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.

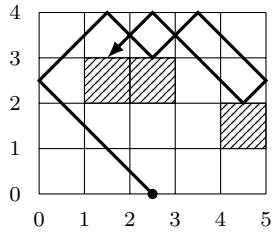
Wyjście

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.

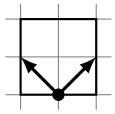
Najprostsze rozwiązanie polega na bezpośredniej symulacji ruchu piłeczki w kolejnych jednostkach czasu. Na początek spróbujmy zrobić to dla pustej planszy.

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.

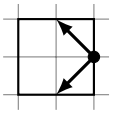
```
1:  function Ruszaj
2:  begin
3:    x := x + dx;
4:    y := y + dy;
5:    czas := czas + 1;
6:    if x = 0 or x = 2m then { odbicie od pionowego brzegu }
7:      dx := −dx
8:    else if y = 0 or y = 2n then { odbicie od poziomego brzegu }
9:      dy := −dy;
10: end
```

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.

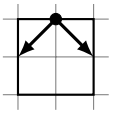
Każde pole ma 4 punkty, w których piłeczka może dotknąć jego brzegu (co, uwzględniając kierunki, daje 8 możliwych pozycji; patrz też rys. 1). Punkty, w których



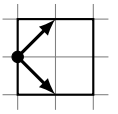
$(sx, sy-1, 1, 1)$
$(sx, sy-1, -1, 1)$



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



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



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

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
```

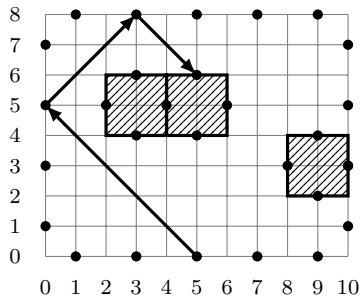
Główna pętla programu symulującego ruch piłeczki jest bardzo prosta. Inicjujemy $czas := 0$ oraz $(x, y, dx, dy) := (m, 0, -1, 1)$, a następnie wywołujemy funkcję Ruszaj dopóty, dopóki liczba niezbitych klocków k jest dodatnia.

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
```

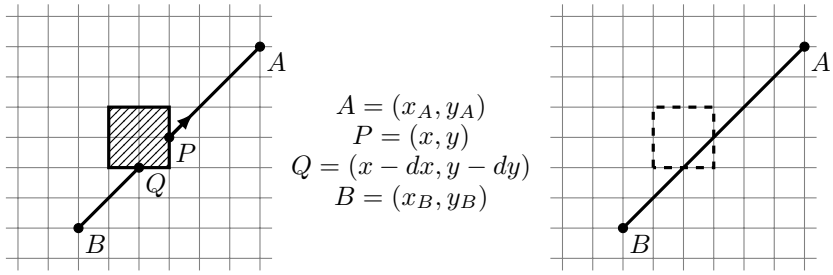
Przejdźmy teraz do kwestii budowania grafu, czyli wyznaczenia wartości *graf*. Załóżmy, że chcemy obliczyć wartość dla wierzchołka (x, y, dx, dy). Poruszamy się zatem z punktu (x, y) w kierunku (dx, dy), aż nie natrafimy na taką pozycję (nx, ny, dx, dy), za którą jest już ściana lub pole z klockiem. 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:

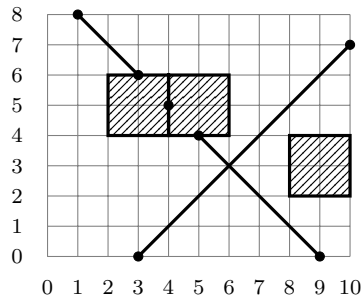
```
1: function UsunKlocek(x, y, dx, dy)
2: begin
3:     (sx, sy) := SrodekPola(x, y, dx, dy);
4:     klocek[sx, sy] := false;
5:     ScalKrawedz(sx + 1, sy, 1, 1);
6:     ScalKrawedz(sx, sy + 1, −1, 1);
7:     ScalKrawedz(sx − 1, sy, −1, −1);
8:     ScalKrawedz(sx, sy − 1, 1, −1);
9: end
```

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.

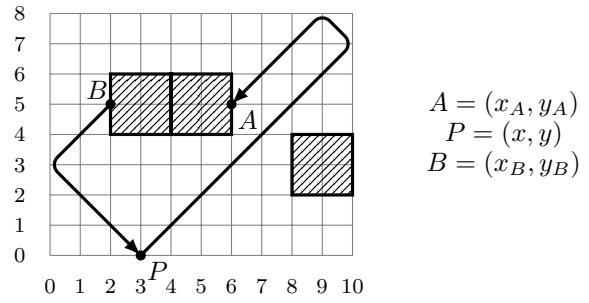
(x_2, y_2) są połączone krawędzią, to $x_1 - y_1 = x_2 - y_2$ (dla prostych nachylonych w prawo) lub $x_1 + y_1 = x_2 + y_2$ (dla prostych nachylonych w lewo). Ustalmy zatem pewną prostą, np. przechodzącą przez punkty (x, y), dla których $x + y = \text{const}$. Aby wyznaczyć krawędzie dla wierzchołków 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 $O(m+n+k)$.

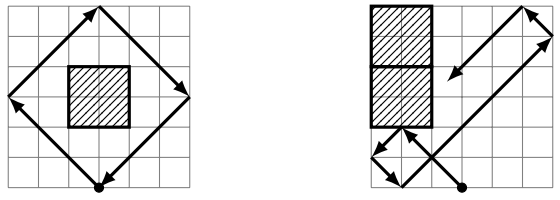
Dlaczego piłeczka zbije wszystkie klocki?

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.

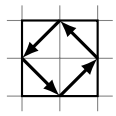
Rozważmy ruch piłeczki po pustej planszy. Piłeczka może poruszać się w nieskończoność, ale ponieważ plansza zawiera skończoną liczbę pozycji, to w pewnym momencie piłeczka drugi raz pojawi się w uprzednio odwiedzonej pozycji i dalej jej ruch będzie się powtarzał w cyklu.

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).

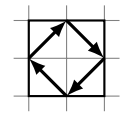


Rys. 6: Przykłady plansz 3×3, dla których piłeczka nie zbije wszystkich klocków.

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



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



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.

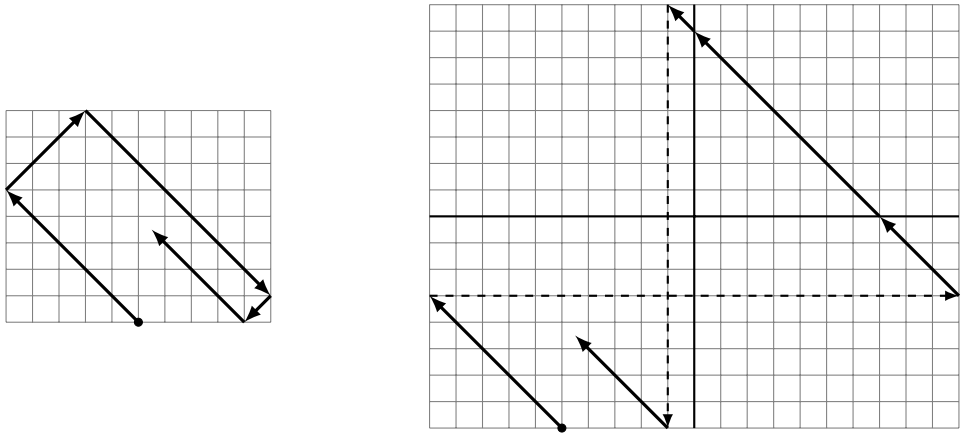
Jeszcze szybsza funkcja do numerów i dowód istnienia dokładnie jednego cyklu

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).