Registros de desplazamiento de bits

Christopher el ingeniero está trabajando en un nuevo tipo de procesador de computadoras.

El procesador tiene acceso a m diferentes celdas de memoria de b bits (donde $m = 100$ y $b = 2000$), que son llamados **registros**, y son numeradas de 0 a $m - 1$. Denotamos los registros por $r[0], r[1], \ldots, r[m-1]$. Cada registro es un arreglo de b bits, numerados del 0 (el bit de más a la derecha) a $b - 1$ (el bit de más a la izquierda). Para cada i $(0 \leq i \leq m - 1)$ y cada j $(0 \leq j \leq b - 1)$, denotamos el j-ésimo bit del registro i por $r[i][j]$.

Para cualquier secuencia de bits $d_0, d_1, \ldots, d_{l-1}$ (de largo arbitrario l), el **valor entero** de la secuencia es igual a $2^0 \cdot d_0 + 2^1 \cdot d_1 + \ldots + 2^{l-1} \cdot d_{l-1}$. Decimos que el **valor entero guardado en un registro** i es el valor entero de la secuencia de sus bits: es decir, es $2^0 \cdot r[i][0] + 2^1 \cdot r[i][1] + \ldots + 2^{b-1} \cdot r[i][b-1]$.

El procesador tiene 9 tipos de **instrucciones** que pueden ser usadas para modificar los bits del registro. Cada instrucción opera sobre uno o más registros y guarda la salida en uno de sus registros. A continuación, utilizaremos $x := y$ para denotar una operación cambiando el valor de x tal que se convierte igual a y. Las operaciones llevadas a cabo por cada tipo de instrucción son descritas a continuación.

- $move(t, y)$: Copiar el arreglo de bits en el registro y al registro t. Para cada j $(0 \leq j \leq b - 1)$, asignar $r[t][j] := r[y][j]$.
- $store(t, v)$: Asignar al registro t el valor de v, donde v es un arreglo de b bits. Para cada, j $(0 \leq j \leq b - 1)$, asignar $r[t][j] := v[j]$.
- $and(t, x, y)$: Tomar el AND de bit a bit de los registros x e y, y guardar el resultado en el registro t. Para cada j $(0 \leq j \leq b - 1)$, asignar $r[t][j] := 1$ si $r[x][j]$ y $r[y][j]$ son **ambos** 1, y asignar $r[t][j] := 0$ de lo contrario.
- $or(t, x, y)$: Tomar el OR de bit a bit de los registros x e y, y guardar el resultado en el registro t. Para cada j $(0 \leq j \leq b - 1)$, asignar $r[t][j] := 1$ si **al menos uno** de $r[x][j]$ o $r[y][j]$ es 1, y asignar $r[t][j] := 0$ de lo contrario.
- $xor(t, x, y)$: Tomar el XOR de bit a bit de los registros x e y, y guardar el resultado en el registro t. Para cada j $(0 \leq j \leq b - 1)$, asignar $r[t][j] := 1$ si **exactamente uno** de $r[x][j]$ y $r[y][j]$ es 1, y asignar $r[t][j] := 0$ de lo contrario.
- $not(t, x)$: Tomar el NOT (negación) de bit a bit de los registros x e y, y guardar el resultado en el registro t. Para cada j $(0 \leq j \leq b - 1)$, asignar $r[t][j] := 1 - r[x][j]$.
- $left(t, x, p)$: Desplazar los bits en el registro x en p posiciones hacia la izquierda, y guardar el resultado en el registro t. El resultado de desplazar los bits en el registro x hacia la izquierda

por p bits es un arreglo v de b bits. Para cada j $(0 \leq j \leq b-1)$, $v[j] = r[x][j-p]$ Si $j \geq p$, y $v[j] = 0$ de lo contrario. Para cada j $(0 \leq j \leq b-1)$, asignar $r[t][j] := v[j]$.

- $right(t, x, p)$: Desplazar los bits en el registro x en p posiciones hacia la derecha, y guardar el resultado en el registro t. El resultado de desplazar los bits en el registro x hacia la derecha por p bits es un arreglo v de b bits. Para cada j $(0 \leq j \leq b-1)$, $v[j] = r[x][j+p]$ Si $j \leq b-1-p$, y $v[j] = 0$ de lo contrario. Para cada j $(0 \leq j \leq b-1)$, asignar $r[t][j] := v[j]$.

- $add(t, x, y)$: Sumar el valor entero guardado en los registros x e y, y guardar el resultado en el registro t. La suma se lleva a cabo módulo 2^b. Formalmente, sea X el valor entero guardado en el registro x, e Y el valor entero guardado en el registro y antes de la operación. Sea T el valor entero guardado en el registro t después de la operación. Si $X + Y < 2^b$, asignar a los bits de t, de tal forma que $T = X + Y$. De lo contrario, asignar los bits de t, tal que $T = X + Y - 2^b$.

Christopher quiere que resuelvas dos tipos de tareas usando el nuevo procesador. El tipo de tarea se denota por un entero s. Para ambos tipos, necesitas producir un **programa** que sea una secuencia de las instrucciones definidas anteriormente.

La **entrada** al programa consiste de n enteros $a[0], a[1], \ldots, a[n-1]$, cada uno con k bits; es decir, $a[i] < 2^k$ ($0 \leq i \leq n-1$). Antes de ejecutar el programa, todos los números de entrada son guardados secuencialmente en el registro 0, de tal manera que para cada i $(0 \leq i \leq n-1)$ el valor entero de la secuencia de k bits $r[0][i \cdot k], r[0][i \cdot k + 1], \ldots, r[0][(i+1) \cdot k - 1]$ es igual a $a[i]$. Nota que $n \cdot k \leq b$. Todo el resto de bits en el registro 0 (es decir, aquellos con índice entre $n \cdot k$ y $b-1$, inclusive) y todos los bits en todos los demás registros son inicializados en 0.

Ejecutar un programa consiste en ejecutar sus instrucciones en orden. Después que la última instrucción es ejecutada, la **salida** del programa se calcula en base al valor final de los bits en el registro 0. Específicamente, la salida es una secuencia de n enteros $c[0], c[1], \ldots, c[n-1]$, donde por cada i ($0 \leq i \leq n-1$), $c[i]$ es el valor entero de una secuencia de los bits $i \cdot k$ a $(i+1) \cdot k - 1$ del registro 0. Nota que después de ejecutar el programa, los bits restantes del registro 0 (con índices al menos $n \cdot k$) y todos los bits de todos los demás registros pueden ser arbitrarios.

- La primera tarea $(s = 0)$ es encontrar el menor entero entre los enteros de entrada $a[0], a[1], \ldots, a[n-1]$. Específicamente, $c[0]$ debe ser el mínimo de $a[0], a[1], \ldots, a[n-1]$. Los valores de $c[1], c[2], \ldots, c[n-1]$ pueden ser arbitrarios.

- La segunda tarea $(s = 1)$ es ordenar los valores de entrada $a[0], a[1], \ldots, a[n-1]$ en orden no decreciente. Específicamente, para cada i ($0 \leq i \leq n-1$), $c[i]$ debe ser igual al $(1+i)$-ésimo menor entero entre $a[0], a[1], \ldots, a[n-1]$ (es decir, $c[0]$ es el menor entero entre los enteros de entrada).

Proporciona a Christopher programas formados por a lo sumo q instrucciones cada uno, que puedan resolver dichas tareas.

Detalles de Implementación

Debes implementar la siguiente función:

```
void construct_instructions(int s, int n, int k, int q)
```

- s: tipo de la tarea.
- n: número de enteros en la entrada.
- k: número de bits en cada entero de entrada.
- q: máximo número de instrucciones permitidas.
- Esta función es llamada exactamente una vez y debe construir una secuencia de instrucciones para realizar la tarea requerida.

Esta función debe llamar una o más de las siguientes funciones para construir una secuencia de instrucciones:

```
void append_move(int t, int y)
void append_store(int t, bool[] v)
void append_and(int t, int x, int y)
void append_or(int t, int x, int y)
void append_xor(int t, int x, int y)
void append_not(int t, int x)
void append_left(int t, int x, int p)
void append_right(int t, int x, int p)
void append_add(int t, int x, int y)
```

- Cada función agrega una instrucción $move(t, y)$, $store(t, v)$, $and(t, x, y)$, $or(t, x, y)$, $xor(t, x, y)$, $not(t, x)$, $left(t, x, p)$, $right(t, x, p)$ o $add(t, x, y)$ al programa, respectivamente.
- Para todas las instrucciones relevantes, t, x, e y deben ser al menos 0 y a lo sumo $m - 1$.
- Para todas las instrucciones relevantes, t, x, e y no son necesariamente distintas.
- Para las instrucciones $left$ y $right$, p debe ser al menos 0 y a lo sumo b.
- Para las instrucciones $store$, el tamaño de v debe ser b.

Puedes también llamar la siguiente función para ayudarte a probar tu solución:

```
void append_print(int t)
```

- Cualquier llamada a esta función será ignorada durante la calificación de tu solución.
- En el calificador de ejemplo, este procedimiento agregará una operación $print(t)$ al programa.
- Cuando el calificador de ejemplo encuentre una operación $print(t)$ durante la ejecución de tu programa, imprimirá n enteros de k bits formados por los primeros $n \cdot k$ bits del registro t (ver la sección de "Calificador de Ejemplo" para más detalles).
- t debe satisfacer $0 \leq t \leq m - 1$.
- Cualquier llamada a esta función no contribuye al número de instrucciones construidas.

Después de agregar la última instrucción, `construct_instructions` debe retornar. Luego el programa es evaluado sobre un número de casos de prueba, cada uno especificando una entrada

que consiste de n enteros de k bits $a[0], a[1], \ldots, a[n-1]$. Tu solución pasa los casos de prueba si la salida del programa $c[0], c[1], \ldots, c[n-1]$ para cada entrada satisface las siguientes condiciones:

- Si $s = 0$, $c[0]$ debe ser el menor de los valores de $a[0], a[1], \ldots, a[n-1]$.
- Si $s = 1$, para cada i ($0 \leq i \leq n-1$), $c[i]$ debe ser el $1+i$-ésimo menor entero entre $a[0], a[1], \ldots, a[n-1]$.

La calificación de tu solución puede resultar en uno de los siguientes mensajes de error:

- `Invalid index`: índice de registro incorrecto (posiblemente negativo) fue dado como parámetro t, x o y para alguna llamada de una de las funciones.
- `Value to store is not b bits long`: el tamaño de v dado a `append_store` no es igual a b.
- `Invalid shift value`: el valor de p dado a `append_left` o `append_right` no está entre 0 y b, inclusive.
- `Too many instructions`: tu procedimiento intentó agregar más de q instrucciones.

Ejemplos

Ejemplo 1

Supongamos que $s = 0$, $n = 2$, $k = 1$, $q = 1000$. Hay dos enteros de entrada $a[0]$ y $a[1]$, cada uno con $k = 1$ bit. Antes de la ejecución del programa, $r[0][0] = a[0]$ y $r[0][1] = a[1]$. Todos los demás bits en el procesador son 0. Después que todas las instrucciones del programa son ejecutadas, debemos tener $c[0] = r[0][0] = \min(a[0], a[1])$, que es el mínimo de $a[0]$ y $a[1]$.

Sólo hay 4 posibles entradas para el programa:

- Caso 1: $a[0] = 0, a[1] = 0$
- Caso 2: $a[0] = 0, a[1] = 1$
- Caso 3: $a[0] = 1, a[1] = 0$
- Caso 4: $a[0] = 1, a[1] = 1$

Notamos que para todos los casos, $\min(a[0], a[1])$ es igual al AND de bit a bit de $a[0]$ y $a[1]$. Por lo tanto, una solución posible es construir un programa haciendo las siguientes llamadas:

1. `append_move(1, 0)`, que agrega una instrucción para copiar $r[0]$ a $r[1]$.
2. `append_right(1, 1, 1)`, que agrega una instrucción que toma todos los bits en $r[1]$, los desplaza hacia la derecha en 1 bit, y luego guarda el resultado en $r[1]$. Ya que cada entero es de largo 1 bit, esto hace que $r[1][0]$ sea igual a $a[1]$.
3. `append_and(0, 0, 1)`, que agrega una instrucción para realizar el AND de bit a bit de $r[0]$ y $r[1]$, luego guarda el resultado en $r[0]$. Después que estas instrucciones se ejecutan, $r[0][0]$ es asignado el AND de bit a bit de $r[0][0]$ y $r[1][0]$, que es igual al AND de bit a bit de $a[0]$ y $a[1]$, como se desea.

Ejemplo 2

Supongamos que $s = 1,\ n = 2,\ k = 1,\ q = 1000$. Como en el ejemplo anterior, únicamente hay 4 posibles entradas para el programa. Para los cuatro casos, $\min(a[0], a[1])$ es el AND de bit a bit de $a[0]$ y $a[1]$, y $\max(a[0], a[1])$ es el OR de bit a bit de $a[0]$ y $a[1]$. Una posible solución es realizar las siguientes llamadas:

1. `append_move(1,0)`
2. `append_right(1,1,1)`
3. `append_and(2,0,1)`
4. `append_or(3,0,1)`
5. `append_left(3,3,1)`
6. `append_or(0,2,3)`

Luego de ejecutar estas instrucciones, $c[0] = r[0][0]$ contiene $\min(a[0], a[1])$, y $c[1] = r[0][1]$ contiene $\max(a[0], a[1])$, que es la entrada ordenada.

Restricciones

- $m = 100$
- $b = 2000$
- $0 \leq s \leq 1$
- $2 \leq n \leq 100$
- $1 \leq k \leq 10$
- $q \leq 4000$
- $0 \leq a[i] \leq 2^k - 1$ (para todo $0 \leq i \leq n - 1$)

Subtareas

1. (10 puntos) $s = 0, n = 2, k \leq 2, q = 1000$
2. (11 puntos) $s = 0, n = 2, k \leq 2, q = 20$
3. (12 puntos) $s = 0, q = 4000$
4. (25 puntos) $s = 0, q = 150$
5. (13 puntos) $s = 1, n \leq 10, q = 4000$
6. (29 puntos) $s = 1, q = 4000$

Calificador de ejemplo

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

- línea $1:\ s\ n\ k\ q$

Esto seguido por algún número de lineas, cada una describiendo un solo caso de prueba. Cada caso de prueba es proporcionado en el siguiente formato:

- $a[0]\ a[1]\ \ldots\ a[n-1]$

y describe un caso de prueba el cual su entrada consiste en n enteros $a[0], a[1], \ldots, a[n-1]$. La descripción de todos los casos de prueba es seguido por una sola linea que contiene unicamente

-1.

El calificador de ejemplo primero llama `construct_instructions(s, n, k, q)`. Si esta llamada viola alguna restricción descrita en la definición del problema, el calificador de ejemplo imprime uno de los mensajes de error listados al final de la sección "Detalles de implementación" y finaliza. De lo contrario, el calificador de ejemplo imprime cada instrucción agregada en orden por `construct_instructions(s, n, k, q)`. Para instrucciones `store`, v es impreso desde el indice 0 hasta el indice $b-1$.

Luego, el calificador de ejemplo procesa los casos de prueba en orden. Para cada caso de prueba, ejecuta el programa construido en la entrasa del caso de prueba.

Para cada operación `print(t)`, sea $d[0], d[1], \ldots, d[n-1]$ una secuencia de enteros, tal que para cada i ($0 \leq i \leq n-1$), $d[i]$ es el valor entero de la secuencia de bits $i \cdot k$ a $(i+1) \cdot k - 1$ del registro t (cuando la operación es ejecutada).

El calificador imprime la secuencia en el siguiente formato: `register` t: $d[0]\ d[1]\ \ldots\ d[n-1]$.

Una vez todas las instrucciones han sido ejecutadas, el calificador de ejemplo imprime la salida del programa.

Si $s = 0$, la salida del calificador de ejemplo para cada caso de prueba es en el siguiente formato:

- $c[0]$.

Si $s = 1$, la salida del calificador de ejemplo para cada caso de prueba es en el siguiente formato:

- $c[0]\ c[1]\ \ldots\ c[n-1]$.

Después de ejecutar todos los casos de prueba, el calificador de ejemplo imprime `number of instructions:` X donde X es el número de instrucciones en tu programa.