messy
Country: DEU

Unordnung

Der Software-Entwickler Ilshat hat eine neue Datenstruktur erfunden. Diese speichert eine Menge nicht-negativer n-Bit-Integers in Binärdarstellung, also Bitstrings der Länge n. Dabei ist n eine Zweierpotenz.

Die Datenstruktur wird wie folgt verwendet:

- Zu Beginn ist die Datenstruktur leer.
- Mit der Funktion `add_element(x)` fügt man nach und nach je ein Element in die Datenstruktur ein. Ist ein einzufügendes Element in der Struktur bereits vorhanden, geschieht nichts.
- Die Funktion `compile_set()` ruft man genau einmal auf, nachdem alle Elemente eingefügt sind.
- Mit der Funktion `check_element(x)` kann man anschließend prüfen, ob ein Element x in der Datenstruktur vorhanden ist. Die Funktion kann mehrfach aufgerufen werden.

Ilshats Implementierung der Funktion `compile_set()` hat einen Bug: In jedem der gespeicherten Bitstrings werden bei einem Aufruf der Funktion auf gleiche Weise die Bits umgeordnet. Ilshat möchte, dass du herausfindest wie.

Gesucht ist also eine Permutation $p = (p_0, \ldots, p_{n-1})$ der Zahlen von 0 bis $n-1$, sodass gilt: Ein Aufruf der Funktion `compile_set()` ersetzt jeden gespeicherten Bitstring $a_0, \ldots, a_{n-1}$ durch den Bitstring $a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}}$.

Beachte, dass auch $p_i = i$ für alle $0 \leq i \leq n-1$ gelten kann.

Ein Beispiel: Es sind $n = 4$ und $p = (2, 1, 3, 0)$, und die Bitstrings `0000`, `1100` und `0111` sind in der Datenstruktur gespeichert. Der Aufruf der Funktion `compile_set()` ersetzt diese Strings durch `0000`, `0101` bzw. `1110`.

Schreibe ein Programm, das durch Interaktion mit der Datenstruktur die Permutation p herausfindet. Es soll in der folgenden Reihenfolge:

1. eine Menge von Bitstrings der Länge n wählen,
2. sie mit der Funktion `add_element(x)` in die Datenstruktur einfügen,
3. die Funktion `compile_set()` aufrufen,
4. mit der Funktion `check_element(x)` das Vorhandensein von Bitstrings in der Struktur überprüfen und
5. die dabei gewonnenen Informationen nutzen, um die Permutation p herauszufinden und zurückzugeben.

Für jede Funktion ist die Anzahl ihrer Aufrufe beschränkt: Dein Programm darf

- `add_element(x)` höchstens w-mal aufrufen (w steht für "writes"),

- `compile_set()` genau einmal aufrufen und
- `check_element(x)` höchstens r-mal aufrufen (r steht für "reads").

Implementierungsdetails

Du sollst folgende Funktion (Methode) implementieren:

- `int[] restore_permutation(int n, int w, int r)`
  - `n`: die Länge der in der Datenstruktur gespeicherten Bitstrings (und auch die Länge von p).
  - `w`: die maximale Anzahl der Aufrufe von `add_element(x)`.
  - `r`: die maximale Anzahl der Aufrufe von `check_element(x)`.
  - Die Funktion soll die herausgefundene Permutation p zurückgeben.

In der Programmiersprache C ist die Funktionssignatur ein wenig anders:

- `void restore_permutation(int n, int w, int r, int* result)`
  - `n`, `w` und `r` haben die gleiche Bedeutung wie oben.
  - Die Funktion soll die herausgefundene Permutation p zurückgeben, indem sie sie in das bereitgestellte Array `result` schreibt: für jedes i soll sie den Wert p_i in `result[i]` speichern.

Library-Funktionen

Um mit der Datenstruktur zu interagieren, soll dein Programm die folgenden drei Funktionen (Methoden) benutzen:

- `void add_element(string x)`
  Diese Funktion fügt ein durch `x` beschriebenes Element in die Datenstruktur ein.
  - `x`: ein String von Zeichen `'0'` und `'1'`. Die Länge von `x` muss genau n sein.

- `void compile_set()`
  Diese Funktion muss genau einmal aufgerufen werden. Dein Programm darf nach diesem Aufruf die Funktion `add_element()` nicht mehr aufrufen. Dein Programm darf die Funktion `check_element()` vor diesem Aufruf nicht aufrufen.
- `boolean check_element(string x)`
  Diese Funktion prüft, ob das durch `x` beschriebene Element in der Datenstruktur vorhanden ist.
  - `x`: ein String von Zeichen `'0'` und `'1'`. Die Länge von `x` muss genau n sein.
  - Gibt `true` zurück, falls das Element `x` in der veränderten Menge ist, und sonst `false`.

Falls dein Programm irgendeine der Vorgaben oben verletzt, wird das Bewertungsergebnis "Wrong Answer" sein.

Der Grader legt die Permutation p fest, bevor die Funktion `restore_permutation` aufgerufen wird.

Die zur Verfügung gestellten Vorlagedateien zeigen dir die Details in deiner

Programmiersprache.

Beispiel

Der Grader macht folgenden Funktionsaufruf:

- `restore_permutation(4, 16, 16)`. Es gilt $n = 4$ und das Programm kann höchstens 16 "writes" und 16 "reads" ausführen.

Das Programm macht folgende Funktionsaufrufe:

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

Nur eine Permutation ist konsistent mit den Werten, die `check_element()` zurückgibt: die Permutation $p = (2, 1, 3, 0)$. Daher soll `restore_permutation` das Array `[2, 1, 3, 0]` zurückgeben.

Teilaufgabe

1. (20 Punkte) $n = 8$, $w = 256$, $r = 256$, $p_i \neq i$ für höchstens 2 Indizes i ($0 \leq i \leq n - 1$),
2. (18 Punkte) $n = 32$, $w = 320$, $r = 1024$,
3. (11 Punkte) $n = 32$, $w = 1024$, $r = 320$,
4. (21 Punkte) $n = 128$, $w = 1792$, $r = 1792$,
5. (30 Punkte) $n = 128$, $w = 896$, $r = 896$.

Beispielgrader

Der Beispielgrader liest die Eingabe im folgenden Format:

- Zeile 1: Integers n, w, r,
- Zeile 2: n Integers, die Elemente von p.