Soluciones entrenos OIE
Unscrambling a Messy Bug

Ilshat es un ingeniero de software que trabaja en estructuras de datos eficientes. Un día inventó una nueva estructura de datos. Esta estructura puede almacenar 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 sean enteros de n bits a la estructura de datos, de a uno por 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 un 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 del conjunto de la misma manera. Ilshat quiere que usted encuentre 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 $a_0, \ldots, a_{n-1}$ (con a_0 siendo el dígito 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. La permutación puede ser arbitraria, incluyendo la posibilidad que $p_i = i$ para cada $0 \leq i \leq n-1$.

Por ejemplo, suponga que $n = 4, p = [2, 1, 3, 0]$, y que se han 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.

Su tarea es escribir un programa que encuentre la permutación p , interactuando para ello con la estructura de datos. El programa 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 disparar el error,
4. verificar la presencia de algunos elementos en el conjunto modificado,
5. usar esa información para determinar y retornar la permutación p

Note que su programa puede llamar la función `compile_set` únicamente una vez.

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

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

Subtareas

- 20 Puntos: $n = 8, w = 256, r = 256$, $p_i \neq i$ como mucho para dos índices i ($0 \leq i \leq n-1$).
- 18 Puntos: $n = 32, w = 320, r = 1024$.
- 11 Puntos: $n = 32, w = 1024, r = 320$.
- 21 Puntos: $n = 128, w = 1792, r = 1792$.
- 30 Puntos: $n = 128, w = 896, r = 896$.

Solución

Una de las primeras cosas que podemos pensar al intentar este problema es cómo podemos determinar una entrada concreta de la permutación. Si insertamos en el conjunto la string $10000\ldots$ entonces podemos determinar a qué posición se envía el primer bit leyendo todas las posibles strings con un bit a 1. Pero si insertamos $01000\ldots$ también, por ejemplo, entonces ya no podemos determinar a qué posición va el primer bit ni a qué posición va el segundo, ya que aunque encontraremos las dos strings con un bit resultantes no podremos distinguir con qué string inicial corresponde cada una. Para ir determinando a qué posición va cada bit tendríamos que insertar las strings $11000\ldots, 11100\ldots, 11110\ldots, \ldots$: en este caso, una vez conocemos la posición primer bit podemos determinar la del segundo (buscando entre todas las strings con el bit correspondiente al primer bit a 1 y otro bit distinto a 1), y después la del tercero, y así vamos determinando todos los bits. Pero en esta estrategia, aunque sólo tengamos que hacer n operaciones de escritura, en el peor caso hacemos $n + (n-1) + \ldots + 1 = \frac{n(n-1)}{2}$ operaciones de lectura, así que no nos sirve para obtener todos los puntos.

Yendo directamente a determinar las posiciones de la permutación no conseguimos la solución más eficiente: ¿hay otra forma de obtener información útil? Hemos dicho que insertando varias strings con un sólo bit a 1 no podemos determinar las posiciones de los bits que hemos puesto porque no podemos distinguir cuál era el string original de cada string resultante, pero sí que obtenemos información: sabemos a qué conjunto de posiciones va a parar el conjunto de bits que hemos puesto. Por ejemplo, si insertamos $10000\ldots$ y $01000\ldots$ en el conjunto y ninguna string más con un solo bit, y después leemos todas las posibles strings de un sólo bit y encontramos dos strings en el conjunto, sabremos cuáles son las posiciones a las que van los dos primeros bits (aunque no sepamos cuál va a cuál) y también sabremos cuáles son las posiciones a las que van los otros $n-2$ bits, que son el resto de posiciones. Esto es útil porque una vez tenemos determinada esta información de a qué posiciones van los bits de un conjunto determinado y los bits del conjunto complementario, lo que nos queda es en cierto modo el mismo problema (determinar una permutación) pero en cada conjunto por separado. Esto da una pista de que podemos ir resolviendo el problema recursivamente.

Lo que haremos será ir dividiendo por la mitad el conjunto de bits e ir determinando para cada mitad a qué conjunto van a parar los bits de esa mitad, hasta que nos queden conjuntos del tamaño de un bit. Para concretar, haremos un ejemplo con $n = 8$. Primero queremos saber a qué conjunto de bits va a parar cada mitad de los bits, así que insertaremos las strings $10000000, 01000000, 00100000, 00010000$. Cuando leamos encontraremos cuatro strings de un bit que corresponderán a las posiciones de los primeros cuatro

bits, y las otras cuatro posiciones serán las de los otros cuatro. Ahora dividimos la primera mitad por la mitad, y queremos encontrar a qué conjunto van los dos primeros bits. No sirve volver a insertar 10000000 y 01000000 porque ya los habíamos insertado antes y no podemos distinguirlos de los otros dos bits. Lo que haremos será insertar 10001111 y 01001111: como ya sabemos cuál es el conjunto al que va a parar los últimos cuatro bits cuando leamos podemos buscar estas strings poniendo a 1 los bits de ese conjunto y preguntando por las strings que tengan otro bit a 1. Análogamente para la otra mitad insertamos 11111000 y 11110100. Por último nos quedan conjuntos de dos bits, en los que volvemos a hacer lo mismo: en el caso del primer conjunto, insertamos 10111111 y como sabemos el conjunto al que va a parar los últimos 6 bits podremos determinar a qué bit va a parar cada uno de los dos bits. En general, cuando estemos dividiendo por la mitad un intervalo, lo que haremos será insertar una string por cada bit de la primera mitad con los bits que queden fuera del intervalo a 1.

¿Cuántas operaciones hacemos con esta estrategia? Sea $n = 2^b$. Para escribir, escribimos $\frac{n}{2} = 2^{b-1}$ strings y luego repetimos recursivamente lo mismo en cada una de las mitades: por tanto, si w_n es el número de operaciones de escritura para un conjunto de tamaño n entonces se satisface la recurrencia $w_{2^b} = 2^{b-1} + 2 \cdot w_{2^{b-1}}$ y desarrollando la recurrencia obtenemos $w_{2^b} = 2^{b-1} \cdot b = \frac{n}{2} \log n$. Para leer, tenemos primero que leer todas las strings de un sólo bit (aunque sólo hayamos escrito la mitad, no sabemos a qué bits van a parar y tenemos que comprobar todas) y después repetimos lo mismo recursivamente en cada mitad, así que $r_{2^b} = 2^b + 2 \cdot r_{2^{b-1}} \implies r_{2^b} = 2^b \cdot b = n \log n$. Por tanto, vemos que esta estrategia nos permite obtener todos los puntos.

Código

C++

```cpp
#include <vector>
#include <iostream>
#include "messy.h"

using namespace std;

//array donde guardaremos la permutacion
int p[128];
//tenemos N como variable global por comodidad
int N;

//Funcion donde desarrollamos la primera etapa (escritura)
//Funciona recursivamente: write(a, b, s) escribe los strings para
//determinar los conjuntos de las dos mitades del intervalo [a, b]
//y s es una string con todos los bits de fuera del intervalo a 1
//(se podria hacer sin este parametro, solamente esta para hacerlo
//mas analogo a la funcion read, donde s si tiene mas sentido ponerlo)
void write(int a, int b, string s) {
        if(a == b) return;

        //anadimos las strings correspondientes a los bits de la primera mitad
        int c = (a+b)/2;
        for(int i=a; i <= c; ++i) {
                s[i] = '1';
                add_element(s);
                s[i] = '0';
        }

    //pasamos los nuevos strings y hacemos las llamadas recursivas
        string ns = string(N, '1');
```

```
        for(int i=a; i <= c; ++i) {
                ns[i] = '0';
        }
        write(a, c, ns);

        ns = string(N, '1');

        for(int i=c+1; i <= b; ++i) {
                ns[i] = '0';
        }
        write(c+1, b, ns);

}

//Funcion donde desarrollamos la segunda etapa (lectura)
//Funciona recursivamente: read(a, b, s) lee los strings para
//determinar los conjuntos de las dos mitades del intervalo [a, b]
//y s es una string con todos los bits de las posiciones finales
//correspondientes a bits de fuera del intervalo a 1
void read(int a, int b, string s) {
    //si a=b, podemos determinar la entrada de la permutacion
    //a partir de la s que nos ha llegado de llamadas anteriores
        if(a == b) {
                for(int i=0; i < N; ++i) {
                        if(s[i] == '0') p[i] = a;
                }
        }
        else {
            //determinamos los bits de la primera mitad del intervalo
            //estos bits los ponemos en ns para hacer la siguiente llamada
                string ns = string(N, '1');
                for(int i=0; i < N; ++i) {
                        if(s[i] == '0') {
                                s[i] = '1';
                                if(check_element(s)) ns[i] = '0';
                                s[i] = '0';
                        }
                }

                int c = (a+b)/2;
                read(a, c, ns);

        //ahora modificamos ns para que corresponda a la segunda mitad
                for(int i=0; i < N; ++i) {
                        if(ns[i] == '0') ns[i] = '1';
                        else {
                                if(s[i] == '0') ns[i] = '0';
                        }
                }

                read(c+1, b, ns);
        }
}

int* restore_permutation(int n, int w, int r) {
```

```
86            N = n;
87
88        write(0, N-1, string(N, '0'));
89        compile_set();
90        read(0, N-1, string(N, '0'));
91
92        return p;
93    }
```