Wojciech Rytter, Bartosz Walczak — Treść zadania, Opracowanie
Jakub Radoszewski — Program

Dostępna pamięć: 64 MB. *OI, Etap III, dzień pierwszy, 01.04.2009*

Słowa

Niech h będzie funkcją określoną na napisach złożonych z cyfr `0` *i* `1`*. Funkcja h przekształca napis w, zastępując (niezależnie i równocześnie) każdą cyfrę* `0` *przez* `1` *i każdą cyfrę* `1` *przez napis „*`10`*”. Na przykład $h($„`1001`”$)=$ „`101110`”, $h($„”$)=$ „” (tzn. funkcja h zastosowana do pustego napisu jest pustym napisem). Zauważmy, że h jest różnowartościowa. Przez h^k oznaczmy k-krotne złożenie funkcji h ze sobą. W szczególności, h^0 to funkcja identycznościowa $h^0(w)=w$.*

Interesują nas napisy postaci $h^k($„`0`”$)$ dla $k=0,1,2,3,\ldots$ Oto kilka pierwszych takich napisów:

„`0`”, „`1`”, „`10`”, „`101`”, „`10110`”, „`10110101`”.

Mówimy, że napis x jest **podsłowem** *napisu y, jeżeli występuje w nim jako spójny (tj. jednokawałkowy) podciąg. Mamy dany ciąg liczb naturalnych $k_1,k_2,\ldots,k_n$. Celem zadania jest sprawdzenie, czy napis postaci*

$$h^{k_1}(\text{„0”})\cdot h^{k_2}(\text{„0”})\cdot\ldots\cdot h^{k_n}(\text{„0”})$$

jest podsłowem $h^m($„`0`”$)$ dla pewnego m. Przy tym operacja „$\cdot$” oznacza sklejenie (konkatenację) napisów.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą t, $1\leqslant t\leqslant 13$, oznaczającą liczbę przypadków testowych do rozważenia. Pierwszy wiersz opisu każdego przypadku zawiera jedną liczbę całkowitą n, $1\leqslant n\leqslant 100\,000$. W drugim wierszu opisu znajduje się n nieujemnych liczb całkowitych $k_1,k_2,\ldots,k_n$ pooddzielanych pojedynczymi odstępami. Suma liczb z drugiego wiersza każdego przypadku jest nie większa niż 10 000 000.

Wyjście

Twój program powinien wypisać na standardowe wyjście t wierszy, po jednym dla każdego przypadku testowego. Wiersz odpowiadający danemu przypadkowi testowemu powinien zawierać jedno słowo: `TAK` *— jeśli w tym przypadku $h^{k_1}($„`0`”$)\cdot h^{k_2}($„`0`”$)\cdot\ldots\cdot h^{k_n}($„`0`”$)$ jest podsłowem $h^m($„`0`”$)$ dla pewnego m, lub* `NIE` *w przeciwnym razie.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`2`	`TAK`
`2`	`NIE`
`1 2`	
`2`	
`2 0`	

Wyjaśnienie do przykładu: *Słowo z pierwszego przypadku testowego to „`110`" — jest ono podsłowem na przykład słowa $h^4($„`0`"$)$ = „`10110`". W drugim przypadku testowym występuje słowo „`100`", które nie jest podsłowem żadnego słowa postaci $h^m($„`0`"$)$.*

Rozwiązanie

Dla wygody wprowadźmy oznaczenie $h_k := h^k(0)$. Słowa h_k są to tzw. słowa Fibonacciego. Zazwyczaj definiuje się je rekurencyjnie w następujący sposób:

$$h_0 = 0, \quad h_1 = 1, \quad h_k = h_{k-1} \cdot h_{k-2} \quad \text{dla } k \geqslant 2. \tag{$*$}$$

Równoważność definicji $(*)$ i tej podanej w treści zadania łatwo pokazać przez indukcję. Rozwiązanie wzorcowe bazuje na definicji przez podstawienie, którą podano w treści zadania. Postać $(*)$ może być jednak pomocna w zrozumieniu niektórych własności, z których będziemy korzystać.

Powiemy, że ciąg $(k_1, \ldots, k_n)$ jest *dobry*, jeżeli $h_{k_1} \ldots h_{k_n}$ jest podsłowem h_m dla pewnego m, oraz że jest *zły* w przeciwnym przypadku. Zadanie polega więc na sprawdzeniu, czy dany na wejściu ciąg $(k_1, \ldots, k_n)$ jest dobry.

Pierwsze podejście do rozwiązania tego zadania może wyglądać następująco: generujemy całe słowo $u = h_{k_1} \ldots h_{k_n}$, po czym sprawdzamy za pomocą algorytmu wyszukiwania wzorca (np. KMP), czy występuje ono jako podsłowo w h_m dla odpowiednio dobranego m. Okazuje się, że jeżeli h_m jest co najmniej 8 razy dłuższe niż wyszukiwane słowo, to rozwiązanie takie działa poprawnie. Uzasadnienie tego spostrzeżenia polega na tym, że jeżeli u jest nie dłuższe niż h_{m-3} dla pewnego m, to jeśli jest ono podsłowem h_{m+1}, to jest też podsłowem h_m — równoważnie, jeżeli nie uda nam się znaleźć u w h_m, to nie ma potrzeby szukania tego słowa w h_{m+1} i dalszych. Faktycznie, jeżeli przy podziale $h_{m+1} = h_m \cdot h_{m-1}$ słowo u leży w ramach h_m lub h_{m-1}, to jest to jasne, a w przeciwnym przypadku wystarczy skorzystać z zapisu $h_{m+1} = h_{m-1}h_{m-2} \cdot h_{m-3}h_{m-4}h_{m-3}$, jako że $h_{m-2} \cdot h_{m-3} = h_{m-1}$.

Już pobieżna analiza pozwala stwierdzić, że przy ograniczeniach na $k_1, \ldots, k_n$ podanych w treści zadania takie rozwiązanie nie ma szans powodzenia na dużych danych wejściowych, gdyż rozmiar słowa h_k rośnie wykładniczo względem k. Dlatego rozwiązanie, którego szukamy, musi operować tylko na ciągu $(k_1, \ldots, k_n)$, bez generowania słowa, które ten ciąg reprezentuje.

Rozwiązanie wzorcowe

Idea rozwiązania wzorcowego polega na przekształcaniu danego ciągu $w = (k_1, \ldots, k_n)$ przez coś w rodzaju funkcji odwrotnej do h. Okazuje się, że dla dużych elementów ciągu w cofanie

funkcji h polega na zwykłym ich zmniejszaniu, a dla małych (nie większych niż 3) trzeba czasem rozpatrywać pewne przypadki szczególne. Operacje wykonywane w algorytmie mają tę własność, że ciąg dobry przekształcają w ciąg dobry, a zły — w zły. Proces ten doprowadza w końcu do ciągu jednoelementowego, który oczywiście jest dobry (wtedy ciąg początkowy też był dobry), lub do ciągu, o którym potrafimy stwierdzić, że na pewno jest zły (wtedy początkowy był zły).

```
1:  w := (k_1, ..., k_n)
2:  while |w| > 1 do begin
3:    if w zawiera fragment k,0 dla k ∉ {1,3} then return false
4:    Wykonaj kolejno następujące operacje na ciągu w:
5:      zamień, jeżeli występuje, pierwszy element 0 → 2
6:      zamień, jeżeli występuje, ostatni element 3 → 2
7:      usuń, jeżeli występuje, ostatni element równy 1
8:      zamień wszystkie fragmenty 1,0 → 2
9:      zamień wszystkie fragmenty 3,0 → 2,2
10:     zmniejsz wszystkie elementy o 1
11: end
12: return true
```

Pełny kod rozwiązania znajduje się w plikach `slo.cpp` oraz `slo1.pas`.

Przykład działania algorytmu

Zobaczmy, jak zadziała rozwiązanie wzorcowe dla ciągu $w = (1,2,4,1,2,5)$. Poniższa tabela przedstawia, co dzieje się z ciągiem w w kolejnych iteracjach pętli **while** pod wpływem instrukcji w liniach 5–10 (numer instrukcji nad strzałką).

nr	przebieg iteracji
1	$(1,2,4,1,2,5) \xrightarrow{10} (0,1,3,0,1,4)$
2	$(0,1,3,0,1,4) \xrightarrow{5} (2,1,3,0,1,4) \xrightarrow{9} (2,1,2,2,1,4) \xrightarrow{10} (1,0,1,1,0,3)$
3	$(1,0,1,1,0,3) \xrightarrow{6} (1,0,1,1,0,2) \xrightarrow{8} (2,1,2,2) \xrightarrow{10} (1,0,1,1)$
4	$(1,0,1,1) \xrightarrow{7} (1,0,1) \xrightarrow{8} (2,1) \xrightarrow{10} (1,0)$
5	$(1,0) \xrightarrow{8} (2) \xrightarrow{10} (1)$

Iteracja 6 już się nie wykona, gdyż otrzymany w wyniku iteracji 5 ciąg w jest jednoelementowy. Jako wynik algorytmu otrzymujemy zatem, że ciąg $(1,2,4,1,2,5)$ jest dobry.

Złożoność rozwiązania wzorcowego

Zanim przystąpimy do dowodu poprawności przedstawionego rozwiązania, zastanówmy się nad czasem jego działania. W szczególności upewnijmy się, że algorytm ten w ogóle się zatrzymuje.

Pokażemy, że czas działania algorytmu na ciągu $(k_1, \ldots, k_n)$ wynosi $O(k_1 + \ldots + k_n + n)$. Przyjrzyjmy się pojedynczej iteracji pętli **while**. Niech n oznacza bieżącą długość ciągu w, a s — bieżącą sumę elementów ciągu w z pominięciem pierwszego, jeżeli jest mniejszy od 2. Pokażemy, że po wykonaniu instrukcji 5–10 wartość $s+n$ maleje o co najmniej $\lfloor \frac{n}{2} \rfloor$. Faktycznie, wykonanie instrukcji 5 (w połączeniu z instrukcją 10) nie zmienia wartości $s+n$, gdyż wówczas początkowe 0 lub 1 w ciągu nie jest uwzględnione w s. Inne zmiany, czyli:

- końcowe 3 przechodzi w 1 (dwukrokowo, wskutek instrukcji 6 i 10),
- końcowe 1 znika,
- fragment $1,0$ przechodzi w 1 (dwukrokowo),
- fragment $3,0$ przechodzi w $1,1$ (dwukrokowo),
- pozostałe elementy zmniejszają się o 1,

powodują zmniejszenie wartości $s+n$ o co najmniej 1 dla każdego elementu lub każdej pary kolejnych elementów ciągu w. Wynika stąd, że $s+n$ maleje łącznie o co najmniej $\lfloor \frac{n}{2} \rfloor$ (zmiana ta równałaby się $\lceil \frac{n}{2} \rceil$, gdyby nie specjalne traktowanie początkowego elementu w). Oczywiście jeżeli $s+n \leqslant 1$, algorytm się zatrzymuje. Jako że czas wykonania pojedynczej iteracji pętli **while** wynosi $O(n)$, jest on w pełni amortyzowany przez zmianę $s+n$. Zatem całkowity czas działania algorytmu wynosi $O(s+n)$.

Kilka prostych własności słów h_k

W przekształceniu $h(h_{k-1}) = h_k$ każda cyfra `0` w h_k powstaje w wyniku podstawienia `1` $\to$ `10`, a każda cyfra `1`, po której nie występuje `0` — w wyniku podstawienia `0` $\to$ `1`. Wynika stąd w szczególności, że:

(a) h_k zaczyna się od `1` dla $k \geqslant 1$,

(b) h_k kończy się na `0` dla k parzystych, a na `1` dla k nieparzystych,

(c) w h_k nie występuje podsłowo `00`,

(d) w h_k nie występuje podsłowo `111` (jest to wniosek z (c) dla słowa h_{k-1}),

(e) w h_k nie występuje podsłowo `101010` (jest to wniosek z (d) dla słowa h_{k-1}).

Poprawność rozwiązania wzorcowego

Zaczniemy od uzasadnienia, że kryterium w linii 3, które odrzuca ciąg w jako zły, jest poprawne.

Lemat 1. Jeżeli $k \notin \{1,3\}$, to h_k`0` nie jest podsłowem żadnego h_m.

Dowód: Jeżeli k jest parzyste, h_k kończy się cyfrą `0`. Wtedy h_k`0` ma sufiks `00`, który nie może być podsłowem żadnego h_m (własność (c)). Jeżeli k jest nieparzyste i $k \geqslant 5$, to można udowodnić (np. korzystając z postaci $(*)$), że słowo h_k ma sufiks $h_5 =$ `10110101`, a więc także `10101`, czyli `101010` jest sufiksem słowa h_k`0`. Ale zgodnie z własnością (e) powyżej, słowo `101010` nie może wystąpić w żadnym h_m. ■

Teraz uzasadnimy, że każda z operacji wykonywanych w liniach 5–10 jest poprawna, tzn. że ciąg w jest dobry po wykonaniu tej operacji wtedy i tylko wtedy, gdy był dobry przed jej wykonaniem. Zauważmy przy tym, że operacje w liniach 5–9 są od siebie niezależne i mogą zostać wykonane w dowolnej kolejności. Dla ciągu $w = (k_1, \ldots, k_n)$ wprowadzamy oznaczenie $h_w = h_{k_1} \ldots h_{k_n}$.

Jeżeli w ciągu w, poza pierwszym elementem, występuje jakiekolwiek 0, to musi być ono poprzedzone przez 1 albo 3, gdyż w przeciwnym przypadku ciąg w zostałby odrzucony w linii 3. Ponieważ $h_1h_0 =$ `10` $= h_2$ i $h_3h_0 =$ `1010` $= h_2h_2$, instrukcje 8 i 9 przekształcają ciąg w w równoważny ciąg w', taki że $h_w = h_{w'}$. To dowodzi, że operacje te są poprawne, ponadto eliminują one wszystkie zera występujące w ramach w poza być może pierwszym.

Aby pozbyć się początkowego zera, zauważmy, że jeżeli h_w występuje jako podsłowo w h_m, to początkowe zero musi być drugą cyfrą fragmentu `10` $= h_2$. Oznacza to, że jeżeli na początku ciągu w zamienimy 0 na 2, to dla tak otrzymanego w' słowo $h_{w'}$ również występuje jako podsłowo w h_m. To dowodzi poprawności operacji w linii 5.

Uzasadnimy teraz poprawność instrukcji 6, 7 i 10. Załóżmy, że w ciągu w nie występuje żadne zero. Oznaczmy przez w' ciąg, który powstaje z w po zastosowaniu operacji w liniach 6 i 7. Można zauważyć, że każda z tych operacji odcina ostatnią cyfrę `1` z h_w, jeżeli ciąg w kończy się na 1 lub 3. Zatem $h_w = h_{w'}$ lub $h_w = h_{w'}$`1`, przy czym w tym pierwszym przypadku ciąg w' nie kończy się na 1 ani na 3. Oznaczmy przez w'' ciąg, który powstaje z w' przez zmniejszenie wszystkich elementów o 1 (instrukcja w linii 10).

Udowodnimy najpierw, że jeżeli w jest dobry, to w'' jest dobry. Załóżmy, że h_w jest podsłowem h_m, i zastanówmy się, z czego powstaje to podsłowo w przekształceniu $h(h_{m-1}) = h_m$. Niech $w' = (k_1, \ldots, k_n)$. Pokażemy, że dla każdego i odpowiedni człon h_{k_i} w $h_{w'} = h_{k_1} \ldots h_{k_n}$ powstaje dokładnie z h_{k_i-1}. Funkcja h jest różnowartościowa, a zatem nie istnieje żadne inne słowo x, takie że $h(x) = h_{k_i}$. Jeśli więc h_{k_i} nie powstaje z h_{k_i-1}, to przekształcenie $h(h_{m-1}) = h_m$ zamienia pewną cyfrę `1` w blok `10`, który występuje na granicy wystąpienia h_{k_i} w h_m, tzn. prawe `0` jest pierwszą cyfrą h_{k_i} lub lewe `1` jest ostatnią cyfrą h_{k_i}. Ten pierwszy przypadek jest niemożliwy, bo h_{k_i} zaczyna się od `1` (jako że $k_i \geqslant 1$). Drugi przypadek może zajść jedynie wtedy, gdy po h_{k_i} występuje `0`. Tak nie jest dla $i < n$ (wówczas po h_{k_i} występuje $h_{k_{i+1}}$) ani dla $i = n$ w przypadku, gdy $h_w = h_{w'}$`1`. Zatem musi być $i = n$ oraz $h_w = h_{w'}$, ale wtedy $k_i \notin \{1,3\}$, więc otrzymujemy sprzeczność z Lematem 1. Tym samym pokazaliśmy, że fragment $h_{k_1} \ldots h_{k_n}$ w h_w jako podsłowie h_m powstaje w wyniku zastosowania przekształcenia h do $h_{k_1-1} \ldots h_{k_n-1} = h_{w''}$. Zatem $h_{w''}$ jest podsłowem h_{m-1}, czyli w'' jest dobry.

Pozostała do wykazania implikacja w drugą stronę. Jeżeli w'' jest dobry, to istnieje takie m, że $h_{w''}$`0` lub $h_{w''}$`1` jest podsłowem h_m (wynika to z $(*)$). Stąd $h(h_{w''}$`0`$) = h_{w'}$`1` lub $h(h_{w''}$`1`$) = h_{w'}$`10` jest podsłowem h_{m+1}, a jedno i drugie zaczyna się od h_w. Zatem w jest dobry. To kończy dowód poprawności rozwiązania wzorcowego.

Rozwiązanie alternatywne

Inne rozwiązanie tego zadania bazuje na następującej własności: słowo x jest prefiksem jakiegoś h_m ($m \geqslant 1$) wtedy i tylko wtedy, gdy x można złożyć ze słów Fibonacciego w następujący sposób (uwaga na odwróconą kolejność indeksów):

$$x = h_{\ell_r} \ldots h_{\ell_1}, \quad \text{gdzie} \quad \ell_1 \in \{1,2\}, \quad \ell_i \in \{\ell_{i-1}+1, \ell_{i-1}+2\} \quad \text{dla } i = 2,\ldots,r. \qquad (**)$$

Na przykład słowo `1011010110110101`, które jest prefiksem h_7, ma rozkład $h_5h_4h_2h_1$. Dowód indukcyjny tego faktu korzysta z rekurencyjnej definicji słów Fibonacciego $(*)$.

Każde podsłowo x słowa h_m rozszerza się z lewej do prefiksu h_m. Powyższa własność pozwala łatwo uzyskać minimalne takie rozszerzenie. Wybieramy $\ell_1 = 1$ lub $\ell_1 = 2$ na podstawie ostatniej cyfry słowa x i obcinamy x z prawej o h_{ℓ_1}. To samo robimy kolejno dla $i = 2,\ldots,r$: wybieramy $\ell_i = \ell_{i-1}+1$ lub $\ell_i = \ell_{i-1}+2$ na podstawie ostatniej cyfry i obcinamy x z prawej o h_{ℓ_i}. Ponieważ dwa kolejne słowa Fibonacciego różnią się ostatnią cyfrą, wybór w każdym kroku jest jednoznaczny. Jeżeli w pewnym kroku końcówka słowa x nie zgadza się z wybranym h_{ℓ_i}, to słowo, od którego zaczęliśmy, nie może być podsłowem żadnego h_m. W przeciwnym razie otrzymujemy prefiks h_m, którego sufiksem (a więc podsłowem h_m) jest początkowe słowo x.

Oczywiście, aby to rozwiązanie mogło zadziałać efektywnie, należy konstruować rozkład $(**)$ słowa $h_{k_1} \ldots h_{k_n}$, posługując się jedynie indeksami $k_1,\ldots,k_n$ oraz $\ell_1,\ldots,\ell_r$. Nie jest to jednak trudne, o czym Czytelnik może się przekonać, zaglądając do pliku `sloa.cpp`.

Testy

Zadanie zostało ocenione na zestawie 10 testów, z których każdy zawierał 13 przypadków testowych.

Nazwa	n	s	k z przedziału	Opis
slo1.in	[1, 1457]	[0, 899]	[0, 26]	test ręcznie generowany
slo2.in	[4,20000]	[27,61803]	[0, 28]	test ręcznie generowany
slo3.in	2000	100000	[100, 400]	test losowy
slo4.in	6000	200000	[150, 700]	test losowy
slo5.in	15000	1000000	[200,2000]	test losowy
slo6.in	50000	10000000	[300,2000]	test losowy
slo7.in	70000	6000000	[300, 800]	test losowy
slo8.in	80000	10000000	[300,1200]	test losowy
slo9.in	100000	10000000	[600,1000]	test losowy
slo10.in	100000	10000000	[300, 600]	test losowy

Testy 1 i 2 zostały ułożone w całości ze specyficznych przypadków testowych, między innymi takich, które odrzucają wybrane rozwiązania błędne. Tylko te testy zalicza rozwiązanie nieoptymalne oparte na wyszukiwaniu wzorca (jego implementacje można znaleźć w plikach `slos1.cpp` i `slos3.pas`).

Każdy z testów 3–10 składa się z trzech przypadków specyficznych oraz 10 głównych, wśród których są przypadki z odpowiedzią pozytywną i negatywną. Testy główne były generowane następująco: generowano losowo ciąg o długości kilkakrotnie większej niż n reprezentujący słowo h_k, po czym wybierano z niego losowy fragment o długości bliskiej n i sumie bliskiej s. W przypadku żądania odpowiedzi pozytywnej po prostu wypisywano wybrany fragment, a w przypadku odpowiedzi negatywnej zmieniano wartość losowego elementu ciągu o ± 2, co prawie gwarantuje przejście do odpowiedzi negatywnej.