Wykrywanie wrednej usterki (Unscrambling a Messy Bug)

Ilshat jest inżynierem oprogramowania pracującym nad wydajnymi strukturami danych. Pewnego dnia wymyślił on nową strukturę danych. Potrafi ona przechowywać zbiór n *nieujemnych* liczb całkowitych, gdzie n jest potęgą dwojki (czyli $n = 2^b$ dla pewnego nieujemnego, całkowitego b).

Struktura danych początkowo jest pusta. Program używający tej struktury musi przestrzegać następujących zasad:

- Program może dodawać do struktury elementy, które są n-bitowymi liczbami całkowitymi, każdą pojedynczo, używając funkcji `add_element(x)`. Jeżeli program spróbuje dodać do struktury element już się w niej znajdujący, nic się nie dzieje.
- Po dodaniu ostatniego elementu program powinien wywołać funkcję `compile_set()` (dokładnie raz).
- Następnie program może wywoływać funkcję `check_element(x)`, aby sprawdzać, czy element x znajduje się w strukturze danych. Ta funkcja może być używana wielokrotnie.

Kiedy Ilshat pierwszy raz zaimplementował tę strukturę danych, miał on buga w funkcji `compile_set()`. Bug ten zmienia kolejność cyfr binarnych każdego elementu zbioru w pewien ustalony sposób. Ilshat chciałby, abyś znalazł dokładne przestawienie cyfr, które powoduje bug.

Formalnie, rozważmy ciąg $p = [p_0, \ldots, p_{n-1}]$, w którym każda liczba od 0 do $n-1$ występuje dokładnie raz. Taki ciąg nazywamy *permutacją*. Rozważmy element zbioru, którego cyfry w zapisie binarnym to kolejno $a_0, \ldots, a_{n-1}$ (gdzie a_0 jest najbardziej znaczącym bitem). W momencie wywołania funkcji `compile_set()`, element ten jest podmieniany na element $a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}}$.

Ta sama permutacja p jest używana do zamiany kolejności cyfr każdego elementu zbioru. Permutacja ta może być dowolna. W szczególności, może się zdarzyć, że $p_i = i$ dla każdego $0 \leq i \leq n-1$.

Dla przykładu, załóżmy, że $n = 4$, $p = [2, 1, 3, 0]$, a Ty dodałeś do struktury liczby, których binarna reprezentacja to `0000`, `1100` oraz `0111`. Wywołanie funkcji `compile_set` zmienia te elementy odpowiednio na `0000`, `0101` i `1110`.

Twoim zadaniem jest napisanie programu, który znajduje permutację p poprzez zadawanie zapytań do struktury danych. Program powinien (w następującej kolejności):

1. wybrać zbiór liczb n-bitowych,
2. dodać te liczby do struktury danych,
3. wywołać funkcję `compile_set`, aby spowodować buga,
4. sprawdzić istnienie pewnych elementów w strukturze,
5. użyć tych informacji aby odkryć i zwrócić permutację p.

Zauważ, że Twój program może wywołać funkcję `compile_set` jedynie raz.

Dodatkowo istnieją pewne ograniczenia na liczbę wywołań funkcji bibliotecznych. Twój program może

- wywołać funkcję `add_element` co najwyżej w razy (w jest od angielskiego słowa zapisy - "writes"),
- wywołać funkcję `check_element` co najwyżej r razy (r pochodzi od angielskiego słowa odczyty - "reads").

Szczegóły implementacji

Twoim zadaniem jest zaimplementowanie jednej funkcji (metody):

- `int[] restore_permutation(int n, int w, int r)`
  - `n`: liczba bitów w binarnej reprezentacji każdego elementu zbioru (a także długość p).
  - `w`: maksymalna liczba operacji `add_element`, którą może wykonać Twój program.
  - `r`: maksymalna liczba operacji `check_element`, którą może wykonać Twój program.
  - Funkcja powinna zwracać znalezioną permutację p.

W języku C sygnatura funkcji jest minimalnie inna:

- `void restore_permutation(int n, int w, int r, int* result)`
  - `n`, `w` i `r` mają takie same znaczenie jak powyżej.
  - Funkcja powinna zwracać znalezioną permutację p poprzez zapisanie jej do dostarczonej tablicy `result`: dla każdego i, powinna ona zapisać wartość p_i do `result[i]`.

Funkcje biblioteczne

Do komunikowania się ze strukturą danych Twój program powinien używać następujących trzech funkcji (metod):

- `void add_element(string x)`
  Funkcja ta dodaje element opisany przez `x` do zbioru.
  - `x`: napis złożony ze znaków `'0'` i `'1'` będący reprezentacją binarną liczby, która powinna być dodana do zbioru. Długość `x` musi wynosić n.
- `void compile_set()`
  Funkcja ta powinna być wywołana dokładnie raz. Twój program nie może wywołać funkcji `add_element()` po wywołaniu opisywanej funkcji. Twój program nie może także wywołać funkcji `check_element()` przed wywołaniem opisywanej funkcji.
- `boolean check_element(string x)`
  Ta funkcja sprawdza, czy element `x` znajduje się w zmodyfikowanym zbiorze.

- `x`: napis złożony ze znaków `'0'` oraz `'1'`, będący reprezentacją elementu, którego istnienie chcemy sprawdzić. Długość `x` musi wynosić n.
- zwraca `true`, jeżeli element `x` jest w zmodyfikowanym zbiorze, natomiast `false` w przeciwnym razie.

Pamiętaj, że jeżeli Twój program złamie którąś z wymienionych reguł, wynikiem jego sprawdzania będzie "Zła odpowiedź" ("Wrong Answer").

Dla każdego napisu pierwszy znak odpowiada za najbardziej znaczący bit odpowiadającej liczby.

Program sprawdzający ustala permutację p przed wywołaniem funkcji `restore_permutation`.

Szczegóły implementacji w Twoim języku programowania znajdują się w dostarczonych plikach z szablonami.

Przykład

Program sprawdzający wykonuje następujące wywołanie:

- `restore_permutation(4, 16, 16)`. Mamy $n = 4$, a program może wykonać co najwyżej 16 zapisów ("writes") i 16 odczytów ("reads").

Program zawodnika wykonuje następujące wywołania:

- `add_element("0001")`
- `add_element("0011")`
- `add_element("0100")`
- `compile_set()`
- `check_element("0001")` zwraca `false`
- `check_element("0010")` zwraca `true`
- `check_element("0100")` zwraca `true`
- `check_element("1000")` zwraca `false`
- `check_element("0011")` zwraca `false`
- `check_element("0101")` zwraca `false`
- `check_element("1001")` zwraca `false`
- `check_element("0110")` zwraca `false`
- `check_element("1010")` zwraca `true`
- `check_element("1100")` zwraca `false`

Jest tylko jedna permutacja zgodna z wynikami funkcji `check_element()`, a mianowicie permutacja $p = [2, 1, 3, 0]$. Tak więc `restore_permutation` powinna zwrócić `[2, 1, 3, 0]`.

Podzadania

1. (20 punktów) $n = 8$, $w = 256$, $r = 256$, $p_i \neq i$ zachodzi dla co najwyżej dwóch indeksów i ($0 \leq i \leq n - 1$),
2. (18 punktów) $n = 32$, $w = 320$, $r = 1024$,
3. (11 punktów) $n = 32$, $w = 1024$, $r = 320$,
4. (21 punktów) $n = 128$, $w = 1792$, $r = 1792$,

5. (30 punktów) $n = 128$, $w = 896$, $r = 896$.

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- wiersz 1: liczby całkowite n, w, r,
- wiersz 2: n liczb całkowitych będących elementami p.