bez straty ogólności możemy zatem założyć, że liczba kamieni dla *każdej* pary jest w zbiorze M.

Przy powyższych założeniach odnośnie strategii graczy A i B, pozostaje im już niewielki wybór. Otóż gracz A może (musi) wybrać, dla której pary stosów rezygnuje z poprawiania wyniku o jeden i czyści tę parę ruchami zwykłymi. Gdy to zrobi, gracz B może (musi) wybrać, dla której pary stosów zaburzy równowagę, czyli pozwoli graczowi A na wykonanie ruchu ulepszonego, który kasuje *jedną* jedynkę z zapisu dwójkowego liczby kamieni. Jeśli to była akurat ostatnia jedynka, to kolejka powraca do gracza B, który znowu musi wybrać, dla której pary stosów zaburzy równowagę. Jeśli nie była to ostatnia jedynka, to znowu musi wybierać gracz A.

Do skompletowania strategii potrzebna jest jeszcze jedna obserwacja:

Obserwacja 4. *Załóżmy, że wszystkie liczby kamieni na parach stosów są ze zbioru M i jest na nich równowaga. Jeśli jest ruch gracza A, to wybiera on najliczniejszą parę i wykonuje na niej ruchy aż do opróżnienia. Jeśli jest ruch gracza B, to wybiera on najliczniejszą parę i zaburza na niej równowagę.*

Aby ściśle uzasadnić tę obserwację (jak również poprzednie), zredukujemy naszą grę (nazwijmy ją G) do pewnej nowej gry (nazwijmy ją H). W grze H będzie n stosów, które dla odmiany zawierają zapałki. Jeśli w grze G początkowa liczba kamieni na i-tej parze stosów w zapisie dwójkowym zaczyna się od j jedynek, to na i-tym stosie w grze H znajduje się początkowo j zapałek. Gracze ruszają się na zmianę, zaczyna gracz A. Gracz A może opróżnić dowolny niepusty stos. Gracz B może natomiast usunąć dowolną dodatnią liczbę zapałek z dowolnie wybranych stosów, pod warunkiem że:

- co najmniej jedna usuwana zapałka nie była ostatnią zapałką na stosie, lub
- ruch opróżnia wszystkie stosy.

Gra kończy się, gdy wszystkie stosy są puste. Celem gracza A jest wykonanie jak najmniejszej liczby ruchów (czyli celem B jest zapewnienie, że A będzie ruszał się jak najwięcej razy). Dotychczasowe rozumowanie sugeruje, że prawdziwy powinien być następujący lemat:

Lemat 1. *Niech s będzie łączna długością (liczbą cyfr) zapisów dwójkowych liczb a_i. Przy optymalnej grze obu graczy w obu grach, liczba ruchów gracza A w G jest o s większa od liczby ruchów gracza A w grze H.*

Zanim udowodnimy ten lemat, powiedzmy, jak rozwiązać grę H. Po pierwsze, oczywiste jest, że gracz B będzie usuwał tylko jedną zapałkę na raz (za wyjątkiem ostatniego ruchu, kiedy to musi opróżnić wszystkie stosy mające po jednej zapałce); możliwość usuwania jeszcze jakichś dodatkowych zapałek nic mu nie daje, skoro chce usunąć ich jak najmniej, zostawiając jak najwięcej graczowi A. Dalej, dość łatwo przekonać się, że graczowi A zawsze najbardziej opłaca się opróżnienie największego stosu (bo gdy zostaną małe stosy, to B, czyszczący stosy po jednej zapałce, zdąży ich więcej opróżnić, zastępując w tym gracza A, który chce minimalizować liczbę swoich ruchów). Jeśli to wiemy, to widzimy także, że graczowi B również najbardziej opłaca się zabranie zapałki z najliczniejszego stosu (bo ten stos i tak zostanie szybko opróżniony przez A, natomiast zapałki z niższych stosów gracz B może usuwać później).

Rozwiązanie zadania wygląda więc następująco. Najpierw konstruujemy grę H (czyli liczymy długości liczb a_i w zapisie dwójkowym oraz liczbę jedynek na początku

każdego zapisu). Następnie gramy w H, wykonując opisane powyżej optymalne strategie obu graczy; liczymy przy tym, ile ruchów wykona gracz A. Możemy po prostu grać ruch po ruchu, bo ruchów jest niewiele (w zasadzie równie dobrze można wykonywać ruchy od razu w G, symulując optymalne strategie obu graczy). Do wyznaczania najliczniejszego stosu w H możemy skorzystać z kolejki priorytetowej (za każdym razem wyjmujemy maksimum i wstawiamy wynik ruchu).

Ponieważ w grze H gracz A wykona co najwyżej n ruchów, a każdy ruch gracza wymaga czasu $O(\log n)$ na kolejce priorytetowej, to powyższe rozwiązanie ma złożoność czasową $O(n \log n)$.

Wszędzie tu pisaliśmy o liczbie ruchów gracza A, natomiast w zadaniu mamy wypisać łączną liczbę ruchów obu graczy. Łatwo jednak zauważyć, że między tymi wielkościami zachodzi następująca zależność: jeśli gracz A ruszał się x razy, to obaj gracze ruszali się $2x - 1$ razy.

Opisane powyżej rozwiązanie zostało zaimplementowane w pliku `wca.cpp`. Alternatywna implementacja w plikach `wca1.cpp`, `wca2.pas` nie używa kolejki priorytetowej. Zamiast tego dla każdego j pamięta, ile liczb zaczyna się od j jedynek (różnych j jest mało, bo co najwyżej $\log_2 a_i$). W takiej tablicy łatwo znajdujemy, jaka jest maksymalna liczba jedynek w jakiejś liczbie; możemy też tę liczbę usunąć, czy zmniejszyć jej liczbę jedynek o jeden.

Odpowiedniość między grami G i H

W tej sekcji udowodnimy lemat 1, czyli formalnie pokażemy odpowiedniość między oryginalną grą G, a grą H.

Lemat 1 (**implikacja w jedną stronę**). *Jeśli A może zakończyć grę H, wykonując x ruchów, to może zakończyć grę G, wykonując $x + s$ ruchów.*

Dowód: Zagrajmy równocześnie w obie gry. Ruchy gracza A w G będziemy konstruować, patrząc na to, co dzieje się w H, a ruchy gracza B w H będziemy konstruować, patrząc na to, co dzieje się w grze G. Gracz A, grając w G oznacza sobie pary stosów jedną z liter X, Z. Początkowo wszystkie stosy oznaczone są literą X.

Będziemy utrzymywać następujący niezmiennik:

- Jeśli i-ta para stosów w G jest oznaczona literą X, to zapis dwójkowy łącznej liczby kamieni na stosach z tej pary zawiera zero, a liczba jedynek na początku tego zapisu jest równa liczbie zapałek na i-tym stosie w H.
- Jeśli i-ta para stosów w G jest oznaczona literą Z, to i-ty stos w H jest pusty.

Widzimy, że niezmiennik jest spełniony, gdy obie gry są w sytuacji początkowej (w szczególności łączna liczba kamieni na obu stosach w parze jest parzysta, czyli jej zapis dwójkowy zawiera zero).

Symulacja przebiega następująco:

1. Na samym początku pozwalamy graczowi A ruszyć się w H; usuwa on wszystkie zapałki ze stosu numer i, a my oznaczamy i-tą parę stosów w G literą Z. Niezmiennik pozostaje spełniony: napisaliśmy Z, a w H stos jest pusty.

2. Niech k_i to liczba kamieni na i-tej parze stosów w G (dla każdego i). Gracz A rusza się w G w następujący sposób:

   (a) Znajduje (o ile istnieje) takie i, że i-ta para jest oznaczona przez X oraz $k_i \notin M \cup \{0\}$ i usuwa z i-tej pary $\lceil \frac{k_i}{2} \rceil$ kamieni (na większym ze stosów jest co najmniej tyle). Ponieważ zapis dwójkowy k_i zawiera zero i $k_i \notin M$ (czyli ostatnia cyfra nie jest jedynym zerem), to po usunięciu ostatniej cyfry nadal będzie jakieś zero, a liczba jedynek na początku nie zmienia się; zatem niezmiennik pozostaje prawdziwy.

   (b) Jeśli nie było i jak wyżej, znajduje (o ile istnieje) takie i, że i-ta para jest oznaczona przez X, $k_i \in M$ oraz kamienie nie są rozłożone po równo na stosach; usuwa z i-tej pary $\frac{k_i}{2} + 1$ kamieni (na większym ze stosów jest co najmniej tyle). Jednocześnie w H gracz B usuwa jedną zapałkę z i-tego stosu (ruchy gracza B w H opisujemy zapałka po zapałce, wiele takich ruchów składa się na jeden pełny ruch). Opisany ruch A w G powoduje usunięcie jednej jedynki w zapisie dwójkowym liczby kamieni w i-tej parze stosów, czyli liczba jedynek na początku tego zapisu spada o 1. Jednocześnie usunęliśmy jedną zapałkę z i-tego stosu w H, więc niezmiennik pozostaje zachowany (a niezmiennik przed ruchem zapewnia, że rzeczywiście na i-tym stosie w H była jakaś zapałka i B może ją usunąć).

   (c) Jeśli nie było i jak wyżej, znajduje (o ile istnieje) takie i, że i-ta para jest oznaczona przez Z oraz $k_i > 0$ i usuwa z i-tej pary $\lceil \frac{k_i}{2} \rceil$ kamieni (na większym ze stosów jest co najmniej tyle). Niezmiennik jest zachowany w trywialny sposób.

   (d) Jeśli nie było i jak wyżej, to kończymy ruch gracza B w H i pozwalamy ruszyć się graczowi A (udowodnimy później, że rzeczywiście uzyskaliśmy poprawny ruch gracza B oraz że gra H jeszcze się nie skończyła); on usuwa wszystkie zapałki ze stosu numer i, a my oznaczamy i-tą parę stosów literą Z. Niezmiennik pozostaje spełniony: napisaliśmy Z, a w H stos jest pusty. Zwróćmy uwagę, że i-ta para stosów była wcześniej oznaczona literą X (gdyby była oznaczona przez Z, to i-ty stos w H byłby pusty) oraz jest tam jakiś kamień (liczba jedynek na początku zapisu dwójkowego liczby kamieni była równa liczbie zapałek na i-tym stosie, która była niezerowa). Następnie próbujemy ponownie wykonać punkt (c), w którym teraz już na pewno szukanie i zakończy się sukcesem.

3. Jeśli wszystkie stosy w G są puste, to gra się kończy. W tej sytuacji niezmiennik zapewnia, że również w H wszystkie stosy są puste.

4. Pozwalamy ruszyć się graczowi B w G i wracamy do punktu 2. Zauważmy, że niezależnie od jego ruchu, niezmiennik nadal jest spełniony.

Musimy teraz udowodnić, że w punkcie 2(d) możemy rzeczywiście zakończyć w H ruch gracza B i czekać na ruch gracza A. Ruch gracza B trwał od ostatniego momentu, gdy jakąś niepustą parę stosów oznaczyliśmy przez Z (w punkcie 1 lub 2(d)); zatem na początku tego ruchu istniała niepusta para oznaczona przez Z (dokładnie jedna, ale to dla nas bez znaczenia), a teraz nie istnieje (skoro jej znalezienie w 2(c) się nie

powiodło i przeszliśmy do 2(d)). Spójrzmy na moment, gdy ostatni kamień z tej pary stosów został usunięty przez A. Stało się to w punkcie 2(c), czyli nie udało się wówczas znalezienie i ani w punkcie 2(a), ani w punkcie 2(b). Oznacza to, że dla każdej pary stosów i oznaczonej przez X zachodziło $k_i \in M \cup \{0\}$ i na obu stosach tej pary było po tyle samo kamieni. Po rozważanym ruchu gracza A z punktu 2(c), usuwającym ostatni kamień z pary stosów oznaczonej Z, gracz B musi wykonać jakiś ruch i jedyne, co może zrobić, to zaburzyć równowagę w pewnej parze stosów i oznaczonej X, dla której $k_i \in M \cup \{0\}$. W kolejnym kroku gracz A na pewno zrobi ruch według punktu 2(b), dla tego i, zdejmując graczem B jedną zapałkę w grze H. Jeśli było $k_i > 2$, to zapałka usunięta w H nie była ostatnią na stosie, co zapewnia poprawność ruchu gracza B w H. Jeśli jednak $k_i = 2$, to A usuwa oba kamienie z tego stosu i B musi ruszyć się gdzie indziej, czyli znowu jedyne co może zrobić to zaburzyć równowagę w pewnej parze stosów i oznaczonej X, dla której $k_i \in M \cup \{0\}$; sytuacja się powtarza. Jedyny sposób wyjścia z tej pętli to albo przypadek $k_i > 2$, albo koniec gry. Skoro jednak ileś ruchów później doszliśmy do punktu 2(d), to musieliśmy przejść przez przypadek $k_i > 2$, czyli zapewnić poprawność ruchu gracza B w H. To kończy dowód poprawności ruchu gracza B w H. Musimy jeszcze zobaczyć, że H jeszcze się nie skończyła i jest sens czekać w niej na ruch gracza A. Skoro jednak G się nie skończyła, a jednocześnie nie ma tam niepustej pary stosów oznaczonej przez Z, to jest niepusta para stosów oznaczona przez X; niezmiennik zapewnia, że odpowiadający jej stos w H także jest niepusty.

Pozostaje porównać liczbę ruchów gracza A w obu grach. Niech s_i będzie liczbą cyfr w zapisie dwójkowym liczby a_i. Widzimy, że dla każdego i zachodzą następujące własności:

1. Jeśli na koniec i-ta para stosów jest oznaczona przez X, to aby ją opróżnić A wykonał s_i ruchów. Istotnie, początkowo na tej parze stosów mamy $2a_i$ kamieni (zapis dwójkowy długości $s_i + 1$, na pewno zawiera jakieś zero). Dopóki liczba kamieni nie jest w M, każdy ruch A dotyczący tej pary (z punktu 2(a)) usuwa ostatnią cyfrę z zapisu dwójkowego, skracając go o jeden. W pewnym momencie na pewno liczba kamieni zacznie być w M (gdy usuniemy wszystkie cyfry za pierwszym zerem). Wówczas każdy ruch A (z punktu 2(b)) usuwa jedną jedynkę z zapisu dwójkowego, w szczególności ostatni ruch przechodzi od $2 = 10_2$ kamieni od razu do 0 kamieni, skracając zapis dwójkowy o dwie cyfry. Ruchów A musiało więc być rzeczywiście s_i.

2. Jeśli na koniec i-ta para stosów jest oznaczona przez Z, to aby ją opróżnić A wykonał $s_i + 1$ ruchów. Istotnie, w tej sytuacji początek gry przebiega jak poprzednio, czyli każdy ruch A skraca zapis dwójkowy łącznej liczby kamieni o jedną cyfrę. W pewnym momencie (gdy jest jeszcze niepusta) para zostaje oznaczona przez Z. Później każdy ruch A na tej parze (z punktu 2(c)) powoduje usunięcie ostatniej cyfry z zapisu dwójkowego łącznej liczby kamieni, czyli skrócenie tego zapisu o jedną cyfrę. Skrócenie liczby $(s_i + 1)$-cyfrowej do 0-cyfrowej zajmuje więc rzeczywiście $s_i + 1$ ruchów.

Jednocześnie widzimy, że oznaczenie pary stosów przez Z występuje dokładnie wtedy, gdy gracz A rusza się w H. Zatem jeśli A w grze H wykonał x ruchów, to w G wykonał $x + s$ ruchów. ■

Lemat 1 (implikacja w drugą stronę). *Jeśli A może zakończyć grę G, wykonując $x + s$ ruchów, to może zakończyć grę H, wykonując x ruchów.*

Dowód: Znowu gramy równocześnie w obie gry. Ruchy gracza A w H będziemy konstruować, patrząc na to, co dzieje się w G, a ruchy gracza B w G będziemy konstruować, patrząc na to, co dzieje się w H. Z każdą parą stosów w grze G skojarzymy dwie liczby: r_i oraz z_i. Liczba r_i początkowo będzie równa długości zapisu dwójkowego liczby $2a_i$ (czyli łącznej liczby kamieni na i-tej parze stosów w G), natomiast z_i będzie równa liczbie jedynek na początku $2a_i$ (czyli liczbie zapałek na i-tym stosie w H). Dla dowolnych liczb naturalnych r, z zdefiniujmy:

$$m(r,z) = \begin{cases} \lfloor (2^z - 1) \cdot 2^{r-z} \rfloor & \text{gdy } z > 0, \\ \lceil 2^{r-1} - 1 \rceil & \text{gdy } z = 0. \end{cases}$$

Zatem gdy $z > 0$, liczba ta w zapisie dwójkowym jest długości r i zaczyna się od $\min(r, z)$ jedynek, po których są same zera. Jeśli zaś $z = 0$, to jest to po prostu $\max(0, r - 1)$ jedynek. Początkowe wyrazy tego ciągu to:

$$\begin{array}{lllll} m(0,0) = 0, & m(1,0) = 0, & m(2,0) = 1, & m(3,0) = 3, & m(4,0) = 7, \\ m(0,1) = 0, & m(1,1) = 1, & m(2,1) = 2, & m(3,1) = 4, & m(4,1) = 8, \\ m(0,2) = 0, & m(1,2) = 1, & m(2,2) = 3, & m(3,2) = 6, & m(4,2) = 12. \end{array}$$

Zauważmy, że zawsze $\lfloor \frac{m(r+1,z)}{2} \rfloor = m(r,z)$ (zmniejszenie r o jeden powoduje podzielenie liczby $m(r,z)$ przez dwa, z zaokrągleniem w dół).

Przed każdym ruchem gracza A w G prawdziwy będzie następujący niezmiennik (dla każdego i):

1. na i-tej parze stosów w G jest co najmniej $m(r_i, z_i)$ kamieni, przy czym na każdym ze stosów w parze co najmniej $\lfloor \frac{m(r_i,z_i)}{2} \rfloor$ kamieni,
2. na i-tym stosie w H jest co najwyżej z_i zapałek,
3. jeśli $r_i = 0$ to $z_i \leqslant 1$, oraz
4. $z_i = 0$ wtedy i tylko wtedy, gdy gra H już się skończyła.

Widzimy, że niezmiennik jest spełniony, gdy obie gry są w sytuacji początkowej (w szczególności zarówno $m(r_i, z_i)$ jak i liczba kamieni na i-tej parze stosów jest długości r_i i zaczyna się od z_i jedynek, przy czym w $m(r_i, z_i)$ po tych jedynkach są same zera, co daje pierwszy punkt).

Podczas symulacji powtarzamy następujące operacje:

1. Czekamy w G na ruch gracza A, który usuwa kamienie z pary stosów numer i. Może to zaburzyć pierwszy punkt niezmiennika, natomiast ciągle wiemy, że na jednym ze stosów w tej parze (na tym, którego A nie ruszał) jest co najmniej $\lfloor \frac{m(r_i,z_i)}{2} \rfloor$ kamieni.
2. Jeśli na którymś ze stosów w i-tej parze pozostało co najmniej $m(r_i, z_i)$ kamieni, to nic nie robimy, a jeśli nie (wtedy w szczególności na pewno jest

$r_i > 0$), to zmniejszamy r_i o 1. Po takim zmniejszeniu już na pewno na którymś ze stosów w i-tej parze pozostało co najmniej $m(r_i, z_i)$ kamieni, bo $\lfloor \frac{m(r_i+1,z_i)}{2} \rfloor = m(r_i, z_i)$; w szczególności punkt 1 niezmiennika jest już prawdziwy. Zauważmy, że może to zaburzyć trzeci punkt niezmiennika, ale tylko, gdy oba stosy stały się puste (jeśli któryś stos był niepusty, a r_i było 1, to byśmy nie zmniejszyli r_i, bo $m(1, z_i) \leqslant 1$).

3. Załóżmy, że po ruchu gracza A na parze stosów numer i pozostały jeszcze jakieś kamienie. Wówczas gracz B odpowiada na tym samym stosie. Jeśli $m(r_i, z_i) = 0$, to po prostu bierze dowolny kamień i przekłada; niezmiennik w trywialny sposób pozostaje prawdziwy. Jeśli $m(r_i, z_i) \geqslant 1$, to gracz B przekłada $\lceil \frac{m(r_i,z_i)}{2} \rceil$ kamieni (to jest $\geqslant 1$) ze stosu, na którym jest co najmniej $m(r_i, z_i)$ kamieni, na drugi stos, zapewniając prawdziwość niezmiennika. W tym przypadku to już koniec rozważań, wracamy do punktu 1, czekając na kolejny ruch gracza A.

4. Teraz rozważamy już tylko sytuację, że po ruchu gracza A na parze stosów numer i nie ma już kamieni. Skoro na którymś ze stosów w tej parze jest co najmniej $m(r_i, z_i)$ kamieni, to $m(r_i, z_i) = 0$ (czyli jeśli $z_i = 0$ to $r_i \leqslant 1$, a jeśli $z_i > 0$, to $r_i = 0$). Mamy trzy podprzypadki:

   (a) Być może gra H już się skończyła. Wtedy także punkt 3 niezmiennika jest prawdziwy (bo $z_i = 0$). Jeśli także gra G już się skończyła, to po prostu kończymy symulację. Załóżmy, że G jeszcze trwa i wybierzmy dowolną niepustą parę stosów j. Gracz B przekłada tam jeden kamień ze stosu nie mniejszego na nie większy (czyli z większego na mniejszy lub między równymi). Niezmiennik zapewnia, że $z_j = 0$, czyli $m(r_j, z_j)$ jest nieparzyste lub jest zerem. Jeśli $m(r_j, z_j) = 0$, to niezmiennik oczywiście pozostaje spełniony. Jeśli $m(r_j, z_j) > 0$, to z niezmiennika wiemy, że na większym ze stosów musiało być co najmniej $\lfloor \frac{m(r_j,z_j)}{2} \rfloor + 1$ kamieni, po ruchu na każdym stosie pozostaje co najmniej $\lfloor \frac{m(r_j,z_j)}{2} \rfloor$ kamieni. Wracamy do punktu 1.

   (b) Przypuśćmy jednak teraz, że H jeszcze się nie skończyła. Wówczas $z_i > 0$, czyli $r_i = 0$. Gracz A wykonuje ruch w H: jeśli i-ty stos jest niepusty, to opróżnia i-ty, a jeśli pusty, to opróżnia dowolny niepusty stos. Swobodnie (tzn. nie martwiąc się o prawdziwość punktu 2 niezmiennika) możemy teraz zmniejszyć z_i do 1, przywracając prawdziwość niezmiennika (w szczególności punktu 3). Jeśli ten ruch nie kończy gry H, to gracz B odpowiada, usuwając pewną liczbę zapałek. Jeśli nie kończy przy tym gry H, to z definicji gry H gracz B usuwa przynajmniej jedną zapałkę z jakiegoś stosu j zawierającego co najmniej 2 zapałki. Załóżmy, że rzeczywiście gra H nadal się nie skończyła. Wtedy $z_j \geqslant 2$ (z punktu 2 niezmiennika) oraz $r_j \geqslant 1$ (z punktu 3 niezmiennika), co daje $m(r_j, z_j) \geqslant 1$. Zmniejszamy z_j o jeden; ponieważ usunęliśmy zapałkę, to nadal zapałek na j-tym stosie w H będzie nie więcej niż z_j (niezmiennik pozostaje prawdziwy). Jeśli $m(r_j, z_j)$ jest nieparzyste, to na j-tej parze stosów gracz B po prostu przekłada jeden kamień z nie mniejszego stosu na nie większy i niezmiennik (w szczególności punkt 1) pozostanie prawdziwy. Jeśli zaś $m(r_j, z_j)$ jest parzyste, to jest ściśle mniejsze niż $m(r_j, z_j + 1)$ (wartość przed zmniejszeniem z_j), więc

także gracz B po prostu przekłada jeden kamień z nie mniejszego stosu na nie większy i niezmiennik pozostanie prawdziwy. Wracamy do punktu 1.

(c) Pozostaje do rozważenia sytuacja, gdy podczas wykonywania punktu 4(b) gra H się skończyła (albo od razu po wykonaniu w niej ruchu gracza A, albo dopiero po ruchu gracza B). Wtedy zmieniamy wszystkie z_k na 0, co zapewnia poprawność punktu 4 niezmiennika dla każdego k. Jednocześnie pozostałe punkty niezmiennika pozostają prawdziwe. Następnie wracamy do punktu 4(a).

Nasza symulacja przebiega aż do momentu, gdy gra G się skończy. Widzimy, że może to nastąpić jedynie w punkcie 4(a), czyli już po tym, gdy gra H się skończyła. Pozostaje porównać liczbę ruchów gracza A w obu grach. Niech s_i będzie liczbą cyfr w zapisie dwójkowym liczby a_i. Początkowo r_i jest równe s_i+1. Liczbę r_i zmniejszamy jedynie w punkcie 2, po tym jak A zrobi ruch w G. Na koniec musi być $m(r_i, z_i) = 0$ (punkt 1 niezmiennika), czyli $r_i \leqslant 1$. Zmniejszenie wszystkich r_i do 1 wymaga więc co najmniej $\sum_i s_i = s$ ruchów A w G. Zobaczmy też, że A rusza się w H jedynie w punkcie 4(b), czyli wtedy, gdy $r_i = 0$; zatem gracz A, aby ruszyć się w H, musi także zrobić dodatkowy ruch w G. Czyli jeśli w H wykonał x ruchów, to w G musiał wykonać przynajmniej $x + s$ ruchów. ■

Optymalne strategie w grze H

Pozostaje nam udowodnienie, jak wyglądają optymalne strategie w grze H. Zaczniemy od gracza A, ale wcześniej udowodnimy lemat pomocniczy.

Lemat 2. *Rozważmy dwie sytuacje C i D w grze H, przy czym D powstaje z C przez usunięcie pewnych zapałek z pewnych stosów. Załóżmy, że z sytuacji C gracz A może zakończyć grę, wykonując co najwyżej x ruchów. Wówczas także z sytuacji D gracz A może zakończyć grę, wykonując co najwyżej x ruchów.*

Dowód: Jeśli w D jest koniec gry, to teza zachodzi w sposób trywialny.

Przyjrzyjmy się najpierw przypadkowi, że w C i D jest kolej gracza B. Zakładając, że w D jest ruch gracza B, który wymusza więcej niż x ruchów gracza A, rozważmy ten właśnie ruch gracza B. Prowadzi on do sytuacji D', z której A musi wykonać w najgorszym razie więcej niż x ruchów. Ale z C również istnieje ruch gracza B, który prowadzi do D': najpierw usuwamy pewne zapałki sprowadzając sytuację do D, a potem ruszamy się jak w D. To jest sprzeczne z założeniem, że z C gracz A może zakończyć grę, wykonując co najwyżej x ruchów.

Rozważmy teraz przypadek, że w C i D jest kolej gracza A. Weźmy ruch gracza A z sytuacji C, który zapewnia zakończenie gry w co najwyżej x ruchach. Prowadzi on do sytuacji C', powstającej z C przez opróżnienie stosu o pewnym numerze i; z C' gracz A może zakończyć grę, wykonując co najwyżej $x-1$ ruchów. Jeśli w D stos i-ty jest niepusty, to rozważamy ruch gracza A opróżniający stos i-ty; jeśli zaś jest pusty, to opróżniamy dowolny niepusty stos. W obu tych przypadkach sytuacja D' powstała po tym ruchu powstaje z C' poprzez usunięcie pewnych zapałek z pewnych stosów. Z przypadku pierwszego dostajemy, że z D' gracz A może zakończyć grę, wykonując

co najwyżej $x-1$ ruchów. Zatem z D gracz A może zakończyć grę, wykonując co najwyżej x ruchów. ■

Lemat 3. *Rozważmy sytuację w grze H, z której A może zakończyć grę, wykonując co najwyżej x ruchów. Wówczas, jeśli A będzie cały czas opróżniał stos zawierający najwięcej zapałek, to także zakończy grę, wykonując co najwyżej x ruchów.*

Dowód: Dowód jest przez indukcję po x. Jeśli jest akurat kolej na gracza B, to wykonujemy jego ruch, sprowadzając do sytuacji, w której jest kolej gracza A. Załóżmy, że jest kolej gracza A. Jeśli w optymalnej strategii gracz A także opróżnia stos zawierający najwięcej zapałek (strategie różnią się później), to korzystamy z założenia indukcyjnego dla $x-1$. Załóżmy, że optymalna strategia opróżnia stos numer i, prowadząc do sytuacji C, natomiast opróżnienie stosu numer j, mającego najwięcej zapałek, prowadzi do sytuacji D. Zamieniając numerami stosy numer i i j w D, możemy założyć, że D różni się od C tylko tym, że w C na j-tym stosie jest więcej zapałek niż w D. Zatem C i D podpadają pod lemat 2, który mówi, że z D gracz A może zakończyć grę, wykonując co najwyżej $x-1$ ruchów (bo z C gracz A może zakończyć grę, wykonując co najwyżej $x-1$ ruchów). ■

Następnie udowodnimy, że także gracz B powinien zawsze zabierać jedną zapałkę ze stosu zawierającego najwięcej zapałek (chyba że na każdym stosie jest już tylko co najwyżej jedna zapałka, to wtedy B musi usunąć wszystkie). Dowód jest podzielony na trzy lematy.

Lemat 4. *Rozważmy sytuację w grze H, z której jest ruch gracza B, przy czym istnieje stos zawierający więcej niż jedną zapałkę. Załóżmy, że B może wymusić, że A wykona więcej niż x ruchów do końca gry. Wówczas istnieje taki ruch B, który usuwa tylko jedną zapałkę i po którym nadal B może wymusić, że A wykona więcej niż x ruchów do końca gry.*

Dowód: Rozważmy ruch gracza B w strategii, która wymusza, że A wykona więcej niż x ruchów do końca gry; prowadzi on do pewnej sytuacji D. Ponieważ istnieje stos zawierający więcej niż jedną zapałkę, to ruch ten usuwa przynajmniej jedną nieostatnią zapałkę (definicja gry H). Rozważmy także alternatywny ruch gracza B, który usuwa tylko tę jedną zapałkę; prowadzi on do pewnej sytuacji C. Widzimy, że D powstaje z C przez usunięcie pewnych zapałek z pewnych stosów (lub $D=C$). Z lematu 2 wynika, że jeśli z C gracz A mógłby zakończyć grę, wykonując co najwyżej x ruchów, to mógłby także z D, co kończy dowód. ■

Lemat 5. *Rozważmy dwie sytuacje C i D w grze H, przy czym D powstaje z C poprzez przeniesienie jednej zapałki z pewnego stosu i zawierającego $z_i \geqslant 2$ zapałek na pewien stos j zawierający $z_j \geqslant z_i$ zapałek. Załóżmy, że z sytuacji C gracz A może zakończyć grę, wykonując co najwyżej x ruchów. Wówczas także z sytuacji D gracz A może zakończyć grę, wykonując co najwyżej x ruchów.*

Dowód: Indukcja po x. Rozważmy najpierw przypadek, że w C i D jest kolej gracza A. Z lematu 3 wiemy, że A może zapewnić sobie zakończenie gry z C w co najwyżej x ruchach, usuwając największy stos; oznaczmy go przez k. Jeśli $z_j > z_i$, to na pewno

$k \neq i$; również jeśli $z_j = z_i$, to możemy założyć, że $k \neq i$ (nie ma znaczenia, czy usuwamy stos i-ty, czy równoliczny z nim j-ty). Rozważmy także ruch gracza A z D polegający na opróżnieniu stosu k. Niech C' i D' to sytuacje po tych ruchach. Jeśli $k = j$, to D' powstaje z C' przez usunięcie jednej zapałki ze stosu i; wystarczy więc skorzystać z lematu 2. Jeśli zaś $k \neq j$ (oraz ciągle $k \neq i$), to nadal D' powstaje z C' poprzez przeniesienie jednej zapałki ze stosu i zawierającego $z_i \geqslant 2$ zapałek na stos j zawierający $z_j \geqslant z_i$ zapałek. Wystarczy teraz skorzystać z założenia indukcyjnego (gracz A z C' może zakończyć grę, wykonując co najwyżej $x - 1$ ruchów).

Przyjrzyjmy się teraz przypadkowi, że w C i D jest kolej gracza B. Zakładając, że z D jest ruch gracza B, który wymusza więcej niż x ruchów gracza A, rozważmy ten właśnie ruch gracza B. Dzięki lematowi 4 (na j-tym stosie w D mamy co najmniej 3 zapałki, więc założenia są spełnione) możemy założyć, że ruch ten polega na usunięciu jednej, nieostatniej, zapałki z jakiegoś stosu k. Wynikową sytuację oznaczmy D'. Jeśli $k = j$, to z C gracz B usuwa zapałkę ze stosu i (nieostatnią, bo $z_i \geqslant 2$), sprowadzając również C do D'. W przeciwnym przypadku z C gracz B usuwa zapałkę ze stosu k. Wynikowa sytuacja C' również ma tę własność, że D' powstaje z C' poprzez przeniesienie jednej zapałki ze stosu i zawierającego $z'_i \geqslant 2$ zapałek na stos j zawierający $z_j \geqslant z'_i$ zapałek. Faktycznie mamy $z'_i \geqslant 2$, bo jeśli $k \neq i$ to $z'_i = z_i$, a jeśli $k = i$, to na i-tym stosie w D musiały być przynajmniej 2 zapałki (bo usuwaliśmy nieostatnią), czyli w C co najmniej 3, czyli w C' znowu co najmniej 2. Możemy więc użyć pierwszego przypadku dla C' i D'. ■

Lemat 6. *Rozważmy sytuację w grze H, z której jest ruch gracza B. Załóżmy, że B może wymusić, że A wykona więcej niż x ruchów do końca gry. Wówczas B, aby to zrobić, może zacząć od usunięcia jednej zapałki z największego stosu (chyba że wszystkie stosy są rozmiaru co najwyżej* 1*; wtedy musi usunąć wszystkie zapałki).*

Dowód: Jeśli wszystkie stosy są rozmiaru co najwyżej 1, to gracz B nie ma wyboru. Załóżmy więc, że jednak istnieje jakiś stos o liczności co najmniej 2. Rozważmy ruch gracza B w strategii, która wymusza, że A wykona więcej niż x ruchów do końca gry; prowadzi on do pewnej sytuacji D. Dzięki lematowi 4 możemy założyć, że ruch ten polega na usunięciu jednej zapałki z pewnego stosu i. Rozważmy także alternatywny ruch B, który usuwa jedną zapałkę z największego stosu; niech jego numer to j, a sytuacja po tym ruchu to C. Jeśli stosy i i j są tak samo liczne, to nie ma co dowodzić. Jeśli zaś stos i jest mniejszy, to D powstaje z C poprzez przeniesienie jednej zapałki ze stosu i zawierającego $z_i \geqslant 2$ zapałek na pewien stos j zawierający $z_j \geqslant z_i$ zapałek. Mamy przy tym $z_i \geqslant 2$, bo skoro tworząc D usunęliśmy z i-tego stosu nieostatnią zapałkę, to w C muszą tam być co najmniej dwie zapałki. Z kolei $z_j \geqslant z_i$ wynika z tego, że w początkowej sytuacji stos j był najliczniejszy, a stos i mniej liczny od niego. Z lematu 5 wynika więc, że jeśli z C gracz A mógłby zakończyć grę, wykonując co najwyżej x ruchów, to mógłby także z D, co kończy dowód. ■

Rozwiązania częściowe

Dla bardzo małych danych wejściowych możemy napisać rozwiązanie, korzystając z ogólnych metod teorii gier (algorytm min-max), bez analizowania optymalnej strategii konkretnej gry. Wystarczy, że każda rozgrywka jest skończona (każdy ruch gracza A zmniejsza łączną liczbę kamieni na stosach o co najmniej 1, zaś ruchy gracza B nie zmieniają tej liczby). W pliku `wcas1.cpp` zaimplementowano tę metodę; rozwiązanie przechodzi drugie podzadanie.

W przypadku pierwszego podzadania (tylko jedna para stosów), wystarczy jedynie odkryć Obserwacje 1 i 2. Odpowiednie rozwiązanie jest zaimplementowane w pliku `wcab1.cpp`.

Z kolei w pliku `wcas2.cpp` zaimplementowane jest rozwiązanie, które korzysta z obserwacji, co najbardziej opłaca się graczom, gdy już wybiorą stos; wie także, który stos powinien wybierać gracz B. Dla gracza A bada wszystkie możliwości wyboru stosu. Rozwiązanie przechodzi pierwsze trzy podzadania.

Zawody III stopnia

opracowania zadań

Michał Włodarczyk
Treść zadania, Opracowanie

Bartosz Kostka
Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień próbny, 12.04.2016*

Równoważne programy

Bajtazar dostał nowy komputer i uczy się go programować. Program składa się z ciągu instrukcji. Jest k różnych rodzajów instrukcji, które dla uproszczenia oznaczamy liczbami od 1 do k. Niektóre pary instrukcji mają tę własność, że jeśli występują w programie bezpośrednio obok siebie (w dowolnej kolejności), to zamieniając je miejscami, nie zmienia się działania programu (czyli uzyskuje się program **równoważny***). Pozostałe pary instrukcji nie mają tej własności i nazywamy je parami* **nieprzemiennymi***. Bajtazar napisał dwa programy o długości n instrukcji każdy i zastanawia się, czy są one równoważne. Pomóż mu!*

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n, k oraz m pooddzielane pojedynczymi odstępami, oznaczające odpowiednio długość programów, liczbę różnych instrukcji komputera oraz liczbę par instrukcji nieprzemiennych.

Kolejne m wierszy zawiera opis tych par: każdy z tych wierszy zawiera dwie liczby całkowite a i b ($1 \leqslant a < b \leqslant k$) oddzielone pojedynczym odstępem, oznaczające, że para instrukcji o numerach a i b jest nieprzemienna. Możesz założyć, że każda para wystąpi w tym opisie co najwyżej raz.

Kolejne dwa wiersze przedstawiają opisy dwóch programów. Każdy z tych wierszy zawiera ciąg n liczb całkowitych $c_1, c_2, \ldots, c_n$ ($1 \leqslant c_i \leqslant k$) pooddzielanych pojedynczymi odstępami, oznaczających numery kolejnych instrukcji programu.

Wyjście

W jedynym wierszu standardowego wyjścia należy wypisać jedno słowo `TAK` *lub* `NIE` *w zależności od tego, czy podane na wejściu programy są równoważne.*

Przykład

Dla danych wejściowych:

```
5 3 1
2 3
1 1 2 1 3
1 2 3 1 1
```

poprawnym wynikiem jest:

```
TAK
```

natomiast dla danych wejściowych:

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

poprawnym wynikiem jest:

```
NIE
```

Wyjaśnienie do pierwszego przykładu: *W pierwszym programie można zamienić instrukcje na pozycjach 2 i 3, a następnie instrukcję na pozycji 5 z instrukcjami na pozycjach 4 i 3. W ten sposób uzyska się drugi program.*

Testy „ocen":

`1ocen`: $n = 50$, $k = 50$, $m = 1$; *programy to* $(1, 2, \ldots, 49, 50)$ *oraz* $(50, 49, \ldots, 2, 1)$; *odpowiedź* `NIE`.

`2ocen`: $n = 99\,999$, $k = 3$, $m = 1$; *instrukcje nieprzemienne to 1 i 2, a programy to* $(1, 2, 3, 1, 2, 3, \ldots, 1, 2, 3)$ *oraz* $(3, 1, 2, 3, 1, 2, \ldots, 3, 1, 2)$; *odpowiedź* `TAK`.

`3ocen`: $n = 100\,000$, $k = 1000$, $m = 50\,000$; *programy to* $(13, 13, 13, \ldots, 13)$ *oraz* $(37, 37, 37, \ldots, 37)$; *odpowiedź to oczywiście* `NIE`.

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów. We wszystkich testach zachodzą warunki $1 \leqslant n \leqslant 100\,000$, $1 \leqslant k \leqslant 1000$, $0 \leqslant m \leqslant 50\,000$.

Podzadanie	Warunki	Liczba punktów
1	$n \leqslant 5$	*5*
2	$k \leqslant 2$	*5*
3	$n \leqslant 1000$	*25*
4	*brak dodatkowych warunków*	*65*

Rozwiązanie

Zadanie polega na rozstrzygnięciu, czy z jednego z danych programów da się otrzymać drugi za pomocą ciągu zamian sąsiednich instrukcji. Przeszkodą jest lista par instrukcji, których nie wolno ze sobą zamieniać. Jeśli istnieje ciąg zamian przekształcający jeden program w drugi, to programy nazywamy *równoważnymi*. Taka nazwa została wybrana nieprzypadkowo. Pojęcie *relacji równoważności* występuje powszechnie w matematyce i jest uogólnieniem relacji opisanej w zadaniu. Wprawdzie znajomość definicji relacji równoważności nie pomaga w żaden szczególny sposób w wymyśleniu efektywnego algorytmu, ale dostarcza języka pomocnego przy opisie rozwiązań.

Definicja 1. Relację $\approx$ nazywamy relacją równoważności, jeżeli jest ona:

1. **zwrotna**, czyli $x \approx x$ dla każdego x,
2. **symetryczna**, czyli $x \approx y$ zachodzi wtedy i tylko wtedy, gdy $y \approx x$,
3. **przechodnia**, czyli $x \approx y$ w połączeniu z $y \approx z$ implikuje $x \approx z$.

Przykładami relacji równoważności są: równość (znak $\approx$ zastępujemy wtedy znakiem $=$), posiadanie takiej samej reszty z dzielenia przez ustalony dzielnik (relacja równoważności określona na liczbach naturalnych) albo podobieństwo figur na płaszczyźnie. Zauważmy też, że opisana w treści zadania równoważność jest relacją równoważności na zbiorze programów. Jeśli oznaczymy ją przez $\approx$, a pod x, y, z podstawimy dowolne programy, to wszystkie trzy warunki powyższej definicji będą spełnione.

Przykładem relacji zwrotnej i symetrycznej, lecz niekoniecznie przechodniej, jest relacja przemienności instrukcji z zadania. Istotnie, może się okazać, że instrukcja p jest w relacji z instrukcjami q i r i może być zamieniona miejscami z każdą z nich, ale kolejność instrukcji q i r ma znaczenie. Za przykład relacji, która nie jest zwrotna ani symetryczna, lecz jest przechodnia, może posłużyć relacja mniejszości liczb ($<$).

Sortowanie z przeszkodami

W rozwiązaniu skorzystamy z przechodniości relacji równoważności i przekształcimy oba programy do prostszych równoważnych postaci, które będziemy umieli łatwo porównać. Gdyby wszystkie pary instrukcji były przemienne, wystarczyłoby posortować numery instrukcji w każdym programie[1] i sprawdzić, czy otrzymaliśmy takie same ciągi. Okazuje się, że podobny pomysł może zadziałać w ogólniejszym przypadku. Niech x' oznacza najmniejszy w porządku leksykograficznym program równoważny z x. Analogicznie dla programu y definiujemy y'. Jeśli $x \approx y$ (programy x i y są równoważne), to z przechodniości relacji równoważności mamy $x' \approx y'$, co z definicji daje $x' = y'$. Z drugiej strony, jeżeli $x' = y'$, to z przechodniości relacji wnioskujemy, że $x \approx y$.

Przykład 1. Zakładając, że nieprzemienne są pary instrukcji 1, 2 oraz 1, 3, następujące programy są równoważne:

$$x = 1,\ 4,\ 2,\ 3,\ 2,\ 1,\ 4,\ 2,\ 1,\ 4 \quad \text{oraz} \quad y = 4,\ 1,\ 4,\ 3,\ 2,\ 2,\ 1,\ 2,\ 1,\ 4.$$

Najmniejszy leksykograficznie program równoważny x oraz y to:

$$x' = y' = 1,\ 2,\ 2,\ 3,\ 1,\ 2,\ 1,\ 4,\ 4,\ 4.$$

Zanim zaczniemy konstruować algorytm, zauważmy, że mając dane dwa numery instrukcji, umiemy w czasie stałym rozstrzygnąć, czy odpowiadające im instrukcje są przemienne. Możemy chociażby trzymać wszystkie nieprzemienne pary w tablicy haszującej. Nie musimy jednak posuwać się do takich optymalizacji – jako że numery instrukcji należą do przedziału $[1, k]$ gdzie $k \leqslant 1000$, bez problemu zmieścimy w pamięci zwykłą tablicę dwuwymiarową indeksowaną numerami instrukcji.

Jak znaleźć najmniejszy leksykograficznie program równoważny z x? Najprościej zacząć od sprawdzenia, czy na pierwszą pozycję można wstawić instrukcję o numerze 1. Aby było to możliwe, taka instrukcja musi występować w programie i wszystkie instrukcje przed nią muszą być z nią przemienne. Jeśli nie jest to możliwe, sprawdzamy instrukcję o numerze 2 i tak dalej. Gdy znajdziemy pierwszą pasującą instrukcję,

[1] Wykorzystując algorytm sortowania przez zliczanie, można to wykonać w czasie $O(n + k)$.

usuwamy ją z programu i powtarzamy proces dla pozycji 2. Dla każdej z n pozycji musimy sprawdzić potencjalnie k kandydatów. Sprawdzenie jednego kandydata wymaga czasu $O(n)$. Prowadzi to do złożoności obliczeniowej algorytmu $O(n^2k)$, co jest dalece niesatysfakcjonujące.

Aby usprawnić algorytm, zawęźmy zbiór kandydatów na pierwszą instrukcję programu. Mogą to być jedynie te instrukcje, których nie poprzedzają żadne nieprzemienne z nimi. Możemy wyznaczyć taki zbiór w czasie $O(n^2)$. Po wyborze najmniejszej wartości na pierwszą pozycję, chcielibyśmy szybko uaktualnić zbiór kandydatów w celu wyłonienia najlepszej instrukcji na pozycję 2 i kontynuować ten proces, aż skonstruujemy x'.

Niech $S^x[i]$ oznacza liczbę instrukcji leżących przed pozycją i w programie x nieprzemiennych z instrukcją $x[i]$. Początkowo kandydaci znajdują się na pozycjach spełniających $S^x[i] = 0$. Kiedy wybierzemy instrukcję na początek programu (niech pochodzi ona z pozycji i_0), przestaje ona blokować dokładnie te instrukcje na pozycjach $i > i_0$, które nie są z nią przemienne, więc możemy uaktualnić dla nich wartości $S^x[i]$. Tym razem pozycje o $S^x[i]$ równym 0 będą zawierać kandydatów do przesunięcia na pozycję 2 i tak dalej.

Algorytm można najszybciej zrozumieć przez analizę poniższego pseudokodu. Gotowy kod znajduje się w pliku `rows3.cpp`.

```
1:  begin
2:    for i := 1 to n do begin
3:      for j := 1 to i − 1 do
4:        if not commute(x[i], x[j]) then
5:          S^x[i] := S^x[i] + 1;
6:      if S^x[i] = 0 then
7:        candidates_x.insert(i);
8:    end
9:    for pos := 1 to n do begin
10:     i_0 := candidates_x.getMin();
11:     candidates_x.popMin();
12:     x'[pos] := x[i_0];
13:     for i := i_0 + 1 to n do
14:       if not commute(x[i_0], x[i]) then begin
15:         S^x[i] := S^x[i] − 1;
16:         if S^x[i] = 0 then
17:           candidates_x.insert(i);
18:       end
19:   end
20: end
```

W pseudokodzie zakładamy, że tablica S^x jest początkowo wyzerowana oraz że dysponujemy funkcją *commute* rozstrzygającą, czy dane instrukcje są przemienne. Ponadto korzystamy ze struktury danych $candidates_x$ obsługującej następujące operacje:

- *insert*(i) : dodaje element i do struktury,
- *getMin*() : zwraca element i o najmniejszej wartości $x[i]$,

- *popMin*() : usuwa element zwracany przez *getMin*().

Najprostszą strukturą danych implementującą powyższe operacje jest lista. Możemy dodać do niej nowy element w czasie stałym, a wyszukiwanie najlepszego kandydata zajmuje czas liniowy. Zazwyczaj na zawodach algorytmicznych lepiej sprawdza się *kolejka priorytetowa*, pozwalająca wykonać wszystkie operacje w czasie co najwyżej logarytmicznym. Zauważmy jednak, że liczba wywołań funkcji *commute* jest rzędu $\Theta(n^2)$, a operacje na strukturze danych wywoływane są jedynie $O(n)$ razy (każdy element zostaje dodany co najwyżej raz). Zatem niezależnie od wyboru struktury danych otrzymujemy rozwiązanie działające w złożoności obliczeniowej $\Theta(n^2)$.

Przedstawiony algorytm można zoptymalizować tak, aby wykonywał jedynie $O(nk)$ operacji, przy wykorzystaniu faktu, że dla każdego rodzaju instrukcji warto rozważać jedynie jej najwcześniejsze wystąpienie w programie. Takie rozwiązanie wymaga jednak większej staranności w doborze struktur danych; jego opis pomijamy. Zamiast tego w następnej sekcji przedstawiamy odmienne rozwiązanie o takiej samej złożoności obliczeniowej, które posiada elegancki dowód poprawności i jest proste w implementacji. Niemniej jednak zachęcamy Czytelnika do próby wymyślenia szybszego sposobu obliczania x'.

Potęga niezmienników

Rozwiązanie wzorcowe wykorzystuje ciekawą własność relacji równoważności, jaką jest występowanie *niezmienników*.

Definicja 2. Dla relacji $\approx$ określonej na zbiorze X niezmiennikiem nazywamy funkcję $f : X \to Y$, spełniającą warunek $x \approx y \Rightarrow f(x) = f(y)$. Jeśli implikacja zachodzi w obie strony, to niezmiennik nazwiemy *silnym*.

Przykładami niezmienników są liczba kątów dla relacji podobieństwa wielokątów albo naiwnie posortowany ciąg instrukcji dla relacji z zadania. Łatwo znaleźć przykłady na to, że żaden z nich nie jest silny. Silnymi niezmiennikami są za to reszta z dzielenia przez p dla relacji przystawania modulo p tudzież uporządkowany ciąg kątów wewnętrznych dla relacji podobieństwa wielokątów. Analizowane w poprzedniej sekcji przyporządkowanie najmniejszego leksykograficznie równoważnego programu z definicji stanowi silny niezmiennik. Niezmiennik, którego chcemy użyć w rozwiązaniu wzorcowym, wymaga bardziej zaawansowanej konstrukcji.

Definicja 3. Dla programu x oraz numeru instrukcji p konstruujemy ciąg x_p następująco:

(1) zastępujemy każde wystąpienie p literą X,

(2) dopisujemy X na początku i na końcu programu x,

(3) pomiędzy każdy parą kolejnych liter X liczymy instrukcje nieprzemienne z p,

(4) tworzymy ciąg x_p, zapisując kolejno wartości obliczone w punkcie (3).

Twierdzenie 1. *Rodzina ciągów $(x_p)_{p=1}^k$ stanowi silny niezmiennik równoważności programów. Innymi słowy, programy x, y są równoważne wtedy i tylko wtedy, gdy dla każdego p zachodzi $x_p = y_p$.*

W powyższym twierdzeniu dwa ciągi uznajemy za równe, jeśli mają tyle samo elementów i elementy o tych samych indeksach są równe.

Dowód: Implikacja $\Rightarrow$ (wykazanie, że przyporządkowanie jest niezmiennikiem). Jeżeli x i y są równoważne, to istnieje sekwencja zamian sąsiednich instrukcji, które są przemienne, przeprowadzająca pierwszy program na drugi. Wystarczy wobec tego zauważyć, że zamiana przemiennych instrukcji nie może zmodyfikować żadnej wartości w żadnym ciągu x_p.

Implikacja $\Leftarrow$. Indukcja po liczbie instrukcji w programie. Programy x, y o długości 1 są równoważne wtedy i tylko wtedy, gdy składają się z tej samej instrukcji q. Dla tejże instrukcji zachodzi $x_q = y_q = (0, 0)$, dla pozostałych mamy zaś $x_p = y_p = (0)$.

Przypuśćmy teraz, że $n > 1$ oraz dla każdego p zachodzi $x_p = y_p$. Niech q będzie pierwszą instrukcją w programie x. Oznacza to, że pierwsza wartość w ciągu x_q (a zarazem y_q) to 0. W takim razie w programie y wszystkie instrukcje znajdujące się przed pierwszym wystąpieniem instrukcji q są z nią przemienne. Niech y' oznacza program otrzymany z y przez przesunięcie pierwszej instrukcji q na początek programu. Oczywiście $y \approx y'$.

Niech x'', y'' oznaczają programy otrzymane z x, y' przez usunięcie początkowej instrukcji q. Zauważmy, że ciąg x''_q różni się od x_q jedynie brakiem początkowego 0 i jest tożsamy z y''_q. Jeśli $p \neq q$ i p jest nieprzemienne z q, to ciągi x''_p, y''_p można otrzymać przez odjęcie 1 od pierwszego wyrazu w ciągach x_p, y'_p. Jeśli zaś p jest przemienne z q, $x''_p = x_p$ i $y''_p = y'_p$. Wobec tego dla każdego p zachodzi $x''_p = y''_p$ i z założenia indukcyjnego wnioskujemy, że $x'' \approx y''$. Dopisanie tej samej instrukcji na początku programu zachowuje równoważność ciągów, zatem $x \approx y'$. Ostatecznie korzystamy z przechodniości, aby otrzymać $x \approx y$. ■

Przykład 2. Dla programu $x = 1,\ 4,\ 2,\ 3,\ 2,\ 1,\ 4,\ 2,\ 1,\ 4$ z przykładu 2 (nieprzemienne są pary instrukcji 1, 2 oraz 1, 3) niezmiennikiem jest:

$$x_1 = (0, 3, 1, 0), \quad x_2 = (1, 0, 1, 1), \quad x_3 = (1, 2), \quad x_4 = (0, 0, 0, 0).$$

Przykładowo, wyznaczając x_1, zapisujemy następujący ciąg:

$$x = \mathrm{X},\ \mathrm{X},\ \cdot,\ \mathrm{N},\ \mathrm{N},\ \mathrm{N},\ \mathrm{X},\ \cdot,\ \mathrm{N},\ \mathrm{X},\ \cdot,\ \mathrm{X},$$

gdzie $\cdot$ oznacza instrukcję przemienną z 1, a N instrukcję nieprzemienną z 1.

Taki sam niezmiennik ma oczywiście program y z przykładu 2.

Rozwiązanie wzorcowe zaimplementowane w pliku `row.cpp` konstruuje wszystkie ciągi x_p, y_p, po czym je porównuje. Wymaga to przeiterowania po obu programach dla każdej wartości $1 \leqslant p \leqslant k$. Jako że umiemy sprawdzić w czasie stałym, czy dwie instrukcje są nieprzemienne, złożoność obliczeniowa algorytmu wynosi $O(nk)$.

Wojciech Nadara
Treść zadania, Opracowanie

Michał Glapa, Marek Sokołowski
Program

Dostępna pamięć: 256 MB.

OI, etap III, dzień pierwszy, 13.04.2016

Posłaniec

Bajtazar po długim okresie swojego panowania w królestwie Bajtocji stwierdził, że to zajęcie bardzo go wyczerpało, i ustąpił z tronu. Jednak przywykł on do życia w wyższych sferach i chciałby pozostać na bieżąco z najważniejszymi wiadomościami dotyczącymi królestwa i dworu. Dlatego postanowił zostać królewskim posłańcem.

Już pierwszego dnia na nowym stanowisku zlecono mu dostarczenie pilnej wiadomości pomiędzy dwoma miastami królestwa. Stwierdził on jednak, że w trakcie pracy zrobi sobie małą wycieczkę krajoznawczą i niekoniecznie pojedzie najkrótszą możliwą trasą. Oczywiście nie może dopuścić, aby nowy król się o tym dowiedział – w końcu posłaniec powinien dostarczać wiadomości tak szybko, jak to tylko możliwe!

Wszystkie połączenia drogowe w Bajtocji są jednokierunkowe. Bajtazar zna doskonale całe królestwo i wie, między którymi miastami istnieją połączenia drogowe. Zadeklarował on liczbę połączeń, których chciałby użyć, przejeżdżając pomiędzy miastami, i planuje on przejechać pomiędzy nimi dowolną ścieżką wymagającą dokładnie tylu połączeń (nie zważając na to, ilu połączeń tak naprawdę wymaga taka podróż). W trakcie swojej podróży Bajtazar nie może jednak pojawić się w początkowym ani końcowym mieście więcej niż raz, gdyż wtedy wzbudziłby podejrzenia u królewskich oficjeli. Może on jednak wielokrotnie pojawiać się w innych miastach, jak też wielokrotnie korzystać z tych samych połączeń drogowych.

Pomóż naszemu bohaterowi i napisz program, który obliczy dla niego, na ile sposobów może on zrealizować swoją wycieczkę krajoznawczą. Innymi słowy, program ma wyznaczyć liczbę różnych ścieżek o zadanej długości pomiędzy dwoma wybranymi miastami królestwa (przy czym w mieście początkowym i końcowym można pojawić się tylko raz). Ponieważ wynik zapytania może być całkiem duży, wystarczy, że program poda resztę z dzielenia wyniku przez pewną wybraną przez Bajtazara liczbę.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n, m i z ($n \geqslant 2$, $0 \leqslant m \leqslant n(n-1)$, $2 \leqslant z \leqslant 1\,000\,000\,000$) pooddzielane pojedynczymi odstępami, oznaczające odpowiednio: liczbę miast w Bajtocji, liczbę jednokierunkowych połączeń między nimi oraz liczbę wybraną przez Bajtazara. Miasta numerujemy liczbami od 1 do n.

Dalej następuje m wierszy; każdy zawiera parę liczb całkowitych a, b ($1 \leqslant a, b \leqslant n$, $a \neq b$) oddzielonych pojedynczym odstępem, opisującą jednokierunkowe połączenie z miasta o numerze a do miasta o numerze b. Żadne połączenie nie jest podane na wejściu wielokrotnie.

W kolejnym wierszu wejścia znajduje się dodatnia liczba całkowita q oznaczająca liczbę zapytań Bajtazara. W każdym z następnych q wierszy znajduje się opis jednego zapytania. Opis taki składa się z trzech liczb całkowitych u_i, v_i i d_i ($1 \leqslant u_i, v_i \leqslant n$, $u_i \neq v_i$, $1 \leqslant d_i \leqslant 50$) pooddzielanych pojedynczymi odstępami, oznaczających, że Bajtazar ma przejechać z miasta o numerze u_i do miasta o numerze v_i, używając dokładnie d_i połączeń.

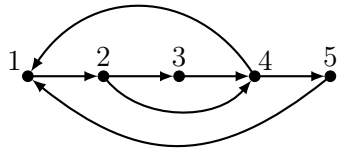
Wyjście

Na standardowe wyjście należy wypisać dokładnie q wierszy. W i-tym wierszu należy wypisać resztę z dzielenia przez z liczby ścieżek z i-tego zapytania.

Przykład

Dla danych wejściowych:

```
5 7 10
1 2
2 3
3 4
4 5
5 1
2 4
4 1
2
2 1 3
5 3 6
```



poprawnym wynikiem jest:

```
2
1
```

Wyjaśnienie do przykładu: *Dla pierwszego zapytania mamy dwie możliwe ścieżki: $2 \to 3 \to 4 \to 1$ oraz $2 \to 4 \to 5 \to 1$; dla drugiego zapytania tylko jedną: $5 \to 1 \to 2 \to 4 \to 1 \to 2 \to 3$.*

Testy „ocen”:

`1ocen`: *$n = 6$, $q = 10$, pięć miast połączonych ze sobą bezpośrednio w obie strony; szóste miasto niepołączone z żadnym innym;*

`2ocen`: *$n = 20$, $q = 100$, miasta w Bajtocji położone są na okręgu; każde dwa sąsiednie miasta na tym okręgu są połączone ze sobą bezpośrednio w obie strony;*

`3ocen`: *$n = 100$, $q = 500\,000$, mapa Bajtocji ma kształt trójzębu.*

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	$n \leqslant 20$, $q \leqslant 100$	*12*
2	$n \leqslant 100$, $m \leqslant 500$, $q \leqslant 100$	*20*
3	$n \leqslant 100$, $q \leqslant 10\,000$	*38*
4	$n \leqslant 100$, $q \leqslant 500\,000$	*30*

Rozwiązanie

Sytuację opisaną w treści zadania można w naturalny sposób przedstawić za pomocą grafu skierowanego. Miasta Bajtocji odpowiadają wierzchołkom grafu, a połączenia – krawędziom. Zbiory wierzchołków i krawędzi grafu oznaczymy jako V oraz E (mamy $|V| = n$, $|E| = m$).

W opisie rozwiązania będziemy używali dwóch sposobów reprezentacji grafu. Pierwszy z nich to macierz sąsiedztwa, czyli tablica M rozmiaru $n \times n$, taka że $M[a][b] = 1$, jeżeli istnieje krawędź z wierzchołka a do wierzchołka b, natomiast w przeciwnym przypadku $M[a][b] = 0$. Drugi natomiast to listy sąsiedztwa, czyli w naszym przypadku tablica n list określających zbiory krawędzi wchodzących do poszczególnych wierzchołków grafu.

Początkowe spostrzeżenia

W opisie rozwiązania będziemy wielokrotnie posługiwali się wartościami typu „liczba sposobów dojścia od wierzchołka a do wierzchołka b za pomocą dokładnie k krawędzi". Oznaczmy taką wartość przez $ways(a, b, k)$. Zauważmy, że $ways(a, b, 0)$ jest równe 0 dla $a \neq b$ i 1 dla $a = b$. Ponadto mamy $ways(a, b, 1) = M[a][b]$. Zastanówmy się, jak można efektywnie obliczać wartości $ways(a, b, k)$ dla $k > 1$. Załóżmy, że chcemy przejść od wierzchołka a do wierzchołka b w k krokach. Wówczas któryś wierzchołek c odwiedzimy bezpośrednio przed wierzchołkiem b. Rozpatrując wszystkie takie wierzchołki c, że istnieje krawędź z c do b, jesteśmy w stanie stwierdzić, że

$$ways(a, b, k) = \sum_{cb \in E} ways(a, c, k - 1). \tag{1}$$

Wzór ten pozwala obliczyć wszystkie wartości $ways(a, b, k)$ dla $a, b \in V$, $0 \leqslant k \leqslant K$ za pomocą programowania dynamicznego, w kolejności rosnących wartości k. Mamy $O(n^2K)$ stanów. Aby oszacować liczbę przejść, zauważmy, że dla ustalonych a oraz k we wzorze rozważamy wszystkie wierzchołki b i dla każdego z nich wszystkie krawędzie wchodzące – dla ustalonych a i k mamy więc m przejść. Tak więc obliczenia wykonamy za pomocą list sąsiedztwa w złożoności $O(nmK)$.

Wyrażając (1) w terminach macierzy sąsiedztwa, możemy także napisać

$$ways(a, b, k) = \sum_{c \in V} ways(a, c, k - 1) \cdot M[c][b]. \tag{2}$$

W tym przypadku suma jest po wszystkich wierzchołkach c, a nie tylko po poprzednikach wierzchołka b, a fakt, czy istnieje krawędź od c do b, wyrażamy za pomocą domnożenia składnika przez $M[c][b]$. Można stąd wysnuć wniosek, że $ways(a, b, k) = M^k[a][b]$, gdzie M^k to macierz sąsiedztwa podniesiona do k-tej potęgi. Warto znać ten fakt, jednak w takiej konkretnie postaci nie będzie on nam dalej potrzebny.

Od tego momentu będziemy zakładać, że potrzebne nam wartości funkcji *ways* zostały obliczone na początku algorytmu.

Rozwiązanie brutalne

Jedno z możliwych rozwiązań zadania polega na użyciu programowania dynamicznego do każdego z zapytań oddzielnie. Ustalmy jedno z nich, z parametrami u, v, d. Będziemy obliczać wartości $dp[w][k]$ oznaczające, na ile sposobów można dojść od wierzchołka u do wierzchołka w za pomocą k krawędzi, nie odwiedzając w trakcie drogi ani u, ani v. Robimy to tak samo, jakbyśmy obliczali $ways(u, w, k)$, z jedyną różnicą, że pomijamy $w = u$ dla wszystkich wartości $k > 0$ oraz $w = v$ dla wszystkich wartości $k < d$. Takie rozwiązanie odpowiada na konkretne zapytanie w złożoności $O(md)$. Jeżeli zatem przez D oznaczymy największą wartość d_i występującą w zapytaniach, to możemy stwierdzić, że rozwiązanie działa w złożoności $O(mqD)$. Rozwiązanie o takiej złożoności wystarczało do przejścia pierwszych dwóch podzadań.

Rozwiązanie wzorcowe

Wprowadźmy najpierw kilka terminów związanych z grafami. Ścieżki w grafie, w których krawędzie i wierzchołki mogą się powtarzać (czyli takie, jakie interesują nas w tym zadaniu), nazywamy *marszrutami*. Ponadto *wnętrzem marszruty* $v_1v_2 \ldots v_{k-1}v_k$ nazwijmy zbiór wierzchołków $\{v_2, \ldots, v_{k-1}\}$. Będziemy mówili, że marszruta ma parametry u, v, d, jeżeli zaczyna się ona w wierzchołku u, kończy w wierzchołku v oraz przechodzi przez d krawędzi. Powiemy także, że zapytanie ma parametry u, v, d, jeżeli jesteśmy w nim proszeni o wyznaczenie liczby marszrut z takimi parametrami, które w swoim wnętrzu nie zawierają u ani v.

Główny pomysł rozwiązania wzorcowego jest dość ogólną metodą liczenia *dobrych obiektów* jako różnicy liczby *wszystkich obiektów* oraz *złych obiektów*. Ustalmy konkretne zapytanie z parametrami u, v, d. *Dobrymi obiektami* są w naszym przypadku marszruty z u do v o długości d odwiedzające u i v tylko na początku i końcu (odpowiednio). *Wszystkimi obiektami* będą marszruty z u do v o długości d, a *złymi obiektami* będą marszruty z u do v o długości d, których wnętrze zawiera u lub v. Niech zatem $good(a, b, k)$ i $bad(a, b, k)$ oznaczają odpowiednio liczbę dobrych i złych marszrut o długości k zaczynających się w a i kończących w b. Będziemy rozważać tylko $a, b \in \{u, v\}$ (przy czym potencjalnie może zachodzić $a = b$ lub $a = v$, $b = u$).

Złym wystąpieniem wierzchołka na marszrucie o parametrach a, b, k nazwijmy każde wystąpienie u lub v w jej wnętrzu. Przyjmijmy, że marszruta od a do b długości k jest zła. Rozważmy ostatnie złe wystąpienie wierzchołka na tej marszrucie. Załóżmy, że było nim odwiedzenie wierzchołka $w \in \{u, v\}$ po l krokach. Taka zła marszruta na odcinku swoich pierwszych l kroków mogła być dowolną marszrutą o parametrach a, w, l, natomiast na kolejnym odcinku o długości $k - l$ była dobrą marszrutą o parametrach $w, b, k - l$. Zatem sumarycznie takich złych marszrut jest $ways(a, w, l) \cdot good(w, b, k - l)$. Ostatnie złe wystąpienie mogło być wystąpieniem zarówno wierzchołka u jak i v oraz l mogło przyjąć jakąkolwiek wartość od 1 do $k - 1$. Stąd wniosek, że

$$\begin{aligned} bad(a, b, k) = {} & (ways(a, u, 1) \cdot good(u, b, k - 1) + \ldots + ways(a, u, k - 1) \cdot good(u, b, 1)) \\ & + (ways(a, v, 1) \cdot good(v, b, k - 1) + \ldots + ways(a, v, k - 1) \cdot good(v, b, 1)). \end{aligned}$$

Natomiast $good(a, b, k) = ways(a, b, k) - bad(a, b, k)$. Dzięki tym wzorom, dla zapytania o parametrach u, v, d możemy napisać krótki pseudokod realizujący programowanie dynamiczne liczące wszystkie interesujące nas wartości $good(a, b, k)$, gdzie $a, b \in \{u, v\}$, $1 \leqslant k \leqslant d$.

```
1: for k := 1 to d do
2:   foreach a in {u, v} do
3:     foreach b in {u, v} do
4:       good[a][b][k] := ways(a, b, k);
5:       foreach last in {u, v} do
6:         for l := 1 to k − 1 do
7:           good[a][b][k] := good[a][b][k] − ways(a, last, l) · good[last][b][k − l];
```

Interesująca nas odpowiedź będzie obliczona w polu $good[u][v][d]$.

Na całe rozwiązanie składa się preprocessing tablicujący wszystkie wartości funkcji $ways(a, b, k)$ dla $k \leqslant D$ w złożoności $O(nmD)$ oraz odpowiadanie na zapytania w złożoności $O(qd^2)$, co daje rozwiązanie w złożoności $O(nmD + qD^2)$.

Wojciech Nadara
Treść zadania, Opracowanie

Paweł Czerwiński
Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień pierwszy, 13.04.2016*

Pracowity Jaś

Jaś miał niedawno urodziny. Jak w każdym szanującym się zadaniu algorytmicznym, Jaś nie dostał w prezencie ani zabawek, ani gier, ani komputera, a jedynie długie tablice wypełnione liczbami, drzewa, mapy różnych krajów z drogami prowadzącymi tunelami i estakadami oraz długie taśmy z wypisanymi 1048576 początkowymi literami słów Fibonacciego i Thuego-Morse'a. Najbardziej z tych wszystkich prezentów spodobała mu się tablica z wypisaną permutacją[1] *pierwszych n liczb naturalnych. Zaczął się zastanawiać, jaka jest poprzednia permutacja w porządku leksykograficznym*[2]*. Po chwili udało mu się ją wymyślić i zapragnął zapisać ją na tej samej tablicy. Jaś potrafi w jednym kroku zamienić miejscami jedynie dwie z liczb zapisanych na tablicy (gdyby operował na większej liczbie liczb naraz, to by się pogubił). Jest jednak na tyle mądry, że przekształcił początkową permutację w poprzednią leksykograficznie, wykonując minimalną liczbę takich zamian. Gdy to zrobił, wpadł w permutacyjny szał i zaczął powtarzać tę operację w kółko, zapisując na tablicy kolejne wcześniejsze permutacje w porządku leksykograficznym!*

Niestety, po pewnym czasie Jaś będzie musiał przerwać swoją zabawę, gdyż dojdzie do permutacji $1, 2, \ldots, n$*, która jest najmniejsza w porządku leksykograficznym. Przyjaciele Jasia trochę się naśmiewają z jego monotonnej zabawy, jednak wiedzą, że nie mają szans go z niej wyrwać. Chcieliby się więc dowiedzieć, kiedy Jaś skończy. Pomóż kolegom Jasia i powiedz im, ile potrwa jego zabawa, jeśli każda zamiana zajmuje mu dokładnie sekundę. Jako że ta rozrywka może być dość długa (Jaś nie bez powodu ma przydomek Pracowity), wystarczy im reszta z dzielenia tej liczby przez* $10^9 + 7$*. W końcu* $10^9 + 7$ *sekund to na tyle długo, że są w stanie co tyle czasu sprawdzać, czy Jaś już skończył swoją zabawę.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się dodatnia liczba całkowita n oznaczająca długość permutacji, którą Jaś otrzymał na urodziny. W drugim wierszu znajduje się ciąg n parami różnych liczb naturalnych $p_1, p_2, \ldots, p_n$ $(1 \leqslant p_i \leqslant n)$ *pooddzielanych pojedynczymi odstępami, opisujący tę permutację.*

Wyjście

Twój program powinien wypisać na standardowe wyjście resztę z dzielenia przez $10^9 + 7$ *liczby zamian, które wykona Jaś, zanim jego zabawa się skończy.*

[1] Permutacja liczb od 1 do n to ciąg różnych liczb całkowitych $p_1, \ldots, p_n$ spełniających $1 \leqslant p_i \leqslant n$ (każda liczba całkowita od 1 do n występuje w takiej permutacji dokładnie raz).

[2] Permutacja $P = (p_1, \ldots, p_n)$ jest wcześniej w porządku leksykograficznym niż permutacja $Q = (q_1, \ldots, q_n)$ (co zapiszemy jako $P < Q$), jeżeli $p_j < q_j$, gdzie j jest najmniejszym takim indeksem, że $p_j \neq q_j$. Permutacja P poprzedza permutację Q w porządku leksykograficznym, jeżeli $P < Q$ oraz nie istnieje taka permutacja R, że $P < R < Q$.

Przykład

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

Wyjaśnienie do przykładu: *Na tablicy Jasia będą się po kolei pokazywały permutacje $(2,3,1)$, $(2,1,3)$, $(1,3,2)$, $(1,2,3)$. Aby je uzyskiwać, będzie musiał wykonać łącznie $2+1+2+1=6$ zamian.*

Testy „ocen":

`1ocen`: $1, 2, 3, \ldots, 10$

`2ocen`: *losowa 5-elementowa permutacja*

`3ocen`: $100, 99, 98, \ldots, 1$.

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	$n \leqslant 5000$	*37*
3	$n \leqslant 1\,000\,000$, *permutacja to* $n, n-1, \ldots, 1$	*15*
4	$n \leqslant 1\,000\,000$	*33*

Rozwiązanie

Poprzednia leksykograficznie permutacja

Pierwszym pytaniem, jakie należy sobie postawić, jest „Dla danej permutacji $P = (p_1, \ldots, p_n)$, jak wygląda jej leksykograficzny poprzednik?". Nazwijmy go Q. Wśród wszystkich permutacji mniejszych leksykograficznie od P, Q jest ma maksymalny wspólny prefiks (początkowy kawałek) z P. Na początek chcielibyśmy zidentyfikować długość tego prefiksu. Podzielmy naszą permutację P na pewien prefiks oraz sufiks (końcowy kawałek) i oznaczmy je jako A i B. Leksykograficznie mniejsza od P permutacja o prefiksie równym A istnieje wtedy i tylko wtedy, gdy ciąg B nie jest posortowany rosnąco. Stąd możemy wysnuć wniosek, że P i Q będą miały wspólny prefiks o długości $k-1$, gdzie k to największy taki indeks, że $p_k > p_{k+1}$ (jeżeli P nie jest permutacją $(1, \ldots, n)$, która nie ma leksykograficznego poprzednika, to taki indeks istnieje). Sufiks $(p_k, \ldots, p_n)$ zaczynający się od k-tej pozycji nazwijmy *ogonem* permutacji P. Permutację Q możemy opisać w następujący sposób: ma ona wspólny prefiks z P o długości $k-1$ (tzn. $q_i = p_i$ dla $i = 1, \ldots, k-1$), q_k jest największą liczbą ze zbioru $\{p_{k+1}, \ldots, p_n\}$ mniejszą od p_k oraz $(q_{k+1}, \ldots, q_n)$ jest posortowanym

malejąco ciągiem złożonym z liczb ze zbioru $\{1, \ldots, n\}$, które nie występują w zbiorze $\{q_1, \ldots, q_k\}$.

Przykładowo dla permutacji $P = (2, 4, 1, 3, 5)$ mamy $k = 2$, zatem $q_1 = p_1 = 2$, największą liczbą z liczb $\{1, 3, 5\}$, która jest mniejsza od $p_2 = 4$, jest $q_2 = 3$, a $(q_3, q_4, q_5) = (5, 4, 1)$. Zatem bezpośrednim poprzednikiem leksykograficznym permutacji P jest permutacja $Q = (2, 3, 5, 4, 1)$. W dalszej części rozwiązania, bezpośredniego leksykograficznego poprzednika permutacji P będziemy oznaczać przez $prev(P)$.

Minimalna liczba zamian

Skoro już wiemy, jak wygląda leksykograficzny poprzednik permutacji P, to zastanówmy się, jaka jest minimalna liczba zamian par liczb, która pozwoli nam przejść od P do $prev(P)$.

Przyjmijmy, że ogon P to sufiks złożony z elementów o indeksach $[k, n]$. Przypomnijmy, że k możemy wyznaczyć jako największy taki indeks, że $p_k > p_{k+1}$. Na potrzeby tej sekcji dla prostoty załóżmy, że ogon ten jest permutacją liczb od 1 do t, i niech jego pierwszy element będzie równy r (mamy $r > 1$). Wtedy ogon wygląda następująco:

$$S_1 = (r, 1, \ldots, r-2, r-1, r+1, \ldots, t),$$

a jego poprzednik to

$$S_2 = (r-1, t, t-1, \ldots, r+1, r, r-2, \ldots, 1).$$

Zauważmy, że od S_1 do S_2 możemy przejść, najpierw zamieniając miejscami liczby r oraz $r-1$, a następnie odwracając sufiks $1, \ldots, r-2, r, r+1, \ldots, t$, otrzymując żądany $t, t-1, \ldots, r+1, r, r-2, \ldots, 1$. Odwrócenie ciągu c-elementowego można wykonać za pomocą $\lfloor \frac{c}{2} \rfloor$ operacji zamiany dwóch elementów[1]. W naszym przypadku $c = t - 1$, zatem jeżeli P ma ogon długości t, to potrafimy przejść z P do $prev(P)$ za pomocą $1 + \lfloor \frac{t-1}{2} \rfloor = \lfloor \frac{t+1}{2} \rfloor$ operacji.

Udowodnimy teraz, że jest to minimalna liczba takich operacji. Jest jasne, że jeżeli chcemy z pewnego ciągu A otrzymać ciąg B za pomocą operacji zamiany par elementów i te ciągi różnią się na d pozycjach, to potrzebujemy co najmniej $\lceil \frac{d}{2} \rceil$ operacji. Przekonamy się, że takie oszacowanie wystarczy nam do udowodnienia tezy.

Zauważmy na starcie, że ciągi S_1 i S_2 różnią się na pozycjach, na których w S_1 stoją liczby r i $r-1$. Dalej rozważymy dwa przypadki ze względu na parzystość t.

1. $t = 2l + 1$:

   Jeżeli ponumerujemy pozycje obu ciągów indeksami od 1 do t, to wtedy w S_1 liczby od 1 do $r-2$ stoją na pozycjach o parzystościach różnych od swoich wartości, a liczby od $r+1$ do t stoją na pozycjach o parzystościach takich samych jak swoje wartości. W przypadku S_2 mamy do czynienia z dokładnie odwrotną sytuacją. Stąd wniosek, że S_1 i S_2 różnią się na wszystkich pozycjach, zatem na otrzymanie S_2 z S_1 potrzeba co najmniej $\lceil \frac{t}{2} \rceil = l + 1 = \lfloor \frac{t+1}{2} \rfloor$ zamian.

[1] Przypomnijmy standardowe oznaczenia matematyczne: $\lfloor x \rfloor$ oraz $\lceil x \rceil$ oznaczają odpowiednio zaokrąglenia x do najbliższej liczby całkowitej w dół i w górę.

2. $t = 2l$:

   Po usunięciu pierwszych elementów z S_1 oraz S_2, pierwszy z nich staje się ciągiem rosnącym, a drugi malejącym. Ciąg rosnący i ciąg malejący mogą mieć co najwyżej jedną pozycję wspólną. Stąd wniosek, że S_1 i S_2 różnią się na co najmniej $t-1$ pozycjach, zatem do otrzymania S_2 z S_1 potrzeba co najmniej $\lceil \frac{t-1}{2} \rceil = l = \lfloor \frac{t+1}{2} \rfloor$ zamian.

W obu przypadkach oszacowaliśmy z dołu liczbę potrzebnych zamian przez liczbę, którą otrzymujemy we wcześniej opisanej metodzie, co dowodzi tego, że przedstawiony algorytm wykonuje minimalną liczbę zamian.

W szczególności dowiedzieliśmy się, że liczba zamian potrzebnych do przekształcenia S_1 w S_2 nie zależy od r, a jedynie od długości tych ciągów t. Pozwala nam to zdefiniować funkcję $g(t) = \lfloor \frac{t+1}{2} \rfloor$, która oznacza najmniejszą potrzebną liczbę zamian do przejścia od permutacji P do $prev(P)$, jeśli ogon permutacji P ma długość t.

Obliczenie sumarycznej długości zabawy

Mając do dyspozycji wniosek z poprzedniego akapitu, jesteśmy już w stanie napisać pewne (bardzo wolne) rozwiązanie. Dopóki nie otrzymamy permutacji identycznościowej, przechodzimy do poprzedniej leksykograficznie permutacji i zwiększamy liczbę potrzebnych zamian o odpowiednią wartość. Jednak takie rozwiązanie może działać bardzo długo, konkretnie w złożoności $O(n! \cdot n)$ lub $O(n!)$, jeżeli będziemy wyznaczali długość ogonu, idąc od końca permutacji. (Dociekliwemu Czytelnikowi pozostawiamy jako nieoczywiste zadanie udowodnienie, że średnią długość ogonu możemy ograniczyć przez stałą). Takie rozwiązanie wystarcza jedynie do rozwiązania pierwszego podzadania; w ogólności potrzebujemy czegoś zdecydowanie szybszego.

Na początku zauważmy, że analogiczny problem możemy zdefiniować dla dowolnych ciągów różnych liczb. Ciąg różnych liczb o długości m, tak samo jak ciąg wszystkich liczb od 1 do m, można spermutować na $m!$ sposobów i algorytm znajdowania leksykograficznego poprzednika wygląda w tym przypadku analogicznie. Dla ciągu $B = (b_1, \ldots, b_m)$ składającego się z różnych liczb możemy zdefiniować ciąg $compr(B) = C = (c_1, \ldots, c_m)$, gdzie c_i oznacza, ile liczb w ciągu B jest równych co najwyżej b_i. Ciąg C jest permutacją liczb od 1 do m. Na przykład $compr((7,3,5)) = (3,1,2)$. Jeżeli dla dowolnego ciągu różnych liczb $D = (d_1, \ldots, d_m)$ przez $ans(D)$ oznaczymy odpowiedź dla problemu analogicznego do problemu z treści, to jest jasne, że $ans(D) = ans(compr(D))$.

Przez $f(n)$ oznaczmy $ans((n, n-1, \ldots, 1))$. Zauważmy, że zaczynając od ciągu $(n, n-1, \ldots, 1)$, w trakcie zabawy Jaś napotka permutację $(n, 1, \ldots, n-1)$, a do tego momentu wykona $f(n-1)$ zamian. Ogon tej permutacji ma długość n, zatem w następnym kroku będzie musiał wykonać $g(n)$ zamian, otrzymując permutację $(n-1, n, n-2, n-3, \ldots, 1)$. Po pewnej liczbie kroków dojdzie do permutacji $(n-1, 1, \ldots, n-3, n-2, n)$, wykonując kolejne $f(n-1)$ zamian. Ogon tej permutacji ponownie ma długość n, zatem aby przejść do $(n-2, n, n-1, n-3, \ldots, 1)$, musi wykonać kolejne $g(n)$ zamian. Kontynuując to rozumowanie, dochodzimy do wniosku, że:

$$f(n) = n \cdot f(n-1) + (n-1) \cdot g(n),$$

co pozwala nam w czasie $O(n)$ wyznaczyć reszty z dzielenia $f(1), f(2), \ldots, f(n)$ przez zadaną stałą (wiedząc, że $f(1) = 0$). Otrzymujemy zarazem rozwiązanie podzadania 3.

Rozwiązanie ogólnego problemu wcale nie jest wiele trudniejsze. Jeżeli rozwiązujemy problem dla permutacji $P = (p_1, \ldots, p_n)$, to po pewnej liczbie zamian dojdziemy do ciągu $(p_1, 1, \ldots, p_1 - 1, p_1 + 1, \ldots, n)$ i od tego momentu wykonamy $(p_1 - 1) \cdot (f(n-1) + g(n))$ zamian, aby dojść do permutacji identycznościowej. Możemy zatem stwierdzić, że:

$$\begin{aligned} ans(P) &= ans((p_2, \ldots, p_n)) + (p_1 - 1) \cdot (f(n-1) + g(n)) \\ &= ans(compr((p_2, \ldots, p_n))) + (p_1 - 1) \cdot (f(n-1) + g(n)). \end{aligned}$$

Wartość $ans(compr((p_2, \ldots, p_n))$ możemy wyznaczyć, obliczając najpierw permutację $compr((p_2, \ldots, p_n))$ i wywołując się rekurencyjnie. Obliczenie permutacji $compr((p_2, \ldots, p_n))$ możemy łatwo wykonać w czasie $O(n)$ (pamiętajmy, że zbiór liczb $\{p_2, \ldots, p_n\}$ to zbiór liczb od 1 do n oprócz p_1), co daje nam rozwiązanie w czasie $O(n^2)$. Jest to zdecydowaną poprawą w stosunku do rozwiązania $O(n!)$ i pozwala zaliczyć podzadanie 2, ale wciąż nie wystarcza na zdobycie kompletu punktów.

Zauważmy jednak, że do obliczenia liczby zamian różniących $ans(P)$ oraz $ans((p_2, \ldots, p_n))$ potrzebujemy jedynie znać wartość elementu p_1. Jeżeli rozpatrujemy problem dla ogólnych ciągów różnych elementów, to wtedy analogiczna wersja wzoru z poprzedniego akapitu przedstawia się jako $ans(P) = ans((p_2, \ldots, p_n)) + s \cdot (f(n-1) + g(n))$, gdzie s to liczba elementów ciągu P mniejszych od p_1. Przewaga takiego zapisu polega na tym, że nie musimy wykonywać kosztownej kompresji przed wywołaniem rekurencyjnym. Niestety ma on też swoją wadę, mianowicie musimy znać wartość s. Zauważmy, że jeżeli oryginalną permutacją z zadania jest $(p_1, \ldots, p_n)$, to będziemy się wywoływać rekurencyjnie jedynie dla jej sufiksów. Po chwili zastanowienia możemy dojść do wniosku, że tak naprawdę wynik naszego zadania przedstawia się jako

$$s(1) \cdot (f(n-1) + g(n)) + s(2) \cdot (f(n-2) + g(n-1)) + \ldots + s(n-1) \cdot (f(1) + g(2)),$$

gdzie $s(i)$ to liczba liczb mniejszych od p_i spośród liczb $p_{i+1}, \ldots, p_n$. Jedyne, co musimy zatem zrobić, to efektywnie obliczyć wartości $s(i)$.

Będziemy potrzebowali struktury danych wspierającej zapytania $insert(i)$ oraz $less(i)$, gdzie $insert(i)$ dodaje element i do zbioru, a $less(i)$ odpowiada na zapytanie „Ile liczb mniejszych od i dodaliśmy już do zbioru?". Co więcej, w naszym zadaniu argumenty operacji są liczbami całkowitymi z przedziału $[1, n]$. Biorąc to wszystko pod uwagę, widzimy, że otrzymany problem to standardowy przykład na użycie drzew potęgowych lub przedziałowych (typu „dodaj punkt, sumuj na przedziale"), które to już wielokrotnie występowały w zadaniach z Olimpiady Informatycznej. Za pomocą podanych typów drzew możemy obsługiwać takie zapytania w złożoności czasowej $O(\log n)$. Do struktury danych będziemy dodawali elementy od prawej do lewej. Będziemy kolejno wykonywać polecenia $less(p_n), insert(p_n), less(p_{n-1}), insert(p_{n-1}), \ldots, less(p_1), insert(p_1)$. Odpowiedź na zapytanie $less(p_i)$ jest wartością $s(i)$.

Podsumowując, otrzymujemy rozwiązanie w złożoności czasowej $O(n \log n)$, które wystarczało do uzyskania kompletu punktów.

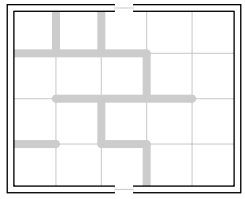
Tomasz Idziaszek Treść zadania | **Juliusz Straszyński** Opracowanie | **Marek Sommer** Program

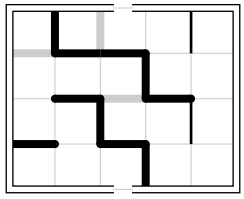
Dostępna pamięć: 128 MB. *OI, etap III, dzień pierwszy, 13.04.2016*

Żywopłot

Królewski ogrodnik Bajtazar ma za zadanie wyhodować w królewskim ogrodzie labirynt z żywopłotu. Ogród można podzielić na $m \times n$ kwadratowych pól. Otoczony jest murem, w którym na środku północnej i południowej ściany są wejścia. Na każdej krawędzi dzielącej dwa pola można zbudować kawałek żywopłotu – z cisu lub tui. Król bardziej lubi cis, więc chciałby mieć w swoim ogrodzie jak najwięcej kawałków żywopłotu z cisu. Niestety, cis wymaga lepszej gleby, więc nie wszędzie go można posadzić.

Aby żywopłot tworzył labirynt, musi spełniać dodatkowy warunek: do każdego pola musi być możliwość dojścia z obu wejść i, co więcej, tylko na jeden sposób. (Z danego pola można przejść bezpośrednio na pole sąsiadujące, jeśli na dzielącej te pola krawędzi nie znajduje się kawałek żywopłotu. Dwa sposoby dojścia uznajemy za różne, jeśli przechodzą przez różne zbiory pól.)





W lewej części powyższego rysunku przedstawiono przykładowy ogród dla $m = 4$ i $n = 5$, zawierający 31 krawędzi. Wyróżniono w nim 13 krawędzi, na których można posadzić żywopłot z cisu.

Na prawej części rysunku przedstawiono przykładowy labirynt składający się z 12 kawałków żywopłotu, z których 10 jest żywopłotem z cisu, a 2 są żywopłotem z tui. Nie istnieje labirynt zawierający więcej kawałków z cisu. Twoim zadaniem będzie napisanie programu, który pomoże Bajtazarowi w zaprojektowaniu labiryntu.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite m i n oznaczające rozmiar ogrodu ($2 \leqslant m, n$ oraz n jest liczbą nieparzystą). Kolejne m wierszy zawiera po $n-1$ znaków, opisujących pionowe krawędzie (czytane rzędami, od lewej do prawej). Znak `C` *oznacza, że na danej krawędzi można posadzić żywopłot z cisu, a znak* `T` *oznacza, że można posadzić żywopłot z tui. Kolejne $m-1$ wierszy zawierające po n znaków opisuje poziome krawędzie (również czytane rzędami, od lewej do prawej).*

Wyjście

W pierwszym wierszu standardowego wyjścia należy wypisać dwie liczby całkowite: liczbę posadzonych kawałków żywopłotu tworzących labirynt oraz maksymalną liczbę kawałków żywopłotu

z cisu. W kolejnych $2m-1$ wierszach należy opisać krawędzie labiryntu (w kolejności jak na wejściu). Należy wypisać znak `Z`*, jeśli krawędź zawiera żywopłot, lub znak* `.` *(kropka) w przeciwnym wypadku.*

Jeśli istnieje wiele rozwiązań spełniających warunki króla, należy wypisać dowolne z nich.

Przykład

Dla danych wejściowych:	*jednym z poprawnych wyników jest:*
`4 5`	`12 10`
`CCTT`	`Z..Z`
`TTCT`	`..Z.`
`TCTT`	`.Z.Z`
`TTCT`	`..Z.`
`CCCTT`	`.ZZ..`
`TCCCT`	`.Z.Z.`
`CTCTT`	`Z.Z..`

Wyjaśnienie do przykładu: *Dane wejściowe opisują ogród z lewej części rysunku; wynik opisuje labirynt z prawej części rysunku.*

Testy „ocen”:

`1ocen`: *$m = 4$, $n = 3$, w każdym miejscu można posadzić cis;*

`2ocen`: *$m = 100$, $n = 99$, na pionowych krawędziach można posadzić cis, na poziomych można posadzić tuję;*

`3ocen`: *$m = 1000$, $n = 999$, w każdym miejscu można posadzić tuję.*

Ocenianie

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

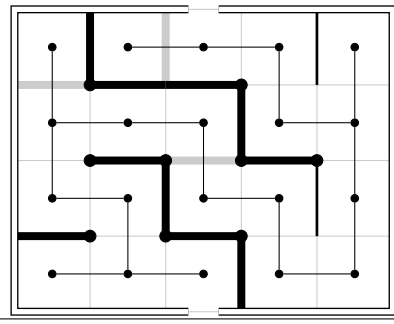
Jeśli Twój program wypisze poprawny pierwszy wiersz, a dalsza część wyjścia nie będzie poprawna, uzyska 52% punktów za dany test. W szczególności, aby uzyskać 52% punktów za test, wystarczy wypisać tylko jeden wiersz wyjścia.

Podzadanie	**Warunki**	**Liczba punktów**
1	$n \cdot m \leqslant 12$	*25*
2	$n, m \leqslant 100$	*25*
3	$n, m \leqslant 1000$	*50*

Rozwiązanie

Przypomnijmy definicję labiryntu: z każdego pola da się dojść do obu wejść na dokładnie jeden sposób. Sytuację z zadania możemy przedstawić jako graf nieskierowany, którego wierzchołki odpowiadają polom, a krawędzie łączą pary sąsiednich pól,

które *nie są* oddzielone kawałkiem żywopłotu. Graf taki jest więc labiryntem, gdy dla każdego wierzchołka istnieje dokładnie jedna ścieżka do wierzchołka oznaczonego wejściem oraz dokładnie jedna do wierzchołka oznaczonego wyjściem. Jest to tożsame z tym, że graf musi być spójny i bez cykli, bo każdy cykl w grafie spójnym oznaczałby istnienie takiego wierzchołka, który ma co najmniej dwie ścieżki do wejścia/wyjścia. Innymi słowy, graf musi być *drzewem* (rys. 1). Liczba wejść do labiryntu nie ma znaczenia – ta interpretacja byłaby prawdziwa również, gdybyśmy mieli jedno, czy też więcej niż dwa wejścia.



Rys. 1: Chcemy znaleźć takie drzewo, które połączy wszystkie pola i usunie z żywopłotu jak najmniej krawędzi z cisem.

Zatem musimy znaleźć drzewo łączące wszystkie wierzchołki o najmniejszym koszcie. Jeżeli w ostatecznym drzewie między sąsiednimi wierzchołkami jest krawędź, to nie posadzimy tam żywopłotu. W zadaniu chcemy mieć jak najwięcej kawałków żywopłotu, które są cisami. Oznacza to, że w drzewie chcemy użyć jak najmniej cisów, czyli koszt krawędzi odpowiadającej cisowi to 1, a krawędzi odpowiadającej tui to 0.

Zadanie sprowadza się więc do klasycznego problemu znajdowania minimalnego drzewa rozpinającego (ang. *MST – minimum spanning tree*). By je rozwiązać, możemy użyć znanych algorytmów znajdujących MST, np. algorytmu Kruskala [6] (`zyw2.cpp`, `zyw5.cpp`) czy też Prima [6] (`zyw3.cpp`). Dla grafu o zbiorze wierzchołków V i zbiorze krawędzi E algorytmy te działają w czasie $O((|V|+|E|)\cdot\log|V|)$. W naszym przypadku $|V| = nm$ i $|E| \leqslant 2nm$, więc ta złożoność to $O(nm\log(nm))$. Umiejętne użycie jednego z tych algorytmów pozwalało na zdobycie 100 punktów.

Nie są to optymalne rozwiązania pod względem złożoności czasowej. By je ulepszyć, należy wykorzystać strukturę grafu, a dokładnie to, że krawędzie mają dwa możliwe koszty: 0 lub 1.

Algorytm Prima zaczyna od jednowierzchołkowego drzewa i zachłannie rozszerza je o nowe wierzchołki, które aktualnie są najtańsze do dodania. Wykorzystując to, że mamy tylko dwa możliwe koszty krawędzi, możemy w kubełkach segregować krawędzie o odpowiednich kosztach, zamiast utrzymywać kosztowną kolejkę priorytetową. Dzięki takiej optymalizacji algorytm działa w czasie $O(|V|+|E|)$, czyli w naszym przypadku $O(nm)$ (`zyw12.cpp`).

W poniższym pseudokodzie zakładamy, że wszystkie krawędzie są początkowo wybrane do żywopłotu, a następnie usuwamy te z nich, które wybieramy do MST. Końce krawędzi oznaczamy jako $v1$ oraz $v2$.

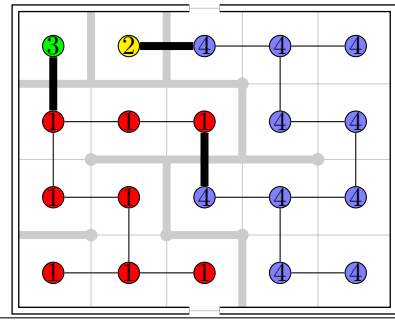
```
 1: procedure PRIM(v_0)
 2: begin
 3:   rozmiarDrzewa := 1;
 4:   dodany[v_0] := true;
 5:   krawedziePoCisach := {krawędzie z v_0 po cisach};
 6:   krawedziePoTujach := {krawędzie z v_0 po tujach};
 7:   while rozmiarDrzewa < n do begin
 8:     if not krawedziePoTujach.puste() then begin
 9:       najtanszaKrawedz := krawedziePoTujach.ostatnia();
10:       krawedziePoTujach.usunOstatnia();
11:     end else begin
12:       najtanszaKrawedz := krawedziePoCisach.ostatnia();
13:       krawedziePoCisach.usunOstatnia();
14:     end
15:     if dodany[najtanszaKrawedz.v1] then nowy := najtanszaKrawedz.v2
16:     else nowy := najtanszaKrawedz.v1;
17:     if not dodany[nowy] then begin
18:       usunZywoplot(najtanszaKrawedz);
19:       krawedziePoCisach += {krawędzie z nowego wierzchołka po cisach};
20:       krawedziePoTujach += {krawędzie z nowego wierzchołka po tujach};
21:       rozmiarDrzewa++;
22:       dodany[nowy] := true;
23:     end
24:   end
25: end
```

Algorytm Kruskala przedstawia się następująco: dopóki nie wszystko jest połączone w jedno drzewo, znajdź najtańszą krawędź, która łączy wierzchołki z niepołączonych jeszcze składowych, i dodaj ją do aktualnego minimalnego drzewa rozpinającego. Składowe są przechowywane za pomocą struktury danych do zbiorów rozłącznych, tzw. *Find-Union*: *Find* znajduje identyfikator składowej, do której należy dany wierzchołek, a *Union* łączy składowe zawierające podane dwa wierzchołki. Zamortyzowany koszt operacji *Find-Union* na zbiorze rozmiaru k to $O(\log^* k)$. Analogicznie jak w przypadku algorytmu Prima rozdzielając krawędzie według kosztów, jesteśmy w stanie sprawić, by algorytm Kruskala działał w czasie $O((|V| + |E|) \cdot \log^* |V|)$.

```
 1: procedure KRUSKAL
 2: begin
 3:   while not krawedzie.puste() do begin
 4:     najmniejsza := krawedzie.wezNajmniejsza();
 5:     krawedzie.usunNajmniejsza();
 6:     if Find(najmniejsza.v1) ≠ Find(najmniejsza.v2) then begin
 7:       usunZywoplot(najmniejsza);
 8:       Union(najmniejsza.v1, najmniejsza.v2);
 9:     end
10:   end
11: end
```

Okazuje się, że w przypadku naszego zadania także algorytm Kruskala możemy zmodyfikować, by działał w czasie liniowym (tj. $O(nm)$). Jako pierwsze w algorytmie Kruskala będą rozpatrzone krawędzie o koszcie 0. W takim razie chcemy podzielić wierzchołki grafu na takie grupy, że w obrębie jednej da się przejść między każdą parą wierzchołków wyłącznie po tujach. Można łatwo takie grupy znaleźć w czasie liniowym, np. za pomocą algorytmu DFS, startując z każdego nieodwiedzonego wierzchołka, odwiedzając wszystkie wierzchołki osiągalne po tujach i oznaczając je numerem grupy. Wierzchołki w obrębie danej grupy jesteśmy w stanie połączyć, nie usuwając z żywopłotu żadnego cisu. Natomiast by połączyć różne grupy nie mamy innego wyboru jak usunąć jakieś cisy na ich granicy (rys. 2). By połączyć k grup w drzewo, musimy postawić dokładnie $k-1$ krawędzi.

Znamy już odpowiedź, ile cisów zostanie w żywopłocie, ale teraz pytanie brzmi: jak szybko znaleźć te cisy, które zostaną usunięte? Możemy to osiągnąć, tworząc graf grup: iterujemy po każdym polu i jeżeli jego grupa to g_1, a grupa jego sąsiada to g_2 i $g_1 \neq g_2$, to g_1 i g_2 są sąsiadami w grafie grup, a krawędź między nimi odpowiada krawędzi między rozważanymi polami. W powstałym grafie znajdujemy dowolne drzewo rozpinające np. za pomocą algorytmu DFS. To rozwiązanie zostało zaimplementowane w pliku `zyw11.cpp`.



Rys. 2: Najpierw łączymy wierzchołki wewnątrz każdej z grup kosztem 0 (cienkie linie), a później łączymy grupy (grube linie).