Bogdan S. Chlebus
treść zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Skok w bok

Plansza do gry w „Skok w bok" jest nieskończoną taśmą pól, nieograniczoną zarówno w lewo jak i w prawo. Na polach planszy stoją pionki. Ich liczba jest skończona. Na jednym polu może stać wiele pionków jednocześnie. Zakładamy, że pierwsze od lewej pole, na którym jest co najmniej jeden pionek, ma numer 0. Pola na prawo od niego są oznaczone kolejno liczbami naturalnymi 1, 2, 3 itd., a pola w lewo liczbami ujemnymi: -1, -2, -3 itd. Ustawienie pionków na taśmie, które będziemy także nazywać **konfiguracją**, *można opisać w ten sposób, że dla każdego pola, na którym jest co najmniej jeden pionek, podaje się numer pola i liczbę pionków na tym polu.*
Są dwa rodzaje ruchów zmieniających konfigurację: skok w prawo i skok w lewo.
Skok w prawo *polega na zabraniu po jednym pionku z wybranych dwóch sąsiednich pól o numerach p i $p+1$ i dodaniu jednego pionka na polu $p+2$.*
Skok w lewo: *zabieramy jeden pionek z pola $p+2$, a dodajemy po jednym na polach p i $p+1$.*
Mówimy, że konfiguracja jest **końcowa,** *jeśli na dowolnych dwóch sąsiednich polach znajduje się co najwyżej jeden pionek.*
Dla każdej konfiguracji istnieje dokładnie jedna konfiguracja końcowa, którą można z niej otrzymać w wyniku skończonej liczby ruchów w prawo lub w lewo.

Zadanie

Ułóż program, który:

- *wczytuje opis konfiguracji początkowej z pliku tekstowego SKO.IN;*
- *znajduje konfigurację końcową, do jakiej można doprowadzić daną konfigurację początkową i zapisuje wynik w pliku tekstowym SKO.OUT.*

Wejście

W pierwszym wierszu pliku SKO.IN jest zapisana jedna liczba całkowita dodatnia n. Jest to liczba niepustych pól danej konfiguracji początkowej. $1 \leqslant n \leqslant 10000$ (dziesięć tysięcy).
W każdym z kolejnych n wierszy znajduje się opis jednego niepustego pola konfiguracji początkowej w postaci pary liczb całkowitych oddzielonych odstępem. Pierwsza liczba to numer pola, a druga — to liczba pionków na tym polu. Te opisy są uporządkowane rosnąco względem numerów pól. Największy numer pola nie przekracza 10000 (dziesięć tysięcy), a liczba pionków na żadnym polu nie przekracza 10^8 (sto milionów).

Wyjście

W pierwszym wierszu pliku tekstowego SKO.OUT należy zapisać konfigurację końcową, do której można przekształcić daną konfigurację początkową — numery niepustych pól konfiguracji końcowej. Numery te powinne być uporządkowane rosnąco. Liczby w wierszu powinny być pooddzielane pojedynczym odstępem.

Przykład

Dla pliku tekstowego SKO.IN:

```
2
0 5
3 3
```

poprawnym rozwiązaniem jest plik SKO.OUT:

```
-4 -1 1 3 5
```

ROZWIĄZANIE

Opis rozwiązania zadania rozpoczniemy od kilku prostych uwag. Chociaż ruchy pionków w grze są opisane w terminach pojedynczych pionków (dla każdego pola gdzie występują zmiany liczby pionków), to efektywne rozwiązanie wymaga symulowania ruchów wieloma pionkami jednocześnie.

Zauważmy, że jeżeli na każdym niepustym polu leży co najwyżej jeden pionek, to taką konfigurację można przekształcić do końcowej wykonując tylko ruchy w prawo (proste ćwiczenie). Jeżeli natomiast wszystkie pionki leżą na jednym polu, to ruch w lewo jest konieczny. Stąd wynika, że algorytm musi przeplatać ciągi ruchów w prawo z ciągami ruchów w lewo.

Spodziewamy się, że zmiany konfiguracji prowadzące do zmniejszania się całkowitej liczby pionków mogą prowadzić do konfiguracji końcowych lub im bliskich. Ruch w prawo zmniejsza liczbę pionków, natomiast ruch w lewo zwiększa. Przed każdym wykonaniem ruchu w lewo musimy więc wykazać ostrożność, aby nie spowodować nadmiernego wzrostu liczby pionków, nawet przy lokalnym uproszczeniu konfiguracji, które w ogólnym rozliczeniu okazać się może pozorne.

JEDNOZNACZNOŚĆ KONFIGURACJI KOŃCOWEJ

Przypominamy pojęcie **liczb Fibonacciego** — tworzą one ciąg nieujemnych liczb całkowitych $\langle F_i \rangle$, dla $i \geq 0$, określony rekurencyjnie w następujący sposób:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_{i+2} = F_i + F_{i+1}$$

Określamy **wyrażenia** jako skończone sumy liczb Fibonacciego, zakładając, że te liczby zapisane są zawsze w kolejności niemalejących wskaźników. Rozważamy następujące operacje na takich wyrażeniach:

- F_{i+2} zastępujemy przez sumę $F_i + F_{i+1}$;
- sumę $F_i + F_{i+1}$ zastępujemy przez F_{i+2}.

Takie przekształcanie wyrażeń nie zmienia ich wartości.

Na związek między sumami liczb Fibonacciego a konfiguracjami pionków na planszy wskazuje naturalne podobieństwo między operacjami wykonania skoku w lewo lub w prawo, a operacjami wymiany podwyrażeń $F_i + F_{i+1}$ i F_{i+2}. Jeżeli od jednego wyrażenia można przejść do drugiego przez wykonanie skończonego ciągu takich operacji, to nazwiemy je **podobnymi**. Jeżeli od jednej konfiguracji można przejść do innej przez wykonanie skończonego ciągu ruchów, to także takie konfiguracje nazwiemy **podobnymi**. Chcielibyśmy konfiguracjom przypisać wyrażenia w taki sposób, żeby podobne konfiguracje odpowiadały podobnym wyrażeniom. Tu napotykamy na pewne trudności. Konfiguracje są jednoznacznie określone przez względne położenie pionków na planszy. Pola planszy są ponumerowane tylko w celu wygodnego opisu konfiguracji, natomiast liczby Fibonacciego są określone przez swoje wskaźniki. Innymi słowy: gdy konfigurację przesuniemy o jedno pole w prawo to będzie ona „taka sama", natomiast gdy w sumie liczb Fibonacciego zwiększymy każdy wskaźnik, to dostaniemy większą liczbę (poza przypadkiem wyrazeń F_1 i F_2). W opisie konfiguracji mogą pojawić się pola o numerach ujemnych, natomiast liczby Fibonacciego mają wskaźniki nieujemne. Największą trudnością logiczną jest jednak to, że wykonując odpowiednio wiele ruchów w lewo możemy utworzyć dowolnie wiele pionków na planszy!

Aby uniknąć takich trudności, najprościej rozważyć **grę pomocniczą**, bardzo podobną do właściwej gry „Skok w bok". Przyjmijmy, że plansza jest ciągiem pól nieskończonym tylko w jedną, prawą stronę. Pola są ponumerowane kolejnymi liczbami całkowitymi dodatnimi, począwszy od skrajnie lewego pola, które ma numer 1. Ruchy w prawo można wykonywać z każdego pola, natomiast ruchy w lewo tylko z pól o numerach nie mniejszych niż 3. Dodatkowo wprowadzamy regułę, że pionki między polami 1 oraz 2 można dowolnie przesuwać. Konfiguracji przyporządkowujemy liczbę $\sum_{i\geqslant 1} a_i \cdot F_i$, gdzie a_i jest liczbą pionków na polu i (suma jest skończona, ponieważ tylko skończenie wiele a_i jest niezerowych). Jest jasne, że takie przyporządkowanie sum liczb Fibonacciego konfiguracjom dla gry pomocniczej jest dobre w tym sensie, że konfiguracjom podobnym odpowiadają podobne wyrażenia.

Wiadomo że każdą liczbę całkowitą dodatnią n można jednoznacznie przedstawić w postaci sumy $\sum_{i>1} b_i \cdot F_i$, gdzie $b_i = 0$ poza skończoną liczbą indeksów i, oraz jeżeli $b_i \neq 0$, to $b_i = 1$ i $b_{i-1} = b_{i+1} = 0$ (patrz [13]). Takie sumy to wyrażenia, w których każda liczba Fibonacciego F_i, dla $i > 0$, występuje co najwyżej jeden raz i dwie liczby o sąsiednich indeksach nie występują jednocześnie. Stąd od razu wynika, że dla każdej konfiguracji istnieje dokładnie jedna podobna do niej konfiguracja końcowa w grze pomocniczej.

Jednoznaczność końcowej konfiguracji dla gry „Skok w bok" także zachodzi. Można to pokazać, odwołując się do powyższej własności liczb Fibonacciego (o jednoznaczności przedstawienia liczb naturalnych w postaci odpowiedniego wyrażenia). Szczegóły pozostawiamy jako ćwiczenie. Jako wskazówkę proponujemy takie ponumerowanie pól planszy aby pierwsze od lewej niepuste pole miało numer o tak dużej wartości, żeby w konfiguracji końcowej występowały tylko pola o dodatnich numerach.

ROZŁADOWYWANIE

Pole z co najmniej trzema pionkami nazywamy **wieżą**. Ciąg ruchów, który zmniejsza

liczbę pionków danej wieży, jednocześnie zmniejszając łączną liczbę pionków na planszy, nazywamy **rozładowaniem wieży**. Poniżej opisujemy jeden z takich sposobów.

Notacja

$$\ldots, a_1, a_2, \ldots, a_n, \ldots$$

oznacza konfigurację, w której na i-tym polu spośród n kolejnych, znajduje się a_i pionków. Wykonanie ruchu i przejście do następnej konfiguracji zapisujemy jako dwie konfiguracje przedzielone strzałką. Na przykład ruch w prawo możemy zapisać jako

$$\ldots, x+1, y+1, z\ldots \quad \Longrightarrow \quad \ldots, x, y, z+1, \ldots$$

Natomiast ruch w lewo zapisujemy jako

$$\ldots, x, y, z+1\ldots \quad \Longrightarrow \quad \ldots, x+1, y+1, z, \ldots$$

Rozważmy następujący ciąg trzech kolejnych ruchów:

$$\begin{aligned} \ldots, a, b, c+3, d, e, \ldots \quad &\Longrightarrow \quad \ldots, a+1, b+1, c+2, d, e\ldots \quad \Longrightarrow \\ &\Longrightarrow \quad \ldots, a+1, b, c+1, d+1, e\ldots \quad \Longrightarrow \\ &\Longrightarrow \quad \ldots, a+1, b, c, d, e+1\ldots \end{aligned}$$

Nazwiemy go **pojedynczym rozładowaniem**. Taka operacja usuwa trzy pionki z pola i umieszcza dwa na innych polach. Jeżeli $3k$ jest maksymalną wielokrotnością 3 nie większą niż liczba pionków na danym polu, to możemy zastosować jednocześnie operację pojedynczego rozładowania do każdej z k trójek, co nazwiemy też **rozładowaniem**. Taka operacja usuwa $3k$ pionków z rozważanego pola i umieszcza po k pionków na dwóch innych polach (razem $2k$).

ALGORYTM OBLICZANIA KONFIGURACJI KOŃCOWEJ

Konfiguracja reprezentowana jest jako skończona lista kolejnych pól. Możemy ograniczyć się tylko do niepustych pól, ale to zależy od szczegółów implementacji. Z polem wiążemy informację o liczbie pionków. Algorytm obliczania konfiguracji końcowej składa się z dwóch faz.

Faza 1

while istnieje pole zawierające co najmniej 3 pionki **do**
przejrzyj listę niepustych pól w kolejności od lewej do prawej
i rozładuj każde zawierające co najmniej 3 pionki

Po każdym rozładowaniu cofamy się o dwa pola, żeby sprawdzić czy nie pojawią się tam co najmniej trzy pionki, ale takich pól nie rozładowujemy w tej iteracji pętli.

Niech *pion* będzie funkcją zwracającą liczbę pionków w polu.

Faza 2

pole := numer skrajnie prawego niepustego pola;
pole := *pole* − 1;
while konfiguracja nie jest końcowa **do**
case
  $pion(pole) > 0$ i $pion(pole + 1) > 0$:
    wykonaj ruch w prawo z *pole*;
    pole := *pole* + 2;
  $pion(pole) = 2$ i $pion(pole - 1) > 0$:
    pole := *pole* − 1;
  $pion(pole) = 2$ i $pion(pole + 1) = 0$:
    wykonaj ruch w lewo z *pole*;
    wykonaj ruch w prawo z *pole* − 1;
    pole := *pole* + 1;
  $pion(pole) = 3$:
    rozładuj *pole*;
    pole := *pole* + 2;
  pole jest numerem skrajnie lewego niepustego pola:
    zakończ fazę 2.;
  w pozostałych przypadkach:
    pole := numer następnego niepustego pola z lewej;

Przyjmujemy, że warunki w konstrukcji **case** są sprawdzane po kolei, a po znalezieniu pierwszego spełnionego warunku wykonujemy stojącą przy nim instrukcję (i na tym wykonanie **case** się kończy).

POPRAWNOŚĆ ALGORYTMU

Faza 1. kiedyś się zakończy, bowiem każde rozładowanie pola zmniejsza łączną liczbę pionków na planszy. Faza 2. zaczyna się w sytuacji gdy na każdym polu leżą co najwyżej dwa pionki. Idziemy od strony prawej do lewej starając się, by na prawo była już sytuacja taka, jak w konfiguracji końcowej. Poprawność fazy 2 wynika z zachowania następujących niezmienników po każdej iteracji instrukcji **case**:

- co najwyżej jedno pole zawiera trzy pionki;
- na prawo od *pole* nie ma pola zawierającego więcej niż dwa pionki;
- jeżeli *pole* i *pole* + 1 są niepuste, to na prawo od *pole* nie ma już innych takich par sąsiednich pól.

Formalny dowód przebiega przez indukcję i polega na rozważeniu wszystkich możliwych przypadków.

Niezmienniki z instrukcji **case** pokazują, że nie nastąpi „spiętrzenie" pionków. Jednocześnie zauważmy, że w każdych czterech kolejnych iteracjach instrukcji **case**

albo następuje zmniejszenie łącznej liczby pionków albo *pole* przyjmuje wartość mniejszą od wszystkich dotychczasowych.

IMPLEMENTACJA ALGORYTMU

Listę pól pamiętamy w tablicy liczb całkowitych; elementami tablicy są liczby pionków stojących na polach. Tablica ma rozmiar taki, jak podany rozmiar planszy, powiększony z obu końców tak, by pomieścić pola potrzebne do trzymania wież powstałych przy rozładowywaniu wysokich wież znajdujących sie na skraju planszy w początkowej konfiguracji. Rozmiar powiększenia wyraża się logarytmem z maksymalnej dopuszczalnej na wejściu wysokości wieży pionków, ponieważ każde rozładowanie zmniejsza wysokość rozładowywanej wieży o jedną trzecią. W fazie 1 wieże, które pozostają jeszcze do rozładowania trzymamy w kolejce, implementowanej w tablicy w taki sposób, że wieże są pobierane w kolejności ich wstawiania. Program realizujący tę implementację znajduje się w pliku SKO.PAS.

INNE ROZWIĄZANIA

Można rozważać inne rozwiązania. Omówimy algorytm oparty na następującej zasadzie: wykonujemy ruchy w prawo, dopóki to jest możliwe, a gdy nie jest, to wykonujemy ruch w lewo. W takich sytuacjach ruch oznacza skok wieloma pionkami jednocześnie. Skok w prawo najlepiej wykonać maksymalną możliwą liczbą pionków jednocześnie. Przy skoku w lewo dobrze jest tak rozłożyć pionki, aby zapewnić sobie potem długi ciąg ruchów w prawo. Można to osiagnąć stosują zasadę „złotego podziału". Niech $\phi = (1+\sqrt{5})/2$. Przypuśćmy, że mamy konfigurację

$$\ldots, x, 0, w, 0 \ldots$$

którą nazywamy **izolowaną wieżą**, o ile $w > 1$. Chcielibyśmy wykonać ruch w lewo:

$$\ldots, x, 0, w, 0 \ldots \implies \ldots, x + a \cdot w, a \cdot w, (1-a) \cdot w, 0, \ldots$$

a następnie w prawo:

$$\ldots, a \cdot w, a \cdot w, (1-a) \cdot w, 0, \ldots \implies \ldots, a \cdot w, 0, (1-2a) \cdot w, a \cdot w, \ldots$$

i kontynuować ruchy w prawo. Najdłuższy możliwy ciąg ruchów w prawo otrzymamy, gdy iloraz wysokości dwóch sąsiednich wież zmienia się jak najmniej, to znaczy gdy dwa ułamki

$$\frac{a}{1-a} \quad \text{oraz} \quad \frac{1-2a}{a}$$

są jak najbliższe co do wartości. Rozwiązując równanie otrzymane przez przyrównanie do siebie ułamków dostajemy, że $a = 1 + \phi$. Szybka implementacja takiego algorytmu wygląda następująco. Zaczynając z prawej strony przesuwamy się w lewo szukając pierwszej możliwości skoku w prawo lub izolowanej wieży. Jeżeli skok w prawo jest

możliwy, wykonujemy serię takich skoków w prawo „do oporu". Jeżeli napotkamy na izolowaną wieżę, wykonujemy ruch w lewo zgodnie z zasadą złotego podziału, po czym kontynuujemy ruchy w prawo.

Omówimy także krótko pułapki tego zadania. Typowe błędy to:

- przyjęcie, że każda konfiguracja zawsze zmieści się w tablicy pamiętającej konfigurację początkową,
- niewłaściwe przeplatanie skoków w lewo ze skokami w prawo, tak, że doprowadza to do zapętlenia.

Rozwiązania poprawne ale bardzo czasochłonne też nie są dobre. Typowy błąd to wykonywanie skoków tylko jednym pionkiem na raz, co prowadzi do algorytmu o dużej złożoności czasowej. Ciekawszy przykład nieefektywnego rozwiązania to algorytm skonstruowany podobnie do metody złotego podziału, ale rozkładający wieże na trzy prawie równe części, co przy odpowiednio dobranych danych prowadzi do bardzo złego zachowania (ćwiczenie dla czytelnika).

TESTY

Testy do tego zadania znajdują się w plikach SKO0.IN–SKO12.IN. Można je podzielić na testy poprawności i testy złożoności. Testy poprawności:

- SKO0.IN — test z treści zadania;
- SKO1.IN — prosty test wymagający jedynie skoków w prawo;
- SKO2.IN — prosty test wymagający skoków w prawo i w lewo;
- SKO3.IN — konfiguracja końcowa tożsama wejściowej — bez skoków;
- SKO4.IN — prosty test z rozwiązaniem zawierającym pola o numerach ujemnych;
- SKO5.IN — dwie sąsiadujące ze sobą wieże Fibonacciego o wysokościach F_{37} i F_{38}.

Testy złożoności:

- SKO6.IN — pojedyncza wieża wysokości 10^8;
- SKO7.IN — dwie wieże wysokości 10^8 na polach K_0 i K_{10000};
- SKO8.IN — ciąg 10000 małych wież $3, 2, 2, \ldots, 2, 2, 3$;
- SKO9.IN — test losowy: ciąg 100 wież wysokości $\leqslant 10^2$;
- SKO10.IN — test losowy: ciąg 1000 wież wysokości $\leqslant 10^4$;
- SKO11.IN — test losowy: ciąg 10000 wież wysokości $\leqslant 10^8$;
- SKO12.IN — konfiguracja rzadka: 120 wież o wysokości 10^8 w odstępach co 80 pól.