Décrypter un bug incompréhensible

Ilshat, un informaticien travaillant sur les structures de données efficaces, en a inventé une nouvelle. Cette structure de données peut stocker un ensemble d'entiers *positifs* de n bits, où n est une puissance de 2, c'est-à-dire $n = 2^b$ pour un entier positif b donné.

La structure de données est initialement vide. Un programme utilisant la structure de données doit suivre les règles suivantes :

- Le programme peut ajouter des entiers de n bits dans la structure de données, un à la fois, en utilisant la fonction `add_element(x)`. Si le programme essaie d'ajouter un élément qui est déjà présent dans la structure, l'appel n'a aucun effet.
- Après avoir ajouté le dernier élément, le programme doit appeler la fonction `compile_set()` exactement une fois.
- Pour finir, le programme peut appeler la fonction `check_element(x)` pour vérifier si l'élément x est présent dans la structure de données. Cette fonction peut être appelée à plusieurs reprises.

Quand Ilshat a implémenté cette structure de données, il a introduit un bug dans la fonction `compile_set()`. Le bug provoque le réordonnancement des chiffres binaires de chaque élément de l'ensemble de la même manière. Ilshat veut que vous trouviez le réordonnancement exact des chiffres provoqué par le bug.

Formellement, si on considère une séquence $p = [p_0, \ldots, p_{n-1}]$ dans laquelle chaque nombre de 0 à $n-1$ apparaît exactement une fois. On appelle une telle séquence une *permutation*. Considérons un élément de l'ensemble, dont les chiffres en binaire sont $a_0, \ldots, a_{n-1}$ (a_0 étant le bit de poids fort). Quand la fonction `compile_set()` est appelée, cet élément est remplacé par l'élément $a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}}$.

La même permutation p est utilisée pour réordonner les chiffres de chaque élément. Toute permutation est possible, y compris la possibilité que $p_i = i$ pour chaque $0 \leq i \leq n-1$.

Par exemple, supposons que $n = 4$, $p = [2, 1, 3, 0]$ et que vous avez inséré dans l'ensemble les entiers dont les représentations binaires sont `0000`, `1100` et `0111`. Appeler la fonction `compile_set` modifie ces éléments en repectivement `0000`, `0101` et `1110`.

Votre tâche est d'écrire un programme qui trouve la permutation p en interagissant avec la structure de données. Il doit (dans cet ordre) :

1. choisir un ensemble d'entiers de n bits,
2. insérer ces entiers dans la structure de données,
3. appeler la fonction `compile_set` pour provoquer le bug,
4. vérifier la présence de certains éléments dans l'ensemble modifié,
5. utiliser cette information pour déterminer et retourner la permutation p.

Notez que votre programme ne peut appeler la fonction `compile_set` qu'une seule fois.

De plus, il y a une limite sur le nombre d'appels que votre programme peut faire sur les fonctions de la librairie, c-à-d. il peut :

- appeler `add_element` au plus w fois (w est pour "writes"),
- appeler `check_element` au plus r fois (r est pour "reads").

Détails d'implémentation

Vous devez implémenter une fonction (méthode) :

- `int[] restore_permutation(int n, int w, int r)`
  - `n` : le nombre de bits dans la représentation binaire de chaque élément de l'ensemble (également la taille de p).
  - `w` : le nombre maximal d'opérations `add_element` que votre programme peut effectuer.
  - `r` : le nombre maximal d'opérations `check_element` que votre programme peut effectuer.
  - la fonction doit retourner la permutation trouvée p.

En C, la signature de la fonction prototype est légèrement différente :

- `void restore_permutation(int n, int w, int r, int* result)`
  - `n`, `w` et `r` ont la même signification que ci-dessus.
  - la fonction doit retourner la permutation p en la stockant dans le tableau `result` fourni : pour chaque i, il doit stocker la valeur p_i dans `result[i]`.

Fonctions de la librairie

Afin d'intéragir avec la structure de données, votre programme doit utiliser les trois fonctions (méthodes) suivantes :

- `void add_element(string x)`
  Cette fonction ajoute l'élément décrit par `x` à l'ensemble.
  - `x` : une chaîne de caractères de `'0'` et `'1'` donnant la représentation binaire d'un entier qui doit être ajouté à l'ensemble. La longueur de `x` doit être n.

- `void compile_set()`
  Cette fonction doit être appelée exactement une fois. Votre programme ne peut plus appeler `add_element()` après cet appel. Votre programme ne peut pas appeler `check_element()` avant cet appel.
- `boolean check_element(string x)`
  Cette fonction vérifie si l'élément `x` est dans l'ensemble modifié.
  - `x` : une chaîne de caractères de `'0'` et `'1'` donnant la représentation

binaire de l'entier dont la présence dans l'ensemble doit être vérifiée. La longueur de `x` doit être n.
- retourne `true` si l'élément `x` est dans l'ensemble modifié, `false` sinon.

Notez que si votre programme viole l'une de ces restrictions, le résultat de l'évaluateur sera "Wrong Answer" (mauvaise réponse).

Pour toutes les chaînes de caractères, le premier caractère donne le bit de poids fort de l'entier.

L'évaluateur fixe la permutation p avant que la fonction `restore_permutation` ne soit appelée.

Veuillez utiliser les fichiers modèles pour obtenir l'implémentation exacte dans votre langage de programmation.

Exemple

L'évaluateur fait l'appel de fonction suivant :

- `restore_permutation(4, 16, 16)`. Nous avons $n = 4$ et le programme peut faire au plus 16 "writes" et 16 "reads".

Le programme fait les appels de fonctions suivants :

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

Seule une permutation est compatible avec ces valeurs retournées par `check_element()` : la permutation $p = [2, 1, 3, 0]$. Donc, `restore_permutation` doit retourner `[2, 1, 3, 0]`.

Sous-tâches

1. (20 points) $n = 8$, $w = 256$, $r = 256$, $p_i \neq i$ pour au plus 2 indices i ($0 \leq i \leq n - 1$),
2. (18 points) $n = 32$, $w = 320$, $r = 1024$,
3. (11 points) $n = 32$, $w = 1024$, $r = 320$,
4. (21 points) $n = 128$, $w = 1792$, $r = 1792$,
5. (30 points) $n = 128$, $w = 896$, $r = 896$.

Evaluateur fourni (grader)

L'évaluateur fourni lit l'entrée dans le format suivant :

- ligne 1 : entiers n , w , r ,
- ligne 2 : n entiers spéficiant les éléments de p .