Registros de Desplazamiento de Bits

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

El procesador tiene acceso a m diferentes casillas de memoria de b bits (donde $m = 100$ y $b = 2000$), que son llamadas **registros**, y están numeradas de 0 a $m - 1$. Los registros se denotan como $r[0], r[1], \ldots, r[m-1]$. Cada registro es n arreglo de b bits, numerados de 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 como $r[i][j]$.

Para cualquier secuencia de bits $d_0, d_1, \ldots, d_{l-1}$ (de tamaño 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 en los registros. Cada instrucción opera sobre uno o más registros y guarda el resultado en uno de los registros. De aquí en adelante, usaremos $x := y$ para denotar una operación que cambia el valor de x de tal forma que se vuelva y. Las operaciones realizadas por cada instrucción se describen en las siguientes líneas.

- $move(t, y)$: 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 un valor igual a v, donde v es un arreglo de b bits. Para cada j $(0 \leq j \leq b - 1)$, asignar $r[t][j] := v[j]$.
- $and(t, x, y)$: Calcula el valor AND bit a bit de los registros x e y, y guarda 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** iguales a 1, asignar $r[t][j] := 0$ caso contrario.
- $or(t, x, y)$: Calcula el valor OR bit a bit de los registros x e y, y guarda el resultado en el registro t. Para cada j $(0 \leq j \leq b - 1)$, asignar $r[t][j] := 1$ si **al menos un bit** entre $r[x][j]$ y $r[y][j]$ es igual a 1, y asignar $r[t][j] := 0$ caso contrario.
- $xor(t, x, y)$: Calcula el valor XOR bit a bit de los registros x e y, y guarda el resultado en el registro t. Para cada j $(0 \leq j \leq b - 1)$, asignar $r[t][j] := 1$ si **exactamente un bit** entre $r[x][j]$ y $r[y][j]$ es igual a 1, y asignar $r[t][j] := 0$ caso contrario.
- $not(t, x)$: Calcula el valor NOT bit a bit del registro x, y guarda 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 todos los bits en el registro x, p veces hacia la izquierda, y guardar el resultado en el registro t. El resultado de desplazar los bits en el registro x p veces hacia la

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

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

- $add(t, x, y)$: Sumar los valores enteros guardados en el registro x y el registro y, y guarda el resultado en el registro t. La suma se realiza modulo 2^b. Formalmente, sea X el valor entero guardado en el registro x, Y el valor entero guardado en el registro y antes de la operación. Sea T el valor entero guardado en el registro t luego de la operación. Si $X + Y < 2^b$, asignar los bits de t tal que $T = X + Y$. Caso contrario, asignar los bits de t tal que $T = X + Y - 2^b$.

A Christopher le gustaría que resuelvas dos tipos de tareas utilizando el nuevo procesador. El tipo de una tarea se denota como un entero s. Para ambos tipos de tareas, debes producir un **programa**, que sea una secuencia de las instrucciones descritas anteriormente.

La **entrada** del programa consiste de n números enteros $a[0], a[1], \ldots, a[n-1]$, cada uno de k bits, es decir, $a[i] < 2^k$ ($0 \leq i \leq n-1$). Antes de que se ejecute el programa, todos los números de la entrada se guardan secuencialmente en el registro 0, tal 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 $a[i]$. Nótese que $n \cdot k \leq b$. Todos los demás bits en el registro 0 (es decir, aquellos con índices entre $n \cdot k$ y $b-1$, inclusive) y todos los bits en los demás registros se inicializan con 0.

Correr un programa consiste en ejecutar sus instrucciones en orden. Luego de que la última instrucción es ejecutada, la **salida** del programa se computa basándose en los valores finales 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 para cada i ($0 \leq i \leq n-1$), $c[i]$ es el valor entero de la secuencia que consiste de los bits $i \cdot k$ a $(i+1) \cdot k - 1$ en el registro 0. Nótese que luego de correr el programa, los bits restantes del registro 0 (con índices de al menos $n \cdot k$) y todos los bits de los demás registros pueden ser valores arbitrarios.

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

- La segunda tarea $(s = 1)$ es ordenar los enteros 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 entero más pequeño entre $a[0], a[1], \ldots, a[n-1]$ (es decir, $c[0]$ es el entero más pequeño entre todos enteros de entrada).

Entrégale programas a Christopher, que consistan de a lo sumo q instrucciones cada uno, que resuelvan estas 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 que realicen la tarea requerida.

Esta función debe llamar a 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 cada instrucción relevante, t, x, y deben ser al menos 0 y a lo sumo $m - 1$.
- Para cada instrucción relevante, t, x, y no son necesariamente distintos entre sí.
- 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.

También puedes llamar a la siguiente función para ayudarte a probar tu solución:

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

- Cualquier llamada a esta función será ignorada durante la evaluación de tu solución.
- En el evaluador de ejemplo, esta función agrega una operación $print(t)$ al programa.
- Cuando el evaluador de ejemplo encuentra una operación $print(t)$ durante la ejecución de un programa, imprime n enteros de k-bits formados por los primeros $n \cdot k$ bits del registro t (ver la sección "Evaluador de ejemplo" para más detalles).
- t debe satisfacer $0 \leq t \leq m - 1$.
- Cualquier llamada a este procedimiento no aumenta el número de instrucciones construidas.

Luego de agregar la última instrucción, `construct_instructions` debe retornar. El programa luego es evaluado en 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 un caso de prueba si la salida del programa $c[0], c[1], \ldots, c[n-1]$ para la entrada dada satisface las siguientes condiciones:

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

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

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

Ejemplos

Ejemplo 1

Suponga que $s = 0$, $n = 2$, $k = 1$, $q = 1000$. Hay dos números enteros de entrada $a[0]$ y $a[1]$, cada uno teniendo $k = 1$ bits. Antes de que se ejecute el programa, $r[0][0] = a[0]$ y $r[0][1] = a[1]$. Se asigna 0 a todos los demás bits del procesador. Después de que todas las instrucciones hayan sido ejecutadas, necesitamos tener $c[0] = r[0][0] = \min(a[0], a[1])$, que es el mínimo entre $a[0]$ y $a[1]$.

Solo hay 4 entradas posibles 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: $a[0] = 1, a[1] = 1$

Podemos notar que para los 4 casos, $\min(a[0], a[1])$ es igual al AND bit a bit de $a[0]$ y $a[1]$. Por tanto, una posible solución es construir un programa realizando las siguientes llamadas:

1. `append_move(1, 0)`, el cual agrega una instrucción para copiar $r[0]$ a $r[1]$.
2. `append_right(1, 1, 1)`, el cual agrega una instrucción que toma todos los bits en $r[1]$, los desplaza a la derecha 1 bit y luego almacena el resultado en $r[1]$. Dado que cada entero tiene longitud de 1 bit, esto resulta en que $r[1][0]$ sea igual a $a[1]$.
3. `append_and(0, 0, 1)`, el cual agrega una instrucción para tomar el AND bit a bit de $r[0]$ y $r[1]$, luego almacena el resultado en $r[0]$. Después de que se ejecuta esta instrucción, $r[0][0]$ es asignado el AND bit a bit de $r[0][0]$ y $r[1][0]$, el cual es igual al AND bit a bit de $a[0]$ y $a[1]$, como fué deseado.

Ejemplo 2

Suponga $s = 1,\ n = 2,\ k = 1,\ q = 1000$. Como el ejemplo anterior, solo hay 4 posibles entradas para el programa. Para los 4 casos, $\min(a[0], a[1])$ es el AND bit a bit de $a[0]$ y $a[1]$ y $\max(a[0], a[1])$ es el OR 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)`

Después de ejecutar estas instrucciones, $c[0] = r[0][0]$ contiene $\min(a[0], a[1])$, y $c[1] = r[0][1]$ contiene $\max(a[0], a[1])$, el cual ordena la entrada.

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) $s = 1, q = 4000$

Evaluador de ejemplo

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

- línea $1:\ s\ n\ k\ q$

Esta es seguida por t líneas, cada una describiendo un simple caso de prueba. Cada caso de prueba se proporciona en el siguiente formato:

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

y describe un caso de prueba cuya entrada consiste de n enteros $a[0], a[1], ..., a[n-1]$. La descripción de todos los casos de prueba esta seguida por una línea simple conteniendo solamente -1.

El evaluador de ejemplo primero llama `construct_instructions(s, n, k, q)`. Si esta llamada viola alguna restricción descrita en el enunciado del problema, el evaluador de ejemplo imprime uno de los mensajes de error listados en el final de la sección "Detalles de Implementación" y termina. De otra manera, el evaluador de ejemplo primero imprime cada instrucción adjuntada por `construct_instructions(s, n, k, q)` en orden. Para $store$ instrucciones, v es imprimida desde el índice 0 al índice $b-1$.

Luego, el evaluador de ejemplo procesa los casos de prueba en orden. Para cada caso de prueba, este corre el programa construido en la entrada de el caso de prueba.

Por cada operación $print(t)$, sea $d[0], d[1], \ldots, d[n-1]$ una secuencia de enteros, tal que por 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 evaluador imprime esta secuencia en el siguiente formato: `register` t: $d[0]\ d[1]\ \ldots\ d[n-1]$.

Una vez que todas las instrucciones hayan sido ejecutadas, el evaluador de ejemplo imprime la salida del programa.

Si $s = 0$, la salida del evaluador de ejemplo por cada caso de prueba está en el siguiente formato:

- $c[0]$.

Si $s = 1$, la salida del evaluador por cada caso de prueba está en el siguiente formato:

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

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