Marek Sokołowski
Treść zadania, Opracowanie

Paweł Parys
Program

Dostępna pamięć: 256/64 MB.

OI, etap I, 19.10–16.11.2015

Nim z utrudnieniem

Ulubioną rozrywką Alicji i Bajtazara jest gra Nim. Do gry potrzebne są żetony, podzielone na kilka stosów. Dwaj gracze na przemian zabierają żetony ze stosów – ten, na którego przypada kolej, wybiera dowolny stos i usuwa z niego dowolną dodatnią liczbę żetonów. Gracz, który nie może wykonać ruchu, przegrywa[1].

Alicja zaproponowała Bajtazarowi kolejną partyjkę Nima. Aby jednak tym razem uczynić grę ciekawszą, gracze umówili się między sobą na dodatkowe warunki. Żetony, których było m, Alicja podzieliła na n stosów o licznościach $a_1, a_2, \ldots, a_n$. Zanim rozpocznie się rozgrywka, Bajtazar może wskazać niektóre spośród stosów, które zostaną natychmiast usunięte z gry. Liczba usuniętych stosów musi być jednak podzielna przez pewną ustaloną liczbę d, a ponadto Bajtazar nie może usunąć wszystkich stosów. Potem rozgrywka będzie toczyć się już normalnie, a rozpocznie ją Alicja.

Niech k oznacza liczbę sposobów, na które Bajtazar może wskazać stosy do usunięcia tak, aby mieć pewność, że wygra partię niezależnie od posunięć Alicji. Twoim zadaniem jest podanie reszty z dzielenia k przez $10^9 + 7$.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie dodatnie liczby całkowite n i d oddzielone pojedynczym odstępem, oznaczające odpowiednio liczbę stosów i ograniczenie „podzielnościowe" zabieranych stosów.

Drugi wiersz opisuje stosy i zawiera ciąg n dodatnich liczb całkowitych $a_1, a_2, \ldots, a_n$ pooddzielanych pojedynczymi odstępami, gdzie a_i oznacza liczbę żetonów na i-tym stosie.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, równą liczbie sposobów (modulo $10^9 + 7$), na które Bajtazar może usunąć stosy tak, aby później na pewno zwyciężyć.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
5 2 1 3 4 1 2	2

Wyjaśnienie do przykładu: *Bajtazar może zabrać 2 lub 4 stosy. Wygra tylko wtedy, gdy zabierze stosy o licznościach 1 i 4 (może to zrobić na dwa sposoby).*

[1] W Internecie łatwo znaleźć więcej informacji na temat gry Nim, a w szczególności opis strategii wygrywającej w tej grze.

Testy „ocen”:

`1ocen`: *$n = 9$, $d = 2$, wynikiem jest 0,*

`2ocen`: *$n = 12$, $d = 4$,*

`3ocen`: *$n = 30$, $d = 10$, wszystkie stosy mają po 30 żetonów,*

`4ocen`: *$n = 500\,000$, $d = 2$, wszystkie stosy mają wysokość 1.*

Ocenianie

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

We wszystkich podzadaniach zachodzą warunki $n \leqslant 500\,000$, $d \leqslant 10$, $a_i \leqslant 1\,000\,000$. Ponadto sumaryczna liczba żetonów $m = a_1 + a_2 + \ldots + a_n$ jest nie większa niż 10 000 000. Zwróć uwagę, że limit pamięci jest różny dla różnych podzadań.

Podzadanie	Dodatkowe warunki	Limit pamięci	Liczba punktów
1	$n \leqslant 20$, $a_1, \ldots, a_n \leqslant 1000$	*256 MB*	*10*
2	$n \leqslant 10\,000$, $a_1 \ldots, a_n \leqslant 1000$	*256 MB*	*18*
3	$d \leqslant 2$	*256 MB*	*25*
4	*brak*	*256 MB*	*27*
5	*brak*	*64 MB*	*20*

Rozwiązanie

W zadaniu prosiliśmy Uczestników o pomoc Bajtazarowi w wygraniu niewielkiej modyfikacji gry Nim. Aby więc móc rozwiązać ten problem, powinniśmy najpierw poznać kilka informacji na temat tej dość znanej gry matematycznej.

Teoria gier dla początkujących

Najpierw potrzebujemy definicji funkcji alternatywy wykluczającej (`xor`); dalej będziemy oznaczać ją $\oplus$. Aby wyznaczyć $a \oplus b$, czyli wartość `xor` liczb całkowitych a oraz b, zapisujemy obie liczby w systemie binarnym, a następnie dodajemy je pisemnie – z tą różnicą, że nie wykonujemy przeniesień. Poniższy przykład opisuje wykonanie operacji `xor` dla liczb $9 = 1001_{(2)}$ oraz $19 = 10011_{(2)}$:

$$\begin{array}{cccccccc} & & & 1 & 0 & 0 & 1 & \\ \oplus & 1 & 0 & 0 & 1 & 1 & & \\ \hline & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & = 26_{(10)} & \end{array}$$

Odpowiada to oczywiście wstawieniu jedynki w danej kolumnie wyniku, gdy znajdowało się w niej nieparzyście wiele jedynek. W przeciwnym przypadku wstawiamy zero.

Funkcję tę można uogólnić na więcej liczb całkowitych – dokładnie w ten sam sposób definiujemy `xor` wielu liczb $a_1 \oplus a_2 \oplus \ldots \oplus a_n$. Łatwo zauważyć, że działanie $\oplus$ jest łączne oraz przemienne, zatem nie ma znaczenia, w jakiej kolejności obliczamy `xor` wielu liczb.

Operacja $a \oplus b$ jest wbudowana w dostępne języki programowania: w C/C++ jest to `a ^ b`, zaś w Pascalu `a xor b`.

Kto wygra Nima?

Okazuje się, że istnieje bardzo prosty sposób na sprawdzenie, kto wygra rozgrywkę Nima:

Twierdzenie 1. *Niech $a_1, a_2, \ldots, a_n$ będą liczbami żetonów na kolejnych stosach w grze Nim. Gracz rozpoczynający grę może wygrać przy optymalnej grze przeciwnika wtedy i tylko wtedy, gdy $a_1 \oplus a_2 \oplus \ldots \oplus a_n \neq 0$.*

Dowód tego twierdzenia można znaleźć w Internecie, jak i w opracowaniu zadania *Kamyki* z XVI Olimpiady Informatycznej [1].

Jako że w naszym problemie Bajtazar jest drugim graczem, dąży on do tego, by na początku właściwej rozgrywki `xor` wszystkich wysokości stosów wynosił 0. Tak więc zadanie sprowadza się do tego, by policzyć liczbę sposobów usunięcia pewnej liczby stosów z rozgrywki tak, aby:

- liczba usuniętych stosów była podzielna przez d ($d \leqslant 10$),
- pozostał co najmniej jeden stos,
- `xor` wysokości wszystkich stosów wynosił 0.

Rozwiązanie przeglądające wszystkie możliwe podzbiory usuniętych stosów i sprawdzające, czy wszystkie trzy warunki zachodzą, zostało zaimplementowane w pliku `nims12.cpp`. Działa w czasie $O(2^n)$ i przechodzi testy w pierwszym podzadaniu.

Programowanie dynamiczne

Znacznie lepsze rozwiązanie można uzyskać, stosując metodę programowania dynamicznego. Zauważmy, że drugi warunek z poprzedniego podrozdziału można łatwo usunąć, ponieważ wpływa on na wynik tylko wtedy, gdy łączna liczba stosów n jest podzielna przez d. Jeśli zignorujemy drugi warunek, otrzymamy wówczas wynik o 1 za duży (wliczymy bowiem do wyniku sytuację, w której na początku usuniemy wszystkie stosy; jest ona wygrywająca dla Bajtazara). Możemy więc wymazać drugi warunek, a potem na samym końcu w razie potrzeby odjąć od wyniku 1.

Teraz próbujmy dodawać stosy po jednym – dla przykładu, od lewej do prawej. W każdym momencie decydujemy, czy pozostawić stos na planszy, czy go usunąć. Zauważmy, że w każdym momencie wystarczy nam pamiętać jedynie resztę z dzielenia przez d liczby usuniętych stosów oraz `xor` pozostawionych do tej pory stosów.

Mamy więc już szkielet programowania dynamicznego. Przez $dp[i][r][x]$ oznaczymy liczbę sposobów usunięcia spośród pierwszych i stosów podzbioru stosów o liczności

dającej resztę r z dzielenia przez d w taki sposób, że pozostałe z tych i stosów mają `xor` równy x. Jeśli oznaczymy przez A najmniejszą potęgę dwójki przekraczającą największą z wysokości stosów, to zachodzi $0 \leqslant i \leqslant n$, $0 \leqslant r < d$, $0 \leqslant x < A$. Aby nie musieć operować na dużych liczbach, w zadaniu jesteśmy proszeni jedynie o podanie reszty z dzielenia wyniku przez $M = 10^9 + 7$. Wobec tego w tablicy dp wystarczy pamiętać jedynie wartości modulo M.

Dla zerowej liczby stosów ($i = 0$) wszystko jest proste – musi zajść $r = 0$ (usuwamy zero stosów) oraz $x = 0$ (`xor` zerowej liczby stosów to 0). Tak więc $dp[0][0][0] = 1$ oraz $dp[0][\star][\star] = 0$ dla pozostałych wartości w tablicy.

Przypuśćmy, że rozważyliśmy do tej pory $i-1$ stosów i dokładamy i-ty, o wysokości a_i. Chcemy obliczyć $dp[i][r][x]$. Mamy dwie możliwości:

1. Zachowujemy i-ty stos. Liczba usuniętych stosów w porównaniu do poprzedniego stanu pozostaje niezmieniona. Natomiast poprzedni `xor` wysokości pozostawionych stosów musiał wynosić $x \oplus a_i$ (teraz dokładamy do poprzedniego `xor`-a wartość a_i i musi wyjść x; mamy zaś $(x \oplus a_i) \oplus a_i = x \oplus (a_i \oplus a_i) = x$). Do wyniku dodajemy liczbę sposobów dojścia do poprzedniego stanu równą $dp[i-1][r][x \oplus a_i]$.

2. Odrzucamy i-ty stos. Liczba usuniętych stosów krok wcześniej była o jeden mniejsza, za to `xor` się nie zmienił. Dodajemy więc do wyniku wartość $dp[i-1][(r-1) \bmod d][x]$. Zakładamy tożsamość $(-1) \bmod d = d-1$, która jest prawdziwa w matematyce, ale nie zachodzi w większości języków programowania.

Ostatecznie więc

$$dp[i][r][x] = (dp[i-1][r][x \oplus a_i] + dp[i-1][(r-1) \bmod d][x]) \bmod M.$$

Odpowiedź odczytujemy jako $dp[n][0][0]$ (rozpatrzyliśmy n stosów, wyrzuciliśmy liczbę stosów podzielną przez d, pozostałe stosy mają `xor` równy 0). W razie potrzeby odejmujemy od wyniku 1.

Złożoność czasowa i pamięciowa takiego rozwiązania wynosi $O(ndA)$. Niestety, proste obliczenia prowadzą do wniosku, że tablica dp zajmie 390 MB pamięci w drugim podzadaniu. Można jednak mocno ograniczyć zużycie pamięci na jeden z dwóch poniższych sposobów:

1. Zauważmy, że $dp[i][\star][\star]$ zależy tylko od $dp[i-1][\star][\star]$. Możemy więc utrzymywać w pamięci tylko dwie ostatnie podtablice: $dp[i][\star][\star]$ i $dp[i-1][\star][\star]$. Po obliczeniu nowej podtablicy możemy nadpisać poprzednią, ponieważ nie będzie ona nam już do niczego potrzebna.

   Metodę tę można zaimplementować, zmniejszając pierwszy wymiar do 2 elementów i utrzymując w nim resztę z dzielenia wartości i przez 2. Wtedy zamiast dokonywać przejścia $dp[i-1][\star][\star] \to dp[i][\star][\star]$, zerujemy podtablicę $dp[i \bmod 2][\star][\star]$ i wykonujemy przejście $dp[(i-1) \bmod 2][\star][\star] \to dp[i \bmod 2][\star][\star]$. Wynik znajdziemy w pozycji $dp[n \bmod 2][0][0]$.

2. Druga obserwacja jest bardziej wnikliwa. Na początek przypomnijmy, że $(x \oplus a_i) \oplus a_i = x$. Wartości $dp[i][\star][x]$ oraz $dp[i][\star][x \oplus a_i]$ zależą tylko od

$dp[i-1][\star][x]$ i $dp[i-1][\star][x \oplus a_i]$. Możemy więc wykonywać obliczenia niezależnie dla rozłącznych par $(x, x \oplus a_i)$, każdorazowo tworząc dwuwymiarową tablicę dp' o wymiarach $d \times 2$ i następie nadpisując fragment oryginalnej tablicy. W ten sposób możemy całkowicie zignorować pierwszy wymiar tablicy (i).

Obie metody ograniczają zapotrzebowanie na pamięć do $O(dA)$. Implementacja drugiego sposobu znajduje się w pliku `nims3.cpp`. Wystarcza do zdobycia punktów za dwa pierwsze podzadania.

Rozwiązanie wzorcowe

W rozwiązaniu wzorcowym skorzystamy w końcu z ograniczenia $m \leqslant 10^7$ na sumę wysokości stosów $a_1 + a_2 + \ldots + a_n$. Wykorzystamy następujący fakt:

Fakt 1. *Niech* $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_i$. *Wtedy* $a_1 \oplus a_2 \oplus \ldots \oplus a_i < 2a_i$.

Dowód: Wybierzmy potęgę dwójki 2^t taką, że $2^t \leqslant a_i < 2^{t+1}$. Ze względu na to, że wszystkie `xor`-owane liczby są mniejsze niż potęga dwójki 2^{t+1} (gdyż w szczególności nie przekraczają a_i), to ich `xor` też jest mniejszy niż 2^{t+1}. Jednak $2a_i \geqslant 2^{t+1}$. To kończy dowód. ■

Będziemy postępować następująco: posortujmy wszystkie wysokości stosów. Teraz $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_n$. W momencie, gdy liczymy $dp[i][\star][\star]$, znajdujemy potęgę dwójki 2^t taką, że $2^t \leqslant a_i < 2^{t+1}$, a następnie liczymy wszystkie wartości $dp[i][r][x]$ tylko dla $0 \leqslant x < 2^{t+1}$. Nie jesteśmy w stanie uzyskać $x \geqslant 2^{t+1}$ za pomocą dotychczas rozważonych stosów $a_1, \ldots, a_i$, więc dla większych x wartości dp będą równe 0.

Czas obliczenia $dp[i][\star][\star]$ nie przekroczy $2da_i$, ponieważ na podstawie powyższego faktu liczymy maksymalnie $2a_i$ podtablic $dp[i][\star][x]$, zaś czas obliczenia pojedynczej to $O(d)$. Łączny czas działania programowania dynamicznego możemy więc oszacować z góry przez

$$d \cdot (2a_1 + 2a_2 + \ldots + 2a_n) = 2dm = O(dm).$$

Do tego dochodzi czas sortowania wysokości stosów ($O(n \log n)$) oraz zaniedbywalny czas wyznaczania odpowiednich potęg dwójki 2^{t+1}.

Złożoność obliczeniowa jest już w zupełności wystarczająca do zaliczenia wszystkich testów, musimy jednak znów poradzić sobie z problemem ograniczonej pamięci. Możemy zaprząc do tego zadania te same sposoby, które omawialiśmy przy poprzednim rozwiązaniu.

1. Utrzymujemy dwie ostatnie podtablice $dp[i-1][\star][\star]$ oraz $dp[i][\star][\star]$. Jako że pojedyncza liczba 32-bitowa zajmuje 4 bajty, to zużycie pamięci wynosi $2 \cdot \max_d \cdot \max_A \cdot 4\,\mathrm{B} = 2 \cdot 10 \cdot 2^{20} \cdot 4\,\mathrm{B} = 80\,\mathrm{MB}$. Pozwala to na zaliczenie podzadań 1–4 z podwyższonym limitem pamięci. Takie rozwiązanie zaimplementowano w plikach `nims2.cpp`, `nims4.cpp` oraz `nims14.cpp`.

2. Usuwamy pierwszy wymiar tablicy i liczymy $dp[\star][x], dp[\star][x \oplus a_i]$ niezależnie dla każdej pary $(x, x \oplus a_i)$ na podstawie poprzednich wartości tablicy. Zużycie pamięci jest oczywiście dwukrotnie niższe (40 MB). Pozwala to już na zdobycie pełnej punktacji. Przykładowe implementacje znajdują się w plikach `nim.cpp`, `nim1.pas` oraz `nim15.cpp`.

Inne rozwiązanie – przypadek $d \leqslant 2$

Istnieje dość prosty sposób na rozwiązanie trzeciego podzadania. Zauważmy najpierw, że przypadek $d = 2$ można prosto sprowadzić do przypadku $d = 1$, do każdej liczby dodając dodatkowy bit (2^{20}) zawsze równy 1. Wtedy `xor` podzbioru liczb jest równy 0 tylko wtedy, gdy w oryginalnych wysokościach stosów `xor` był równy 0 oraz liczba stosów była parzysta (ten warunek jest wymuszany przez dodatkowy bit). Tak samo jak poprzednio, odejmujemy 1 od wyniku tylko w przypadku, gdy d jest dzielnikiem n. Pozostało więc rozwiązać przypadek $d = 1$.

Udowodnijmy następujące fakty:

Fakt 2. *Niech S będzie zbiorem liczb, które można uzyskać za pomocą* `xor`*-a pewnej (być może zerowej) liczby wysokości stosów. Jeśli $x, y \in S$, to też $x \oplus y \in S$.*

Dowód: Niech $X = \{x_1, \ldots, x_p\}$ będzie podzbiorem stosów wykorzystanych do uzyskania `xor` równego x, zaś $Y = \{y_1, \ldots, y_q\}$ – `xor` równego y. Wtedy oczywiście $x_1 \oplus \ldots \oplus x_p \oplus y_1 \oplus \ldots \oplus y_q = x \oplus y$. Jeśli pewien stos a_i znajduje się zarówno w X, jak i w Y, to w powyższym równaniu wystąpi dwukrotnie i można go pominąć, gdyż $a_i \oplus a_i = 0$. Rozważmy zbiór stosów $Z = \{x : x \text{ jest doiładnie w jednym ze zbiorów } X, Y\}$. Wtedy `xor` wysokości stosów ze zbioru Z jest równy dokładnie $x \oplus y$. ■

Fakt 3. *Każdy z elementów S można uzyskać na tyle samo sposobów.*

Dowód: Prosta indukcja po liczbie stosów i. Na początku $i = 0$ i baza jest oczywista (S jest jednoelementowy, tj. $S = \{0\}$). Weźmy teraz zbiór S uzyskany dla stosów $a_1, \ldots, a_{i-1}$, zaś z niech będzie liczbą sposobów na uzyskanie każdego spośród elementów S. Dołóżmy stos a_i. W opisie dp mamy

$$dp[i][0][x] = dp[i-1][0][x] + dp[i-1][0][x \oplus a_i].$$

1. Jeśli $a_i \in S$, to nie może zajść jednocześnie $x \in S$, $x \oplus a_i \notin S$ (natychmiastowy wniosek z faktu 2) – podobnie nie zajdzie jednocześnie $x \notin S$, $x \oplus a_i \in S$. Tak więc albo oba składniki będą równe z, albo oba będą równe 0. Wobec tego $dp[i][0][\star] \in \{0, 2z\}$.

2. Jeśli $a_i \notin S$, to nie może być jednocześnie $x, x \oplus a_i \in S$ (wtedy $a_i = x \oplus (x \oplus a_i) \in S$). Stąd maksymalnie jeden składnik jest równy z. Tak więc $dp[i][0][\star] \in \{0, z\}$.

W obu przypadkach wszystkie niezerowe wartości dp są równe. ■

Z powyższych faktów i ich dowodów wynika bardzo prosty algorytm – będziemy utrzymywać zbiór S niezerowych pozycji w dp oraz wartość tych pozycji z modulo M. Początkowo $S = \{0\}$, $z = 1$. Dla każdego nowego stosu a_i:

1. jeśli $a_i \in S$, to z rośnie dwukrotnie;

2. jeśli $a_i \notin S$, to do S dodajemy wszystkie wysokości stosów postaci $x \oplus a_i$ dla $x \in S$ (widzimy ze wzoru na dp, że tylko one staną się niezerowe).

Wynikiem jest ostatecznie $z \bmod M$ (minus jeden w razie potrzeby). Rozwiązanie bazujące na podobnych pomysłach zostało zaimplementowane w pliku `nimb11.cpp`. Przechodzi ono wszystkie testy w trzecim podzadaniu.

Miłośnikom matematyki podpowiemy, że S posiada strukturę przestrzeni liniowej nad ciałem rzędu 2 ($1 \oplus 1 = 0$), rozpinanej przez zbiór wysokości stosów. Przedstawiony wyżej algorytm można interpretować jako metodę eliminacji Gaussa na odpowiednio skonstruowanej macierzy. Każdą liczbę a_i zapisujemy w postaci binarnej: $a_i = a_{i,0} \cdot 2^0 + a_{i,1} \cdot 2^1 + \ldots + a_{i,20} \cdot 2^{20}$. Konstruujemy macierz binarną A (z dodawaniem bitów takim, jak w operacji `xor`):

$$A = \begin{pmatrix} a_{1,0} & a_{1,1} & \ldots & a_{1,20} \\ a_{2,0} & a_{2,1} & \ldots & a_{2,20} \\ a_{3,0} & a_{3,1} & \ldots & a_{3,20} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,0} & a_{n,1} & \ldots & a_{n,20} \end{pmatrix}.$$

Okazuje się, że wynikiem jest $2^{n-\operatorname{rk} A}$, gdzie $\operatorname{rk} A$ jest tak zwanym *rzędem* macierzy A, który można łatwo wyznaczyć metodą eliminacji Gaussa.

Rozwiązania błędne

- Uczestnicy mogli zapomnieć o odjęciu 1 w programowaniu dynamicznym w przypadku $d \mid n$. Przykład takiego rozwiązania jest w pliku `nimb5.cpp`. Rozwiązanie nie otrzymuje żadnych punktów, ale błąd jest wyłapywany przez dostępne publicznie testy `ocen`.

- Program zapominający o tym, iż trzeba podać resztę z dzielenia wyniku przez $10^9 + 7$, znajduje się w pliku `nimb6.cpp`. Przechodzi pierwsze podzadanie.

- Bardzo trudnym do wychwycenia przypadkiem (w szczególności podczas testowania z wygenerowanymi losowo testami) była sytuacja, w której musimy odjąć 1 od wyniku podzielnego przez $10^9 + 7$. Reszta przed odjęciem jedynki wynosiła 0. Rozwiązanie wypisujące w tym przypadku -1 zamiast 10^9+6 można znaleźć w pliku `nimb7.cpp`. Nie przechodzi pojedynczego testu w podzadaniu 2.