Desenredando un Bug Enmarañado

Ilshat es un ingeniero de software que trabaja escribiendo estructuras de datos eficientes. Un día, inventó una nueva estructura de datos. Esta estructura de datos puede guardar un conjunto de enteros no negativos de n bits, donde n es una potencia de dos. Es decir, $n = 2^b$ para algún entero no negativo b.

La estructura de datos comienza vacía. Las reglas para que un programa use la estructura de datos son:

- El programa puede agregar enteros de n bits a la estructura de datos, uno a la vez, usando la función `add_element(x)` (agregar elemento). Si el programa intenta agregar un elemento que ya está presente en la estructura de datos, no pasa nada.
- Después de agregar el último elemento, el programa debe llamar la función `compile_set()` (compilar conjunto) exactamente una vez.
- Por último, el programa puede llamar la función `check_element(x)` (checar elemento) para checar si el elemento x está presente en la estructura de datos. Esta función se puede usar varias veces.

Cuando Ilshat implementó esta estructura de datos, cometió un error y metió un bug a la función `compile_set()`. El bug reordena los dígitos en binario de cada elemento siempre de la misma manera. Ilshat quiere que le ayudes a encontrar exactamente cómo es que el bug reordena los dígitos.

Formalmente, considera una secuencia $p = [p_0, \ldots, p_{n-1}]$ en la que cada número del 0 al $n-1$ aparece exactamente una vez. A las secuencias de este tipo les llamamos *permutaciones*. Ahora, considera un elemento en el conjunto de la estructura de datos, con dígitos en binario $a_0, \ldots, a_{n-1}$ (siendo a_0 el bit más significativo). Cuando se llama la función `compile_set()`, este elemento es reemplazado por el elemento $a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}}$.

El bug usa la misma permutación p para reordenar los dígitos de todos los elementos. La permutación puede ser cualquiera, incluyendo la posibilidad de que $p_i = i$ para todo $0 \leq i \leq n-1$.

Por ejemplo, supón que $n = 4$, $p = [2, 1, 3, 0]$, y que insertaste al conjunto los enteros con representación binaria `0000`, `1100` y `0111`. Al llamar la función `compile_set`, los elementos cambian a `0000`, `0101` y `1110`, respectivamente.

Tu tarea es escribir un programa que encuentre la permutación p, interactuando con la estructura de datos. Debería hacer lo siguiente:

1. elegir un conjunto de enteros no negativos de n bits,

2. insertar esos enteros en la estructura de datos,
3. llamar a la función `compile_set` para que ocurra el bug,
4. checar si están o no algunos elementos en la estructura de datos,
5. usar esa información para determinar y regresar la permutación p.

Nota que tu programa sólo puede llamar a la función `compile_set` una vez.

Adicionalmente, hay un límite en la cantidad de veces que tu programa puede llamar a las funciones de la biblioteca. En particular, puede:

- llamar a `add_element` a lo más w veces (w viene del inglés "writes", de escrituras),
- llamar a `check_element` a lo más r veces (r viene del inglés "reads", de lecturas).

Detalles de la implementación

Debes implementar la función (método):

- `int[] restore_permutation(int n, int w, int r)`
  - `n`: el número de bits en la representación binaria de cada elemento del conjunto (que también es la longitud de p).
  - `w`: el número máximo de llamadas a `add_element` que puedes hacer.
  - `r`: el número máximo de llamadas a `check_element` que puedes hacer.
  - la función debe regresar la permutación restaurada p.

En el lenguaje C, la declaración de la función es ligeramente diferente:

- `void restore_permutation(int n, int w, int r, int* result)`
  - `n`, `w` y `r` son iguales que antes.
  - la función debe regresar la permutación restaurada p guardándola en el arreglo `result`: para cada i, debe guardar el valor p_i en `result[i]`.

Funciones de la biblioteca

Para interactuar con la estructura de datos, tu programa debe usar las siguientes tres funciones (métodos):

- `void add_element(string x)`
  Esta función añade al elemento descrito por `x` al conjunto.
  - `x`: una cadena de caracteres `'0'` y `'1'` con la representación en binario del entero que se agregará al conjunto. La longitud de `x` debe ser n.

- `void compile_set()`
  Esta función debe llamarse exactamente una vez. Tu programa no puede llamar a `add_element()` después de esta llamada. Tu programa no puede llamar a `check_element()` antes de esta llamada.
- `boolean check_element(string x)`
  Esta función checa si `x` está en el conjunto modificado o no.
  - `x`: una cadena de caracteres `'0'` y `'1'` con la representación en binario del entero a checar. La longitud de `x` debe ser n.
  - regresa `true` si `x` está en el conjunto modificado, y `false` de lo contrario.

Nota que si tu programa viola cualquiera de las restricciones anteriores, será

jueceado con "Wrong Answer" (Respuesta Incorrecta).

Para todas las cadenas de caracteres, el primer caracter representa el bit más significativo del entero correspondiente.

El evaluador fija la permutación p antes de llamar a `restore_permutation`.

Por favor utiliza la plantilla proveída para más detalles de la implementación en tu lenguaje de programación.

Ejemplo

El evaluador hace la siguiente llamada a tu función:

- `restore_permutation(4, 16, 16)`. Tenemos $n = 4$ y el programa puede hacer a lo más 16 "escrituras" y 16 "lecturas".

El programa hace las siguientes llamadas:

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

Solo una permutación es consistente con los valores regresados por `check_element()`: la permutación $p = [2, 1, 3, 0]$. Por lo tanto, `restore_permutation` debe regresar `[2, 1, 3, 0]`.

Subtareas

1. (20 puntos) $n = 8$, $w = 256$, $r = 256$, y sólo dos bits serán intercambiados, es decir, $p_i \neq i$ para a lo más 2 índices i ($0 \leq i \leq n - 1$),
2. (18 puntos) $n = 32$, $w = 320$, $r = 1024$,
3. (11 puntos) $n = 32$, $w = 1024$, $r = 320$,
4. (21 puntos) $n = 128$, $w = 1792$, $r = 1792$,
5. (30 puntos) $n = 128$, $w = 896$, $r = 896$.

Evaluador de ejemplo

El evaluador de ejemplo lee la entrada en el siguiente formato:

- línea 1: los enteros n, w, r,
- línea 2: n enteros describiendo los elementos de p.