Razreševanje nadležnega hrošča

Ilshat je razvijalec programske opreme in trenutno razvija napredne podatkovne strukture. Razvil je novo podatkovno strukturo, ki je zmožna hraniti množico *nenegativnih* n-bitnih celih števil, kjer je n potenca števila dva (2). Torej je $n = 2^b$ za neko nenegativno celo število b.

Podatkovna struktura je na začetku prazna. Program, ki uporablja to podatkovno strukturo, se drži naslednjih pravil:

- Program lahko vstavlja v podatkovno strukturo elemente, ki so n-bitna cela števila. Vstavlja jih lahko enega po enega, s klicem funkcije `add_element(x)`. Če program poizkusi vstaviti element, ki je že v podatkovni strukturi, se ne zgodi nič.
- Ko program vstavi zadnji element, mora poklicati funkcijo `compile_set()`. To lahko stori le enkrat.
- Na koncu lahko program kliče funkcijo `check_element(x)`, s čimer preveri, ali element x obstaja v podatkovni strukturi. To funkcijo lahko kliče večkrat.

Ko je Ilshat prvič implementiral to podatkovno strukturo, je odkril, da v funkciji `compile_set()` obstaja nadležen hrošč. Ta hrošč preuredi binarne številke vsakega elementa v množici po istem vzorcu. Ilshat te prosi, da poiščeš natančno preureditev številk, ki jo povzroči hrošč.

Bolj formalno: predstavljajmo si zaporedje $p = [p_0, \ldots, p_{n-1}]$, v katerem se vsako število od 0 do $n-1$ pojavi natanko enkrat. Takšno zaporedje imenujemo *permutacija*. Vzemimo element množice, čigar binarne številke so $a_0, \ldots, a_{n-1}$ (kjer je a_0 najpomembnejši bit - MSB). Ko se pokliče funkcijo `compile_set()`, se ta element spremeni v element $a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}}$.

Vemo, da se vsi elementi preuredijo po poljubni, vendar isti, permutaciji p. Obstaja možnost, da je $p_i = i$ za vsak $0 \leq i \leq n-1$.

Na primer, denimo da je $n = 4$, $p = [2, 1, 3, 0]$ in da smo v množico vstavili števila, katerih binarni zapisi so: `0000`, `1100` in `0111`. Po klicu funkcije `compile_set` se ti elementi spremenijo v: `0000`, `0101` in `1110`.

Tvoja naloga je napisati program, ki določi permutacijo p preko interakcij s podatkovno strukturo. Upoštevati mora naslednje zaporedje:

1. Izbere množico n-bitnih celih števil,
2. ta števila vstavi v podatkovno strukturo,
3. pokliče funkcijo `compile_set`, s čimer sproži hrošča,
4. v preurejeni množici preveri prisotnost nekaterih elementov,

5. uporabi pridobljene informacije, da določi in vrne permutacijo p.

Ne pozabi, da lahko tvoj program pokliče fukcijo `compile_set` le enkrat.

Dodatno velja, da obstajajo omejitve, koliko krat lahko tvoj program kliče funkcije knjižnice. Natančnjene lahko

- kliče funkcijo `add_element` največ w krat (w velja za "zapisovanje"),
- kliče funkcijo `check_element` največ r krat (r velja za "branje").

Opombe k implementaciji

Implementirati moraš eno funkcijo (metodo):

- `int[] restore_permutation(int n, int w, int r)`
  - `n`: število bitov v binarnem zapisu vsakega elementa množice (tudi dolžina p.
  - `w`: največe dovoljeno število klicov funkcije `add_element`.
  - `r`:največje dovoljeno število klicov funkcije `check_element`.
  - Funkcija naj vrne permutacijo p.

V programskem jeziku C je podpis funkcije nekoliko drugačen:

- `void restore_permutation(int n, int w, int r, int* result)`
  - `n`, `w` in `r` imajo enako vlogo kot zgoraj.
  - Funkcija naj vrne permutacijo p tako, da jo shrani v dodeljeno polje `result`: za vsak i naj shrani vrednost p_i v `result[i]`.

Funkcije knjižnice

Da lahko tvoj program komunicira s podatkovno strukturo, mora program uporabiti sledeče tri funkcije (metode):

- `void add_element(string x)`
  Ta funkcija vstavi element, opisan z `x`, v množico.
  - `x`: niz `'0'` in `'1'`, ki predstavlja binarni zapis števila, ki ga želimo vstaviti v množico. Dolžina `x` mora biti natanko n.

- `void compile_set()`
  Ta funkcija mora biti poklicana natanko enkrat. Za tem program ne sme več klicati funkcije `add_element()`. Prav tako pa program ne more klicati funkcije `check_element()`, preden ne pokliče funkcije `compile_set()`.

- `boolean check_element(string x)`
  Ta funkcija preveri, ali je element `x` v spremenjeni množici.
  - `x`: niz `'0'` in `'1'`, ki predstavlja binarni zapis števila, ki ga preverjamo. Dolžina `x` mora biti natanko n.
  - Vrne `true`, če je element `x` v spremenjeni množici, in `false`, če ni.

Pomni, da tvoj program ne sme kršiti nobene izmed zgoraj opisanih omejitev, sicer bo ocenjevalni sistem vrnil "Napačen odgovor".

Za vse nize velja, da prvi znak predstavlja najpomembnejši bit (MSB) števila.

Ocenjevalnik določi permutacijo p pred klicem funkcije `restore_permutation`.

Uporabi predložne datoteke za več informacij o implementaciji v izbranem programskem jeziku.

Primer

Ocenjevalnik izvede naslednji klic funkcije:

- `restore_permutation(4, 16, 16)`. Imamo $n = 4$, program pa lahko izvede največ 16 "zapisovanj" in 16 "branj".

Program izvede naslednje klice funkcij:

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

Edina permutacija, ki ustreza odgovorom funkcije `check_element()`, je permutacija $p = [2, 1, 3, 0]$. Zato `restore_permutation` vrne `[2, 1, 3, 0]`.

Podnaloge

1. (20 točk) $n = 8$, $w = 256$, $r = 256$, $p_i \neq i$ za največ dva (2) indeksa i ($0 \leq i \leq n - 1$),
2. (18 točk) $n = 32$, $w = 320$, $r = 1024$,
3. (11 točk) $n = 32$, $w = 1024$, $r = 320$,
4. (21 točk) $n = 128$, $w = 1792$, $r = 1792$,
5. (30 točk) $n = 128$, $w = 896$, $r = 896$.

Vzorčni ocenjevalnik

Vzorčni ocenjevalnik bere vhod sledeče oblike:

- $1.$ vrstica: cela števila n, w, r.
- $2.$ vrstica: n števil, ki podajajo elemente p.