

## Unscrambling a Messy Bug

Ilshat este un programator care lucrează la dezvoltarea unor structuri de date eficiente. Într-o zi, a inventat o nouă structură de date. Această structură de date poate stoca o mulțime de întregi *nenegativi* pe  $n$  biți, unde  $n$  este o putere a lui 2. Mai exact,  $n = 2^b$  pentru un întreg nenegativ  $b$ .

Structura de date este goală inițial. Un program care folosește această structură trebuie să respecte următoarele reguli:

- Programul poate insera întregi de  $n$  biți în structura de date, unul câte unul, utilizând funcția `add_element(x)`. Dacă programul încearcă să insereze un element care este deja prezent în structură, nu se întâmplă nimic.
- După ce a inserat ultimul element, programul trebuie să apeleze funcția `compile_set()` exact o dată.
- În final, programul poate apela funcția `check_element(x)` pentru a verifica dacă elementul  $x$  este prezent în structura de date. Această funcție poate fi apelată de mai multe ori.

Când Ilshat a implementat prima oară această structură de date, a creat un bug în funcția `compile_set()`. Acest bug reordonează cifrele binare ale fiecărui element din mulțime în aceeași manieră. Ilshat vrea ca voi să găsiți exact în ce fel au fost reordonate cifrele ca urmare a acestui bug.

Formal, considerați o secvență  $p = [p_0, \dots, p_{n-1}]$  în care fiecare număr de la 0 la  $n - 1$  apare exact o dată. Numim o asemenea secvență o *permutare*. Considerați un element din mulțime, ale cărui cifre în binar sunt  $a_0, \dots, a_{n-1}$  (cu  $a_0$  fiind cel mai semnificativ bit). Când funcția `compile_set()` este apelată, acest element este înlocuit de elementul  $a_{p_0}, a_{p_1}, \dots, a_{p_{n-1}}$ .

Aceeași permutare  $p$  este folosită pentru a reordona cifrele tuturor elementelor. Permutarea este arbitrară, poate fi inclusiv adevărat că  $p_i = i$  pentru toți  $0 \leq i \leq n - 1$ .

Spre exemplu, să presupunem că  $n = 4$ ,  $p = [2, 1, 3, 0]$  și ați inserat în set întregii care au ca reprezentări binare stringurile `0000`, `1100` și `0111`. Apelând funcția `compile_set` aceste elemente se vor schimba în `0000`, `0101` și `1110`, respectiv.

Sarcina voastră este să scrieți un program care găsește permutarea  $p$  interacționând cu structura de date. Acest program ar trebui să facă următoarele lucruri (în această ordine):

1. să aleagă o mulțime de întregi de  $n$  biți,
2. să insereze acei întregi în structura de date,

3. să apeleze funcția `compile_set` pentru a declanșa bug-ul,
4. să verifice prezența unor elemente din mulțimea modificată,
5. să folosească informația primită pentru a determina și a întoarce permutarea  $p$ .

Rețineți că programul vostru poate apela funcția `compile_set` exact o dată.

În plus, numărul de apeluri ale funcțiilor din librărie este limitat. Mai exact, programul poate

- apela `add_element` de cel mult  $w$  ori ( $w$  de la "writes"),
- apela `check_element` de cel mult  $r$  ori ( $r$  de la "reads").

## Detalii de implementare

Trebuie să implementați o funcție (metodă):

- `int[] restore_permutation(int n, int w, int r)`
  - $n$ : numărul de biți din reprezentarea binară a fiecărui element al mulțimii (de-asemenea, lungimea lui  $p$ ).
  - $w$ : numărul maxim de apeluri ale funcției `add_element` pe care le poate face programul vostru.
  - $r$ : numărul maxim de apeluri ale funcției `check_element` pe care le poate face programul vostru.
  - funcția va returna permutarea descoperită,  $p$ .

În limbajul C, funcția are o semnătură puțin diferită:

- `void restore_permutation(int n, int w, int r, int* result)`
  - $n, w$  și  $r$  au aceeași semnificație ca mai sus.
    - funcția va returna permutarea descoperită,  $p$ , stocând-o în array-ul furnizat, `result`: pentru fiecare  $i$ , va stoca  $p_i$  în `result[i]`.

## Funcțiile din librărie

Pentru a interacționa cu structura de date, programul vostru va folosi următoarele trei funcții (metode):

- `void add_element(string x)`

Această funcție inserează elementul descris de  $x$  în mulțime.

  - $x$ : un string de caractere '0' și '1' semnificând reprezentarea binară a întregului care va fi inserat. Lungimea lui  $x$  trebuie să fie egală cu  $n$ .
- `void compile_set()`

Această funcție trebuie apelată exact o dată. Programul vostru nu poate apela `add_element()` după acest apel. Programul vostru nu poate apela `check_element()` înainte de acest apel.
- `boolean check_element(string x)`

Această funcție verifică dacă elementul  $x$  este în mulțimea alterată.

  - $x$ : un string de caractere '0' and '1' semnificând reprezentarea binară a întregului care va fi interogată. Lungimea lui  $x$  trebuie să fie egală cu  $n$ .
  - întoarce `true` dacă elementul  $x$  este în setul alterat, și `false` altfel.

Dacă programul vostru încalcă vreuna din restricțiile de mai sus, rezultatul evaluării va fi "Wrong Answer".

Pentru toate stringurile, primul caracter denotă cel mai semnificativ bit al întregului corespunzător.

Grader-ul fixează permutarea  $p$  înainte ca funcția `restore_permutation` să fie apelată.

Folosiți fișierele template furnizate pentru detalii de implementare în limbajul vostru de programare.

## Exemple

Grader-ul face următorul apel:

- `restore_permutation(4, 16, 16)`. Avem  $n = 4$  și programul poate face cel mult 16 "writes" și 16 "reads".

Programul face următoarele apeluri de funcții:

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

O singură permutare respectă răspunsurile furnizate de apelurile funcției `check_element()`: permutarea  $p = [2, 1, 3, 0]$ . Astfel, `restore_permutation` ar trebui să întoarcă `[2, 1, 3, 0]`.

## Subtaskuri

1. (20 de puncte)  $n = 8$ ,  $w = 256$ ,  $r = 256$ , iar  $p_i \neq i$  pentru cel mult 2 indici  $i$  ( $0 \leq i \leq n - 1$ ),
2. (18 puncte)  $n = 32$ ,  $w = 320$ ,  $r = 1024$ ,
3. (11 puncte)  $n = 32$ ,  $w = 1024$ ,  $r = 320$ ,
4. (21 de puncte)  $n = 128$ ,  $w = 1792$ ,  $r = 1792$ ,
5. (30 de puncte)  $n = 128$ ,  $w = 896$ ,  $r = 896$ .

## Sample grader

Sample grader-ul citește input-ul în următorul format:

- linia 1: întregii  $n$ ,  $w$ ,  $r$ ,
- linia 2:  $n$  întregi, reprezentând elementele lui  $p$ .