Marek Sokołowski
Treść zadania, Opracowanie

Tomasz Syposz
Program

Dostępna pamięć: 8 MB. *OI, etap III, dzień drugi, 17.04.2015*

Kolekcjoner Bajtemonów

Bajtazar bardzo lubi kolekcjonować karty z Bajtemonami. Na każdej karcie w jego talii narysowany jest jeden Bajtemon wraz z numerem katalogowym, będącym liczbą całkowitą z przedziału $[1, 2 \cdot 10^9]$. Bajtazar nie zgromadził jeszcze wszystkich Bajtemonów. Jego kolekcja jest dość osobliwa: każdy Bajtemon, który się w niej znajduje, występuje na kilku kartach, co więcej, każdy na takiej samej liczbie kart.

Pewnego dnia Bajtazar zorientował się, że ktoś podkradł mu kilka (jedną lub więcej) kart z jego kolekcji. Wie, że na brakujących kartach był ten sam Bajtemon i że na szczęście został mu przynajmniej jeden egzemplarz tej karty. Niestety, nasz bohater ma bardzo mały rozumek i zdążył już zapomnieć, jaki to był Bajtemon. Czy jesteś w stanie mu pomóc i przypomnieć, jakich kart mu brakuje? Twój program też musi pamiętać o ograniczeniach pamięci...

Napisz program komunikujący się z biblioteką służącą do przeglądania talii kart Bajtazara, który znajdzie numer katalogowy podkradzionego Bajtemona.

Komunikacja

Aby użyć biblioteki, należy wpisać na początku programu:

- **C/C++***:* `#include "ckollib.h"`
- **Pascal***:* `uses pkollib;`

Biblioteka udostępnia następujące funkcje i procedury:

- `karta` – *Daje w wyniku liczbę całkowitą z przedziału $[1, 2 \cdot 10^9]$ bądź 0. Wartość 0 oznacza koniec talii kart, zaś dodatnia liczba całkowita – numer katalogowy Bajtemona zapisany na kolejnej karcie. Po otrzymaniu wyniku funkcji 0, Twój program może wciąż wywoływać funkcję* `karta` *– odpowiada to kolejnemu przeglądaniu talii kart przez Bajtazara. Przy każdym przeglądaniu talii, numery katalogowe Bajtemonów podawane są w tej samej kolejności.*
  - **C/C++:** `int karta();`
  - **Pascal:** `function karta: LongInt;`
- `odpowiedz(wynik)` – *Odpowiada bibliotece, że* `wynik` *to numer katalogowy Bajtemona, który znajduje się na podkradzionych kartach. Wywołanie tej funkcji* **kończy działanie Twojego programu**.
  - **C/C++:** `void odpowiedz(int wynik);`
  - **Pascal:** `procedure odpowiedz(wynik: LongInt);`

Twój program **nie może** *czytać żadnych danych (ani ze standardowego wejścia, ani z plików).* **Nie może** *również nic wypisywać do plików ani na standardowe wyjście. Może pisać na standardowe wyjście diagnostyczne (*`stderr`*) – pamiętaj jednak, że zużywa to cenny czas.*

Ocenianie

Twój program otrzyma za dany test pełną punktację, jeśli przejrzy całą talię nie więcej niż raz (tj. nie wywoła funkcji `karta` *po tym, gdy pierwszy raz da ona w wyniku wartość 0).*

Jeśli program rozpocznie przeglądanie talii po raz drugi, otrzyma tylko 40% punktów za dany test.

W przypadku, gdy program zacznie przeglądać talię po raz trzeci, biblioteka przerwie działanie programu i nie uzyskasz za ten test żadnych punktów.

Liczba kart w talii nie przekroczy 60 000 000. Ponadto, w testach o łącznej wartości 20% punktów liczba kart w talii nie przekroczy 200 000.

Przykładowy przebieg programu

C/C++	Pascal	Wynik	Wyjaśnienie
`k = karta();`	`k := karta;`	13	Numer katalogowy Bajtemona z pierwszej karty.
`k = karta();`	`k := karta;`	13	
`k = karta();`	`k := karta;`	39	
`k = karta();`	`k := karta;`	26	
`k = karta();`	`k := karta;`	26	
`k = karta();`	`k := karta;`	0	Koniec talii.
`k = karta();`	`k := karta;`	13	Nowy obieg talii; karty są w tej samej kolejności, co poprzednio.
`odpowiedz(39);`	`odpowiedz(39);`		Udzielamy odpowiedzi. Program kończy swoje działanie.

Powyższy przebieg programu jest poprawny (dokonuje nie więcej niż dwóch przeglądnięć talii i daje poprawny wynik). Jednakże rozpoczyna on drugie przeglądanie talii, więc program otrzyma w tym przypadku 40% punktów. Aby uzyskać pełną punktację, należy dokonać tylko jednego przeglądnięcia talii.

Eksperymenty

W katalogu `dlazaw` *dostępna jest przykładowa biblioteka, która pozwoli Ci przetestować poprawność formalną rozwiązania. Biblioteka wczytuje opis talii kart ze standardowego wejścia w następującym formacie:*

- *w pierwszym wierszu dodatnia liczba całkowita n – liczba kart w talii,*
- *w drugim wierszu n liczb całkowitych z przedziału $[1, 2 \cdot 10^9]$ – numery Bajtemonów na kolejnych kartach talii.*

Przykładowa biblioteka **nie sprawdza**, *czy faktycznie w talii istnieje tylko jeden Bajtemon narysowany na mniejszej liczbie kart niż wszystkie pozostałe Bajtemony występujące w talii.*

Przykładowe wejście dla biblioteki znajduje się w pliku `kol0.in`. *Po wywołaniu procedury* `odpowiedz`, *biblioteka wypisuje na standardowe wyjście informację o udzielonej odpowiedzi, liczbie wywołań funkcji* `karta` *i liczbie rozpoczętych przebiegów.*

W tym samym katalogu znajdują się przykładowe rozwiązania `kol.c`*,* `kol.cpp` *i* `kol.pas` *korzystające z biblioteki. Rozwiązania te nie są poprawne i zawsze odpowiadają, że szukanym numerem Bajtemona jest numer umieszczony na ostatniej karcie w talii.*

Do kompilacji rozwiązania wraz z biblioteką służą polecenia:

- **C***:* `gcc -O2 -static ckollib.c kol.c -lm`
- **C++***:* `g++ -O2 -static ckollib.c kol.cpp -lm -std=gnu++0x`
- **Pascal***:* `ppc386 -O2 -XS -Xt kol.pas`

Plik z rozwiązaniem i biblioteka powinny znajdować się w tym samym katalogu.

Rozwiązanie

W zadaniu mamy dany ciąg liczb. Wiadomo o nim, że każdy element występuje w nim dokładnie tyle samo razy – poza jedną z liczb x, której liczba wystąpień jest mniejsza. Naszym zadaniem jest znaleźć element x.

Inaczej niż w klasycznych zadaniach olimpijskich, tutaj nie możemy wczytywać danych ze standardowego wejścia. Jedynym sposobem na poznanie liczb jest odpytywanie funkcji z biblioteki dostarczonej w trakcie zawodów przez organizatorów. Na zawodników czekał jeszcze szereg kolejnych utrudnień – początkowo ani nie znamy długości ciągu (jego długość poznajemy dopiero, gdy dojdziemy do końca ciągu), ani nie wiemy, ile razy każda liczba w tym ciągu występuje. Dodatkowym problemem jest też niewielki rozmiar pamięci dostępnej dla programu. Tak więc szukamy rozwiązania działającego w pamięci stałej lub niemal stałej. Ostatecznie wolno nam przejrzeć cały ciąg tylko raz (lub dwa razy, jeśli celujemy w rozwiązanie uzyskujące częściową punktację), w kolejności określonej przez bibliotekę.

W dalszym rozumowaniu oznaczamy przez n długość ciągu, przez A największy element w ciągu, natomiast przez k liczbę różnych elementów występujących w ciągu.

Rozwiązania pamięciochłonne

Najbardziej brutalnym rozwiązaniem problemu może być siłowe wyznaczenie liczby wystąpień każdego elementu w ciągu. Jedną z metod osiągnięcia tego celu może być zapisanie całego ciągu do tablicy. Następnie elementy tablicy możemy łatwo posortować, a potem w czasie liniowym określić, który element występuje najrzadziej. Rozwiązanie to ma złożoność obliczeniową $\Theta(n \log n)$, jednakże złożoność pamięciowa wynosi $\Theta(n)$ i jest zdecydowanie niedopuszczalna przy limicie pamięci równym 8 MB. Implementacja tej metody znajduje się w plikach `kols1.cpp`, `kols3.pas`, `kols4.c` i dostawała na zawodach 20% punktów.

Można nieco usprawnić poprzednie rozwiązanie i zauważyć, że niepotrzebnie tracimy pamięć na przechowywanie elementów o tej samej wartości. Korzystając ze struktury umiejącej grupować wystąpienia identycznych elementów (np. struktury `map` z biblioteki STL w C++), możemy dla każdego dotychczas napotkanego elementu utrzymywać liczbę jego wystąpień. Otrzymujemy w ten sposób rozwiązanie działające w czasie $\Theta(n \log k)$ i pamięci $\Theta(k)$. Implementacja znajduje się w pliku `kols2.cpp` i na zawodach otrzymywała 30% punktów.

Gdyby każdy element występował maksymalnie dwukrotnie

Poniżej przedstawiamy dość znany problem. Czytelnikowi polecamy zastanowić się nad jego rozwiązaniem przed przeczytaniem rozwiązania.

Mamy bardzo długi ciąg liczb, o którym wiadomo, że każda liczba występuje w nim dwukrotnie, oprócz jednego elementu, który nie ma swojego „wspólnika". Zadanie polega na znalezieniu „samotnika" w czasie liniowym i pamięci stałej (zakładamy, że operacje na liczbach są wykonywane w czasie stałym).

Istnieje bardzo proste rozwiązanie tej zagadki. Okazuje się bowiem, że poszukiwana liczba jest dokładnie alternatywą wykluczającą (tj. `xor`) wszystkich liczb w ciągu.

Operację tę najłatwiej zrozumieć, wypisując w rzędach rozwinięcia binarne rozważanych przez nas liczb. Wynik operacji ma ustawiony i-ty bit, gdy nieparzyście wiele `xor`-owanych liczb ma ten bit ustawiony. Poniższy przykład wyznacza `xor` liczb 14, 11, 14, 5 i 11:

$$\begin{array}{rcccc} 14_{(10)} = & 1 & 1 & 1 & 0 \\ 11_{(10)} = & 1 & 0 & 1 & 1 \\ 14_{(10)} = & 1 & 1 & 1 & 0 \\ 5_{(10)} = & 0 & 1 & 0 & 1 \\ 11_{(10)} = & 1 & 0 & 1 & 1 \\ \hline & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} \end{array} = \mathbf{5_{(10)}}.$$

Zauważmy, że jeśli jakaś liczba występuje w ciągu dwukrotnie, to nie wpływa na parzystość bitów w wyniku. Wobec tego na wynik wpływa tylko pojedyncza samotna liczba i to ona jest wynikiem.

Rozwiązanie naiwnie zakładające, że każdy element występuje w ciągu co najwyżej dwukrotnie, znajduje się w pliku `kolb9.cpp`. Nie dostaje żadnych punktów.

W kierunku dobrego rozwiązania

Na szczęście jednak powyższe przemyślenia można wykorzystać w poprawnym algorytmie, choć wymagającym dwóch przejrzeń ciągu.

Niech p oznacza liczbę wystąpień każdego elementu, którego żadna kopia nie została usunięta z ciągu. Liczbę p możemy wyznaczyć w pierwszym przebiegu po ciągu. Bierzemy bowiem dwa najwcześniejsze różne elementy w ciągu i liczymy, ile razy występują one w ciągu. Oznaczmy te liczby wystąpień przez p_1, p_2. Wtedy:

- Jeśli nie znaleźliśmy dwóch różnych elementów w ciągu, oznacza to, że szukaną liczbą jest którykolwiek element ciągu. Jest to przypadek szczególny – pominięcie go narażało zawodników na utratę części punktów.
- Jeśli $p_1 = p_2$, to $p = p_1 = p_2$; wciąż nie wiemy jednak, który element występuje mniej niż p razy.

- Jeśli $p_1 < p_2$, to pierwszy napotkany element w ciągu występuje mniej razy niż inne elementy; to on jest wynikiem. Analogicznie rozpatrujemy przypadek $p_1 > p_2$.

Skoro już wiemy, ile razy powinien wystąpić każdy element w ciągu, możemy wykonać operację podobną do powyższej. Zamiast jednak wyznaczać parzystość liczby wystąpień każdego bitu, możemy wyznaczyć resztę z dzielenia przez p tej liczby. Dla przykładu, dla liczb $14, 11, 11, 5, 14, 14, 5, 11$ dowiadujemy się, że $p = 3$. Wtedy:

$$\begin{array}{rcccc} 14_{(10)} = & 1 & 1 & 1 & 0 \\ 11_{(10)} = & 1 & 0 & 1 & 1 \\ 11_{(10)} = & 1 & 0 & 1 & 1 \\ 5_{(10)} = & 0 & 1 & 0 & 1 \\ 14_{(10)} = & 1 & 1 & 1 & 0 \\ 14_{(10)} = & 1 & 1 & 1 & 0 \\ 5_{(10)} = & 0 & 1 & 0 & 1 \\ 11_{(10)} = & 1 & 0 & 1 & 1 \\ \hline & \mathbf{0} & \mathbf{2} & \mathbf{0} & \mathbf{2} \end{array} \quad \rightarrow \mathbf{5_{(10)}}.$$

Podobnie jak poprzednio, liczby występujące p-krotnie w ciągu nie wpływają na wynik. Wobec tego wynik wyznacza liczba występująca $m < p$ razy. Po wyznaczeniu reszty z dzielenia przez p liczby wystąpień kolejnych bitów każdy bit wynikowej liczby wystąpi w wyniku dokładnie m razy.

Podane rozwiązanie wykonuje dwa przebiegi po ciągu i udziela poprawnej odpowiedzi w czasie $\Theta(n \log A)$ i w pamięci $\Theta(\log A)$. Implementacja znajduje się w pliku `kol3.cpp` i otrzymuje 40% punktów.

Rozwiązanie wzorcowe

Powyższą metodę można nieznacznie zmodyfikować w taki sposób, by wykorzystać tylko jeden przebieg po ciągu.

Zauważmy bowiem, iż podczas rozważania kolejnych elementów nie musimy dla każdego bitu wyniku od razu liczyć reszty z dzielenia przez p liczby elementów ciągu z ustawionym tym bitem. Możemy tę resztę wyznaczyć dopiero po przejrzeniu całego ciągu. Wtedy równolegle możemy też otrzymać wartość p. Wynik obliczamy w taki sam sposób, jak poprzednio.

Rozwiązanie zostało zaimplementowane w pliku `kol2.cpp`. Działa w czasie $\Theta(n \log A)$ i pamięci $\Theta(\log A)$, dokonuje jednego przebiegu ciągu i otrzymuje maksymalną punktację.

Jeszcze szybciej

Za pomocą jednego prostego zabiegu możemy przyspieszyć program. Do tej pory zliczaliśmy bowiem bity, czyli „przytwierdziliśmy się" do systemu pozycyjnego o podstawie 2. Nic jednak nie stoi na przeszkodzie, by tę podstawę zwiększyć do pewnego d. Dla ułatwienia rozważań obierzmy $d = 2^{16}$. Wtedy dla każdej pozycji cyfry w liczbie

zliczamy, ile razy każda cyfra wystąpiła na tej pozycji. Dla każdej wynikowej cyfry jej liczba wystąpień będzie liczbą niepodzielną przez p.

Ponieważ $d^2 > 2 \cdot 10^9$, to potrzebujemy w obecnej chwili tylko dwóch tablic. Mają one co prawda większy rozmiar (d), jednak każda kolejna napotkana liczba wymaga jedynie dwóch operacji na tablicach. Odzyskanie wyniku wymaga jedynie przejrzenia obu tablic.

Zauważmy, że nie możemy ustalić tutaj $d = 2 \cdot 10^9$, gdyż tablica rozmiaru $\Theta(d)$ nie miała prawa zmieścić się w tak małej pamięci.

Rozwiązanie jest zaimplementowane w plikach `kol.cpp`, `kol4.cpp`, `kol5.pas`. Oczywiście również dostaje maksymalną liczbę punktów.

Rozwiązanie alternatywne

Można było też nieznacznie zmienić typ informacji wyliczanych w rozwiązaniu. Ustalmy $d \geqslant 1$. Możemy w łatwy sposób dla każdej reszty od 0 do $d-1$ wyznaczyć, ile liczb w ciągu ma tę resztę z dzielenia przez d. Wtedy ta reszta, której liczba wystąpień jest niepodzielna przez p, jest resztą z dzielenia przez d szukanej liczby.

Zamiast jednak zapisywać liczbę w systemie pozycyjnym o podstawie d, możemy wybrać kilka różnych liczb d_i i obliczyć resztę z dzielenia wyniku przez d_i. W ten sposób mamy wyznaczony wynik, jednak w dość nietypowej postaci – w formie reszt z dzielenia przez kilka wybranych przez nas liczb. Na szczęście umiemy na podstawie podanych liczb odzyskać wynik:

Twierdzenie 1 (Chińskie twierdzenie o resztach). *Mamy dane względnie pierwsze liczby naturalne* $1 \leqslant d_1, d_2, \ldots, d_q$ *oraz reszty* $0 \leqslant r_1 < d_1$, $0 \leqslant r_2 < d_2$, $\ldots$, $0 \leqslant r_q < d_q$. *Wtedy istnieje dokładnie jedna liczba całkowita* x, *że* $0 \leqslant x < d_1 d_2 \ldots d_q$ *oraz*

$$\begin{aligned} x \bmod d_1 &= r_1, \\ x \bmod d_2 &= r_2, \\ &\ldots, \\ x \bmod d_q &= r_q. \end{aligned}$$

Konstruktywny dowód tego twierdzenia można znaleźć w opracowaniu zadania *Permutacja* z XV Olimpiady Informatycznej [15].

Program `kol6.cpp` ustala $q = 2$ oraz liczby pierwsze $d_1 = 45\,007$ i $d_2 = 45\,011$. Ponieważ $d_1 d_2 > 2 \cdot 10^9$, to wynik jest wyznaczony poprawnie. Rozwiązanie otrzymuje maksymalną liczbę punktów.

Testy

Ponieważ dane wejściowe mogły być bardzo duże, niemożliwym było wczytywanie całego ciągu z pliku wejściowego. Wobec tego potrzebna była metoda pozwalająca na generowanie kolejnych elementów losowo wyglądającego ciągu, który jednak będzie miał własności opisane w treści zadania. Zainteresowanych Czytelników odsyłamy do implementacji generatora testów `kolingen.cpp`.