Unscrambling a Messy Bug

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

La estructura de datos está inicialmente vacía. Un programa que usa la estructura de datos tiene que seguir las siguientes reglas:

- El programa puede añadir elementos que son enteros de n bits a la estructura de datos, uno a la vez, usando la función `add_element(x)`. Si el programa trata de añadir un elemento que ya está presente en la estructura de datos, no pasa nada.
- Después de añadir el último elemento el programa debe llamar a la función `compile_set()` exactamente una vez.
- Finalmente, el programa puede llamar a la función `check_element(x)` para verificar si el elemento x está presente en la estructura de datos. Esta función puede ser usada varias veces.

Cuando Ilshat implementó por primera vez esta estructura de datos, cometió un error en la función `compile_set()`. El error reordena los dígitos binarios de cada elemento en el conjunto de la misma manera. Ilshat quiere que encuentres el reordenamiento exacto de dígitos causado por el error.

Formalmente, considere una secuencia $p = [p_0, \ldots, p_{n-1}]$ en la cual cada número desde 0 a $n-1$ aparece exactamente una vez. Llamamos a tal secuencia una *permutación*. Considere un elemento del conjunto, cuyos dígitos en binario son $a_0, \ldots, a_{n-1}$ (siendo a_0 el bit más significativo). Cuando se llama a la función `compile_set()`, este elemento es reemplazado por el elemento $a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}}$.

La misma permutación p es usada para reordenar los dígitos de cada elemento almacenado en la estructura de datos. Cualquier permutación es posible, incluyendo la posibilidad de que $p_i = i$ para cada $0 \leq i \leq n-1$.

Por ejemplo, supon que $n = 4$, $p = [2, 1, 3, 0]$, y que has insertado en el conjunto enteros cuya representación binaria es `0000`, `1100` y `0111`. Llamar a la función `compile_set` cambia esos elementos a `0000`, `0101` y `1110`, respectivamente.

Tu tarea es escribir un programa que encuentre la permutación p interactuando con la estructura de datos. Este debería (en el siguiente orden):

1. elegir un conjunto de enteros de n bits,
2. insertar esos enteros en la estructura de datos,

3. llamar a la función `compile_set` para gatillar el error,
4. verificar la presencia de algunos elementos en el conjunto modificado,
5. usar esa información para determinar y retornar la permutación p.

Notar que el programa puede llamar a la función `compile_set` únicamente una vez.

Adicionalmente, hay un límite al número de veces que el programa puede llamar a las funciones de librería. A saber, puede:

- llamar `add_element` a lo más w veces (w es de "writes"),
- llamar `check_element` a lo más r veces (r es de "reads").

Detalles de implementación

Debes implementar una función (método):

- `int[] restore_permutation(int n, int w, int r)`
  - `n`: el número de bits en la representación binaria de cada elemento en el conjunto (y también la longitud de p).
  - `w`: el número máximo de operaciones `add_element` que el programa puede ejecutar.
  - `r`: el número máximo de operaciones `check_element` que el programa puede ejecutar.
  - la función debe retornar la permutación p restaurada.

En el lenguaje C, la firma de la función es un poco diferente:

- `void restore_permutation(int n, int w, int r, int* result)`
  - `n`, `w` y `r` tienen el mismo significado que antes.
  - la función debe retornar la permutación p restaurada almacenandola en el arreglo proporcionado `result`: para cada i, debe almacenar el valor p_i en `result[i]`.

Funciones de librería

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

- `void add_element(string x)`
  Esta función añade el elemento descrito por `x` al conjunto.
  - `x`: es un string de `'0'` y `'1'` dando la representación binaria de un entero que debe ser añadido al conjunto. La longitud de `x` debe ser n.

- `void compile_set()`
  Esta función debe ser llamada 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 verifica si el elemento `x` está en el conjunto modificado.
  - `x`: un string de `'0'` y `'1'` dando la representación binaria del elemento que debe ser verificado. La longitud de `x` debe ser n.
  - retorna `true` si el elemento `x` está en el conjunto modificado, y `false` en otro caso.

Notar que si el programa viola cualquiera de las restricciones antes mencionadas, tu calificación será "Wrong Answer".

Para todos los strings, el primer caracter da el bit más significativo del entero correspondiente.

El grader fija la permutación p antes de que la función `restore_permutation` sea llamada.

Por favor usa los archivos con las plantillas para ver detalles de implementación en tu lenguaje de programación.

Ejemplo

El grader hace el siguiente llamado de función:

- `restore_permutation(4, 16, 16)`. Tenemos $n = 4$ y el programa puede hacer a lo mas 16 "writes" y 16 "reads".

El programa realiza los siguientes llamados de función:

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

Solamente una permutación es consistente con esos valores retornados por `check_element()`: la permutación $p = [2, 1, 3, 0]$. Por lo tanto, `restore_permutation` debe retornar `[2, 1, 3, 0]`.

Subtareas

1. (20 puntos) $n = 8$, $w = 256$, $r = 256$, $p_i \neq i$ para a lo más 2 indices 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) $n = 128$, $w = 896$, $r = 896$.

Grader de ejemplo

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

- línea 1: enteros n, w, r,

- línea 2: n enteros dando los elementos de p.