Paweł Parys
Treść zadania, Opracowanie

Szymon Wrzyszcz
Program

Dostępna pamięć: 128 MB.

OI, Etap I, 20.10–17.11.2008

Kamyki

Jaś i Małgosia grają w „kamyki". Początkowo na stole ułożona jest pewna liczba kamyków, pogrupowanych w n kupek. Kupki znajdują się obok siebie w jednym rzędzie. Ustawienie kamieni spełnia dodatkowo własność, że na każdej kupce jest nie mniej kamieni niż na kupce sąsiadującej z nią po lewej (nie dotyczy to pierwszej kupki, na lewo od której nie ma już nic). Gracze na przemian usuwają z jednej, wybranej przez siebie w danym ruchu kupki dowolną dodatnią liczbę kamieni. Muszą to jednak robić tak, aby na tej kupce nie zostało mniej kamieni niż na kupce o jeden w lewo. Początkowa własność ustawienia zostaje więc zachowana. Gdy ktoś nie ma już możliwości wykonania ruchu (tzn. przed jego ruchem wszystkie kamienie są zdjęte ze stołu), to przegrywa. Jaś zawsze zaczyna.

Małgosia jest bardzo dobra w tę grę i jeśli tylko może, to gra tak, aby wygrać. Jaś prosi Cię o pomoc — chciałby wiedzieć, czy przy danym ustawieniu początkowym ma w ogóle szanse wygrać z Małgosią. Napisz program, który to określi.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita u ($1 \leqslant u \leqslant 10$), oznaczająca liczbę ustawień początkowych gry do przeanalizowania. W kolejnych $2u$ wierszach znajdują się opisy tych ustawień; każdy z nich zajmuje dokładnie dwa wiersze.

W pierwszym wierszu każdego opisu znajduje się jedna liczba całkowita n, $1 \leqslant n \leqslant 1\,000$ — liczba kupek kamieni. W drugim wierszu opisu znajduje się n nieujemnych liczb całkowitych a_i, pooddzielanych pojedynczymi odstępami i reprezentujących liczby kamieni na poszczególnych kupkach, od lewej do prawej. Liczby te spełniają nierówności $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_n$. Łączna liczba kamieni w żadnym opisie nie przekracza 10 000.

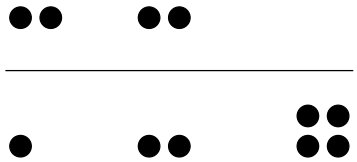
Wyjście

Na standardowe wyjście powinno zostać wypisanych u wierszy. W i-tym wierszu wyjścia (dla $1 \leqslant i \leqslant u$) powinno znajdować się słowo `TAK`*, jeśli Jaś może wygrać, zaczynając z i-tego na wejściu ustawienia początkowego, lub słowo* `NIE`*, jeśli Jaś zawsze przegra przy optymalnej grze Małgosi.*

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:
```
NIE
TAK
```

Rozwiązanie

Analiza problemu

W tym zadaniu, podobnie jak w wielu innych zadaniach o grach, należy określić, czy pierwszy gracz ma strategię wygrywającą, jeśli gra rozpoczyna się od danego ustawienia kamyków na kupkach. Takie ustawienia będziemy nazywali *pozycjami*; pozycja charakteryzowana jest przez liczby kamieni na poszczególnych kupkach.

Poczyńmy wpierw oczywiste spostrzeżenie: gracze są nierozróżnialni. Obojętnie czy z danego stanu rusza się Jaś, czy Małgosia, mają takie same możliwe ruchy do wykonania. Również warunek wygranej jest symetryczny: jeśli gracz nie ma do wyboru żadnego ruchu, to przegrywa, niezależnie czy jest to Jaś czy Małgosia. Ma więc sens mówienie o pozycjach wygrywających i przegrywających (dla aktualnego gracza). Pozycję nazywamy *wygrywającą*, gdy gracz rozpoczynający w niej grę ma strategię wygrywającą. W przeciwnym przypadku pozycję nazwiemy *przegrywającą*. Dla ścisłości dopowiedzmy jeszcze, że po każdym ruchu łączna liczba kamieni zmniejsza się co najmniej o jeden, więc gra kończy się zawsze po skończonej liczbie ruchów. Zatem pojęcie pozycji przegrywającej ma sens — nie tylko pierwszy gracz nie ma możliwości wygrania, zaczynając od niej, ale też drugi gracz może na pewno wygrać, czyli doprowadzić do przegranej pierwszego gracza. Zauważmy, że:

1. Pozycja końcowa (wszystkie kupki puste) jest przegrywająca.
2. Jeżeli z danej pozycji istnieje ruch prowadzący do pozycji przegrywającej, to pozycja ta jest wygrywająca (jeśli tak się ruszymy, to przeciwnik przegra).
3. Jeżeli z danej pozycji wszystkie ruchy prowadzą do pozycji wygrywających, to pozycja ta jest przegrywająca (cokolwiek zrobimy, przeciwnik wygra).

Warunek 1 wynika z warunku 3, ale został dodany dla zwiększenia czytelności.

Rozwiązanie wykładnicze

Powyższe rozważania prowadzą bezpośrednio do rozwiązania, w którym sprawdzamy, czy dana pozycja początkowa jest wygrywająca, korzystając wprost z powyższych warunków. Sprowadza się ono do prostej funkcji rekurencyjnej parametryzowanej pozycją. Poprawność rozwiązania jest oczywista. Po chwili zastanowienia widzimy, że czas działania jest wykładniczy względem rozmiaru pozycji, czyli łącznej liczby kamyków. Rozwiązanie takie, zaimplementowane w plikach `kams1.cpp`, `kams3.pas` i `kams4.java`, przechodziło tylko jeden test.

Powyższe rozwiązanie można trochę poprawić poprzez spamiętywanie wyników dla już rozpatrzonych pozycji. Jest to znacząca optymalizacja, gdyż wiele sekwencji ruchów może

prowadzić do tego samego układu. Jednak liczba pozycji osiągalnych z pozycji początkowej nadal jest wykładnicza, więc to rozwiązanie również ma szansę przejść jedynie małe testy (konkretnie przechodziło ono trzy pierwsze testy). Implementacje takiego rozwiązania można znaleźć w plikach `kams2.cpp` i `kams5.java`.

Rozwiązanie wzorcowe

Aby rozwiązać nasze zadanie w sensownej złożoności czasowej, potrzebujemy łatwiej obliczalnego kryterium pozwalającego określać, czy pozycja jest wygrywająca, czy przegrywająca. W przypadku zadań podobnych do tego często udaje się przypisać każdej pozycji p pewną (łatwą do obliczenia) liczbę $value(p)$ (nazywaną *wartością*) taką, że:

1. Wartość pozycji końcowej wynosi 0.
2. Jeśli wartość jakiejś pozycji jest niezerowa, to istnieje z niej ruch prowadzący do pozycji o wartości 0.
3. Jeśli wartość jakiejś pozycji p jest równa zeru, to każda pozycja, do której prowadzi ruch z p, ma wartość niezerową.

Przypuśćmy, że dla naszej gry udałoby się zdefiniować taką funkcję *value*. Porównajmy powyższe punkty z własnościami 1-3 pozycji wygrywających i przegrywających. Widzimy, że pozycje przegrywające to dokładnie te, których wartość wynosi 0. Formalnie możemy dowieść tego indukcyjnie po maksymalnej liczbie ruchów do końca gry z danej pozycji. Dostajemy więc bardzo proste rozwiązanie zadania: wystarczy obliczyć wartość pozycji początkowej; jeśli wychodzi 0, odpowiadamy „`NIE`”, w przeciwnym przypadku „`TAK`”.

Pozostaje zdefiniować funkcję *value*. Niech a_i oznacza liczbę kamieni na i-tej kupce. Dla ułatwienia zapisu przyjmijmy, że liczba kupek n jest parzysta (jeśli jest nieparzysta, to na początek możemy wstawić dodatkową kupkę zawierającą 0 kamieni i nic to nie zmieni w grze). Do definicji wartości często (w tym przypadku także) używa się operacji xor (oznaczenie: $\oplus$) na liczbach całkowitych, czyli sumy modulo 2 na bitach (cyfrach w zapisie dwójkowym) tych liczb. Innymi słowy, liczba $a \oplus b$ ma jedynki w zapisie dwójkowym na tych pozycjach, na których dokładnie jedna spośród liczb a, b ma jedynkę. Przykładowo:

$$12 \oplus 10 = [01100]_2 \oplus [01010]_2 = [00110]_2 = 6.$$

Zauważmy, że operacja xor jest przemienna ($a \oplus b = b \oplus a$), łączna ($(a \oplus b) \oplus c = a \oplus (b \oplus c)$) oraz odwrotna do samej siebie ($a \oplus b \oplus b = a$). Funkcję *value* określimy jako

$$(a_2 - a_1) \oplus (a_4 - a_3) \oplus \ldots \oplus (a_n - a_{n-1}). \tag{1}$$

Bierzemy więc xor-a wszystkich różnic liczb kamieni kolejnych par kupek o numerach parzystym i nieparzystym. Zauważmy, że w wyniku otrzymujemy liczbę całkowitą nieujemną.

Dlaczego taka funkcja spełnia wymagane własności? Własność 1 jest oczywista. Uzasadnienie własności 3 także nie jest trudne. Przypuśćmy, że wartość jakiejś pozycji wynosi 0. Po wykonaniu jednego ruchu zmniejszy się rozmiar dokładnie jednej kupki, zatem zmieni się dokładnie jedna różnica będąca argumentem naszego xor-a. Zauważmy, że jeśli zmieniamy dokładnie jeden argument xor-a z s na s', to wartość całego xor-a także na pewno się zmieni (konkretnie: z 0 na $s \oplus s' \neq 0$).

Aby udowodnić własność 2, przypuśćmy, że wartość w pewnej pozycji jest niezerowa i wynosi x. Niech i będzie numerem najbardziej znaczącego bitu x, który jest równy 1. Wówczas przynajmniej jeden spośród argumentów naszego xor-a ma i-ty bit równy 1. Oznaczmy pewien taki argument przez s (jest to $a_j - a_{j-1}$ dla pewnego parzystego j). Wówczas xor pozostałych argumentów jest równy $x \oplus s$. W liczbie tej i-ty bit jest równy 0, a wszystkie bardziej znaczące bity są równe tym w liczbie s, gdyż w x bity te są wszystkie wyzerowane. Stąd $x \oplus s < s$. Jeśli w najbliższym ruchu zmienimy s na $s' = x \oplus s$, to nowa wartość wyniesie:

$$x \oplus s \oplus s' = x \oplus s \oplus x \oplus s = 0.$$

Zauważmy, że taki ruch rzeczywiście zawsze istnieje. Jest to bowiem ruch, w którym zmniejszamy różnicę $s = a_j - a_{j-1}$. Różnicę tę można zmniejszyć o dowolną niezerową (i nie większą niż s) liczbę poprzez zdjęcie właśnie tylu kamieni ze stosu a_j. To kończy uzasadnienie faktu, że nasza definicja (1) wartości pozycji spełnia własności 1-3.

Pozostaje wciąż pytanie, w jaki sposób można dojść do powyższej definicji funkcji *value*? Trudno opisać to ściśle, ale spróbujemy przedstawić pewne intuicje. Zauważmy najpierw, że dowód powyższych własności opierał się praktycznie tylko na własnościach operacji xor, a w bardzo niewielkim stopniu na tym, co konkretnie xor-ujemy. Aby punkty 1 i 3 były spełnione, wystarczyłoby xor-ować cokolwiek, byleby każdy a_i występował w dokładnie jednym argumencie xor-a (na przykład można by wziąć xor wszystkich a_i) — wówczas zmiana a_i zmienia wartość. Aby zapewnić własność 2, wystarczy (jak już widzieliśmy), aby jakiś legalny ruch zmniejszał dowolny (tj. każdy możliwy) argument xor-a o dowolną wartość. Gdybyśmy po prostu xor-owali wszystkie a_i, to tak by nie było: a_i możemy zmniejszać tylko do a_{i-1}, a nie do zera. Tym co można zmniejszać do dowolnej nieujemnej wartości, są właśnie różnice. Wreszcie wspomniana własność występowania a_i w dokładnie jednym argumencie wymusza, aby brać tylko co drugą różnicę.

Rozwiązanie wzorcowe zostało zaimplementowane w plikach `kam.cpp`, `kam1.pas`, `kam2.java`. Jego złożoność czasowa jest liniowa względem n, więc może ono działać w rozsądnym czasie nawet dla liczby kupek rzędu 1 000 000; liczba kamieni na każdej kupce mogłaby bez problemu być ograniczona przez 10^9. W zadaniu obrano niższe limity na dane wyłącznie celem zmylenia zawodników. Autor zadania oraz jurorzy Olimpiady nie przypuszczają, aby istniało jakieś rozwiązanie (np. zachłanne lub dynamiczne), które działałoby tylko dla tak zmniejszonych limitów.

Gra „Staircase Nim”

Zadanie *Kamyki* ma rozwiązanie nieco podobne jak zadanie *Gra* z XI Olimpiady Informatycznej. W opisie rozwiązania tamtego zadania w [11] została zdefiniowana gra „Staircase Nim”. Występują w niej schody złożone z pewnej liczby stopni, które są numerowane kolejnymi liczbami naturalnymi, przy czym najniższy stopień ma numer 1. Na każdym stopniu znajduje się pewna liczba kamieni. W grze bierze udział dwóch graczy, którzy wykonują ruchy naprzemiennie. Ruch polega na przełożeniu dowolnej (dodatniej) liczby kamieni z dowolnie wybranego stopnia na stopień o numerze o jeden mniejszym (jeśli wybrano stopień o numerze 1, to kamienie są usuwane z dalszej gry). Gracz, który nie może wykonać ruchu (na żadnym stopniu nie ma już kamieni), przegrywa. Dana pozycja gry „Staircase Nim” jest przegrywająca wtedy i tylko wtedy, gdy xor liczby kamieni na

stopniach o nieparzystych numerach jest równy zeru (dowód tego faktu można znaleźć we wspomnianym już opracowaniu zadania *Gra* w książeczce XI OI).

Oznaczmy $r_i = a_i - a_{i-1}$ dla $1 \leqslant i \leqslant n$, przyjmując $a_0 = 0$. Mając dane wszystkie r_i, możemy łatwo odzyskać wyjściowy ciąg a_i, gdyż $a_i = \sum_{j=1}^{i} r_j$ dla każdego $i = 1, 2, \ldots, n$. Wynika stąd, że obie reprezentacje stanu gry są równoważne. Warunek $a_{i-1} \leqslant a_i$ przekłada się na $r_i \geqslant 0$. Wykonanie ruchu polega na wybraniu numeru kupki i oraz zdjęciu z niej pewnej liczby k kamieni. W wyniku takiej operacji a_i zmniejsza się o k. Przy reprezentacji układu ciągiem r, taki ruch odpowiada zmniejszeniu r_i o k i zwiększeniu r_{i+1} o k (o ile $i \leqslant n-1$), co można sobie wyobrazić jako przerzucenie k kamieni z i-tego na $(i+1)$-szy stopień. Zauważmy, że otrzymana gra jest równoważna grze „Staircase Nim" z „odwróconymi schodami" (tzn. najniższy stopień ma numer n, a najwyższy — numer 1). Tym samym rozwiązanie sprowadza się do wykonania xor-owania r_n, r_{n-2}, itd. Pozycja jest przegrywająca wtedy i tylko wtedy, gdy w ten sposób otrzymamy 0.

Testy

Zadanie było sprawdzane za pomocą 14 testów, z których każdy składa się z dziesięciu układów do rozwiązania. Wszystkie testy zostały wygenerowane (do jakiegoś stopnia) losowo. W każdym zestawie występuje co najmniej jeden test z odpowiedzią `TAK` i co najmniej jeden z odpowiedzią `NIE`. Liczby układów wygrywających i przegrywających w ramach jednego testu są podobne, mimo że losowy układ ma dużo większe prawdopodobieństwo bycia wygrywającym (fakt stwierdzony eksperymentalnie). Poniżej podajemy rozmiary poszczególnych testów, kolumna n_{max} to maksymalna (po wszystkich układach w teście) liczba kupek, a s_{max} to maksymalna łączna liczba kamieni.

Nazwa	n_{max}	s_{max}	Opis
kam1.in	5	10	bardzo małe dane
kam2.in	7	25	małe dane
kam3.in	12	47	małe dane
kam4.in	15	74	
kam5.in	17	121	
kam6.in	80	10000	dużo kamieni
kam7.in	153	155	
kam8.in	302	4582	
kam9.in	450	8812	
kam10.in	545	10000	
kam11.in	767	4364	
kam12.in	930	10000	
kam13.in	1000	7690	
kam14.in	1000	10000	maksymalne dane