Unscrambling a Messy Bug

Илшад е софтуерен инженер, работещ върху ефективни структури от данни. Един ден той измисли нова структура от данни. Тази структура може да съхранява множество от *цели неотрицателни* n-битови числа, където n е степен на двойката. Така $n = 2^b$ за някое неотрицателно цяло число b.

Първоначално тази структура от данни е празна. Програма, която използва тази структура от данни, трябва да спазва следните правила:

- Програмата може да добавя в тази структура елементи, които са n-битови цели числа, използвайки функцията add_element(x), като при всяко извикване на функцията се добавя по едно число. Ако програмата се опита да добави елемент, който вече съществува в структурата от данни, той не се добавя и структурата не променя съдържанието си.
- След добавяне на последния елемент, програмата трябва да извика функцията compile_set() точно веднъж.
- Накрая, програмата може да извика функцията check_element(x), за да провери дали елементът x е в структурата от данни. Тази функция може да се използва многократно.

Когато Илшат за първи път имплементира тази структура от данни, той допусна бъг във функцията compile_set(). Този бъг пренарежда двоичните цифри на всеки елемент от множеството по един и същи начин. Илшат иска да намери как точно този бъг пренарежда цифрите на числата.

Формално, разглеждаме редицата $p = [p_0, \ldots, p_{n-1}]$, в която всяко от целите числа от 0 до $n-1$ участва точно по веднъж. Ще наричаме такава редица *пермутация*. Да разгледаме елемент от множеството, чиито цифри в двоичното представяне са $a_0, \ldots, a_{n-1}$ (a_0 е старшия бит). При извикването на функцията compile_set() този елемент се заменя с елемента $a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}}$.

Същата пермутация p се използва за пренареждането на цифрите на всички елементи на множеството. Всяка пермутация е възможна, включително е възможно $p_i = i$ за всяко $0 \leq i \leq n-1$.

Например, нека $n = 4$, $p = [2, 1, 3, 0]$ и да сме добавили в множеството цели числа, чиито двоични представяния са 0000, 1100 и 0111. След извикване на функцията compile_set тези числа се преобразуват съответно в 0000, 0101 и 1110.

Вашата задача е да напишете програма, която намира пермутацията p, взаимодействайки си със структурата от данни. Програмата трябва (в

посочената последователност):

1. да избере множество от n-битови цели числа,
2. да вмъкне тези числа в структурата от данни,
3. да извика функцията `compile_set`, която да задейства бъга,
4. да провери дали някои елементи са в модифицираното множество,
5. да използва тази информация, за да открие пермутацията p.

Обърнете внимание, че вашата програма може да извика функцията `compile_set` само веднъж.

В допълнение, има ограничение за броя на извикванията на библиотечните функции. По-точно, може:

- да извикате `add_element` най-много w пъти (w е от "writes"),
- да извикате `check_element` най-много r пъти (r е от "reads").

Детайли по имплементацията

Вие трябва да имплементирате една функция (метод):

- `int[] restore_permutation(int n, int w, int r)`
  - `n`: броят на битовете в двоичното представяне на всеки елемент от множеството (а също и дължината на p).
  - `w`: максималният брой извиквания на функцията `add_element` от вашата програма.
  - `r`: максималният брой извиквания на функцията `check_element` от вашата програма.
  - функцията трябва да върне възстановената пермутация p.

За езика C прототипът на функцията е малко по-различен:

- `void restore_permutation(int n, int w, int r, int* result)`
  - `n`, `w` и `r` имат същите значения като гореописаните.
  - функцията трябва да върне възстановената пермутация p като я запише в масива `result`: за всяко i, стойността на p_i трябва да се запише в `result[i]`.

Библиотечни функции

За да взаимодейства със структурата от данни, вашата програма трябва да използва следните три функции (метода):

- `void add_element(string x)`
  Тази функция добавя елемента `x` в множеството.
  - `x`: низ от символи `'0'` и `'1'`, описващ двоичното представяне на числото, което трябва да се добави в множеството. Дължината на `x` трябва да бъде n.

- `void compile_set()`
  Тази функция трябва да бъде извикана точно веднъж. Вашата програма не може да извиква функцията `add_element()` след това извикване. Вашата програма не може да извика функцията `check_element()` преди това извикване.

- `boolean check_element(string x)`
  Тази функция проверява дали елементът `x` е в модифицираното множество.
  - `x`: низ от символи `'0'` и `'1'`, описващ двоичното представяне на елемента, който трябва да бъде проверен. Дължината на `x` трябва да бъде n.
  - връща `true`, ако елементът `x` е в модифицираното множество, връща `false` в противен случай.

Имайте предвид, че ако вашата програма наруши някое от тези ограничения, ще получите съобщение "Wrong Answer".

За всички низове първият символ съответства на старшия бит в двоичното представяне на числото.

Грейдърът фиксира пермутацията p преди извикването на функцията `restore_permutation`.

Моля използвайте предоставените шаблонни файлове за подробности, свързани с имплементацията на програмния език, който използвате.

Пример

Грейдърът извършва следното извикване на функция:

- `restore_permutation(4, 16, 16)`. Имаме $n = 4$ и програмата може да направи най-много 16 операции "writes" и 16 операции "reads".

Програмата извършва следните извиквания на функции:

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

Само една пермутация е съвместима с върнатите стойности от функцията `check_element()`: пермутацията $p = [2, 1, 3, 0]$. Следователно `restore_permutation` трябва да върне `[2, 1, 3, 0]`.

Подзадачи

1. (20 точки) $n = 8$, $w = 256$, $r = 256$, $p_i \neq i$ за най-много два индекса i (

$0 \leq i \leq n-1$),
2. (18 точки) $n = 32$, $w = 320$, $r = 1024$,
3. (11 точки) $n = 32$, $w = 1024$, $r = 320$,
4. (21 точки) $n = 128$, $w = 1792$, $r = 1792$,
5. (30 точки) $n = 128$, $w = 896$, $r = 896$.

Примерен грейдър

Примерният грейдър чете входа в следния формат:

- ред 1: цели числа n, w, r,
- ред 2: n цели числа, описващи елементите на p.