Un bug difficile da decifrare

Ilshat è un programmatore che lavora su strutture dati efficienti. Un giorno ha inventato una nuova struttura dati, che è in grado di memorizzare un insieme di interi *non-negativi* di n bit, dove n è una potenza di due (ovvero $n = 2^b$ per un qualche intero non-negativo b).

La struttura dati è inizialmente vuota. Ogni programma che usa tale struttura dati deve seguire le seguenti regole:

- Il programma può aggiungere interi di n bit alla struttura dati, uno alla volta, usando la funzione `add_element(x)`. Se il programma prova ad aggiungere un elemento già presente nella struttura dati, non succede nulla.
- Una volta aggiunti tutti gli elementi, il programma deve chiamare la funzione `compile_set()` esattamente una volta.
- Infine, il programma può chiamare la funzione `check_element(x)` per verificare se un certo intero x è presente nella struttura dati. Questa funzione può essere usata più volte.

Quando Ilshat ha implementato questa struttura dati per la prima volta, ha introdotto un bug nella funzione `compile_set()`. Tale bug riordina le cifre binarie di ogni elemento dell'insieme nello stesso modo. Ilshat vuole che tu trovi esattamente in che modo le cifre vengono permutate dal bug.

Formalmente, consideriamo una sequenza $p = [p_0, \ldots, p_{n-1}]$ in cui ogni numero da 0 a $n-1$ appare esattamente una volta. Diciamo che tale sequenza è una *permutazione*. Consideriamo un elemento inserito nella struttura dati, le cui cifre in base due sono $a_0, \ldots, a_{n-1}$ (con a_0 che è la cifra più significativa). Quando la funzione `compile_set()` viene chiamata, tale elemento è sostituito da $a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}}$.

La stessa permutazione p viene usata per riordinare le cifre di ogni elemento della struttura dati. Ogni permutazione è possibile, e in particolare può essere che $p_i = i$ per ogni $0 \leq i \leq n-1$.

Per esempio, supponiamo che $n = 4$, $p = [2, 1, 3, 0]$, e sono stati inseriti nell'insieme gli interi la cui rappresentazione binaria è `0000`, `1100`, `0111`. Chiamare la funzione `compile_set` cambia tali elementi in `0000`, `0101` e `1110` rispettivamente.

Il tuo compito è scrivere un programma che trovi la permutazione p interagendo con la struttura dati. Deve, nel seguente ordine:

1. scegliere un insieme di interi a n bit,
2. inserire tali interi nella struttura dati,
3. chiamare la funzione `compile_set` per causare il bug,
4. controllare se alcuni interi sono presenti nel set,
5. usare tale informazione per individuare e restituire p.

Nota che il tuo programma può chiamare la funzione `compile_set` solo una volta.

Inoltre, c'è un limite al numero di volte che il tuo programma può chiamare le funzioni di libreria, ovvero può:

- chiamare `add_element` al più w volte (w è "writes"),
- chiamare `check_element` al più r volte (r è "reads").

Dettagli di implementazione

Devi implementare la seguente funzione (metodo):

- `int[] restore_permutation(int n, int w, int r)`
  - `n`: il numero di bit nella rappresentazione binaria di ogni elemento dell'insieme (e anche la lunghezza di p).
  - `w`: il massimo numero di operazioni `add_element` che il tuo programma può eseguire.
  - `r`: il massimo numero di operazioni `check_element` che il tuo programma può eseguire.
  - la funzione deve restituire la permutazione p.

Nel linguaggio C, la signature della funzione è leggermente diversa:

- `void restore_permutation(int n, int w, int r, int* result)`
  - `n`, `w` e `r` hanno lo stesso significato di sopra.
  - la funzione deve restituire la permutazione p salvandola nell'array `result` fornito: per ogni i, deve scrivere p_i in `result[i]`.

Funzioni di libreria

Per interagire con la struttura dati, il tuo programma deve usare le seguenti tre funzioni (metodi):

- `void add_element(string x)`
  Questa funzione aggiunge `x` all'insieme.
  - `x`: una stringa di n caratteri `'0'` e `'1'` che fornisce la rappresentazione binaria dell'intero che deve essere aggiunto all'insieme.

- `void compile_set()`
  Questa funzione deve essere eseguita esattamente una volta. Il tuo programma non può eseguire `add_element()` dopo questa chiamata. Il tuo programma non può eseguire `check_element()` prima di questa chiamata.

- `boolean check_element(string x)`
  Questa funzione controlla se `x` appartiene all'insieme modificato.
  - `x`: una stringa di n caratteri `'0'` e `'1'` che fornisce la rappresentazione binaria dell'intero che deve essere controllato. La lunghezza di `x` *deve* essere n.
  - restituisce `true` se un elemento `x` appartiene all'insieme modificato, e `false` altrimenti.

Nota che se il tuo programma viola una qualsiasi delle restrizioni precedenti, il risultato della sua esecuzione sarà "Wrong Answer".

Per ogni stringa, il primo carattere fornisce il bit più significativo dell'intero corrispondente.

Il grader decide la permutazione p prima che la funzione `restore_permutation` sia chiamata.

Vedi i template forniti per ulteriori dettagli di implementazione nel tuo linguaggio di programmazione.

Esempio

Supponiamo che grader esegua la seguente chiamata a funzione:
- `restore_permutation(4, 16, 16)`

Abbiamo quindi $n = 4$ e il programma può eseguire al massimo 16 "writes" e 16 "reads".

Il programma esegue quindi le seguenti chiamate a funzione:
- `add_element("0001")`
- `add_element("0011")`
- `add_element("0100")`
- `compile_set()`
- `check_element("0001")` restituisce `false`
- `check_element("0010")` restituisce `true`
- `check_element("0100")` restituisce `true`
- `check_element("1000")` restituisce `false`
- `check_element("0011")` restituisce `false`
- `check_element("0101")` restituisce `false`
- `check_element("1001")` restituisce `false`
- `check_element("0110")` restituisce `false`
- `check_element("1010")` restituisce `true`
- `check_element("1100")` restituisce `false`

Solo una permutazione è compatibile con i valori restituiti da `check_element()`: la permutazione $p = [2, 1, 3, 0]$.

Dunque, `restore_permutation` deve restituire `[2, 1, 3, 0]`.

Subtask

1. (20 punti) $n = 8$, $w = 256$, $r = 256$, $p_i \neq i$ per al massimo due indici i ($0 \leq i \leq n-1$),
2. (18 punti) $n = 32$, $w = 320$, $r = 1024$,
3. (11 punti) $n = 32$, $w = 1024$, $r = 320$,
4. (21 punti) $n = 128$, $w = 1792$, $r = 1792$,
5. (30 punti) $n = 128$, $w = 896$, $r = 896$.

Grader di esempio

Il grader di esempio legge l'input nel seguente formato:

- riga 1: i tre interi n, w, r,
- riga 2: gli n interi $p_0, \ldots, p_{n-1}$.