Registros con bits desplazados

Cristóbal el ingeniero está trabajando en un nuevo tipo de procesador.

El procesador puede acceder a m celdas de memoria diferentes, de b bits cada una ($m = 100$ y $b = 2000$). Estas celdas se llaman **registros**, se numeran del 0 al $m - 1$ y los denotamos con $r[0], r[1], \ldots, r[m-1]$.

Cada registro es un arreglo de b bits, numerados del 0 (el bit más a la derecha) hasta el $b - 1$ (el bit más a la izquierda). Para cada i ($0 \leq i \leq m - 1$) y cada j ($0 \leq j \leq b - 1$), decimos que $r[i][j]$ es el bit j del registro i.

Para cada secuencia de bits $d_0, d_1, \ldots, d_{l-1}$ (de una longitud arbitraria 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 es el valor entero dado por la secuencia de sus bits, es decir $2^0 \cdot r[i][0] + 2^1 \cdot r[i][1] + \ldots + 2^{b-1} \cdot r[i][b-1]$.

El procesador tiene 9 tipos de **instrucciones** que sirven para modificar los bits de los registros. Cada instrucción opera sobre uno o más registros y guarda el resultado en uno de los registros. En la descripción de las operaciones, la notación $x := y$ significa que cambiamos el valor de x para que coincida con el de y. Las 9 operaciones y sus descripciones se detallan a continuación

- $move(t, y)$: Copia el arreglo de bits del registro y al registro t. Para cada j $(0 \leq j \leq b - 1)$, ponemos $r[t][j] := r[y][j]$.
- $store(t, v)$: Pone en el registro t el valor v, donde v es un arreglo de b bits. Para cada j $(0 \leq j \leq b - 1)$, ponemos $r[t][j] := v[j]$.
- $and(t, x, y)$: Toma el **and** bit-a-bit entre los registros x e y, y guarda el resultado en el registro t. Para cada j $(0 \leq j \leq b - 1)$, pone $r[t][j] := 1$ si **tanto** $r[x][j]$ como $r[y][j]$ son 1, pone $r[t][j] := 0$ caso contrario.
- $or(t, x, y)$: Toma el **or** bit-a-bit entre los registros x e y, y guarda el resultado en el registro t. Para cada j $(0 \leq j \leq b - 1)$, pone $r[t][j] := 1$ si **al menos uno** de entre $r[x][j]$ y $r[y][j]$ es 1, y pone $r[t][j] := 0$ caso contrario.
- $xor(t, x, y)$: Toma el **xor** bit-a-bit entre los registros x e y, y guarda el resultado en el registro t. Para cada j $(0 \leq j \leq b - 1)$, pone $r[t][j] := 1$ si **exactamente uno** de entre $r[x][j]$ y $r[y][j]$ es 1, y pone $r[t][j] := 0$ caso contrario.
- $not(t, x)$: Toma el **not** bit-a-bit del registro x y guarda el resultado en el registro t. Para cada j $(0 \leq j \leq b - 1)$, pone $r[t][j] := 1 - r[x][j]$.

- $left(t, x, p)$: Shiftea los bits del registro x hacia la **izquierda** p posiciones, y guarda el resultado en el registro t. El resultado de shiftear los bits en el registro x hacia la izquierda p posiciones 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 $v[j] = 0$ caso contrario. Para cada j $(0 \leq j \leq b-1)$, pone $r[t][j] := v[j]$.

- $right(t, x, p)$: Shiftea los bits del registro x hacia la **derecha** p posiciones, y guarda el resultado en el registro t. El resultado de shiftear los bits en el registro x hacia la derecha p posiciones 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 $v[j] = 0$ caso contrario. Para cada j $(0 \leq j \leq b-1)$, pone $r[t][j] := v[j]$.

- $add(t, x, y)$: Suma los valores enteros guardado en los registros x e y, y pone el resultado en el registro t. La suma se hace módulo 2^b. Formalmente, sea X el valor entero guardado en el registro x, y Y el valor entero guardado en el registro y antes de la operación. Sea T el entero guardado en el registro t luego de la operación. Si $X + Y < 2^b$, pone los bits de t, de tal forma que $T = X + Y$. Caso contrario, los pone de tal manera que $T = X + Y - 2^b$.

Cristóbal quiere que le ayudes a resolver dos tipos de tareas usando el nuevo procesador. el tipo de tarea se denota con un entero s. Para ambos tipos de tareas, tenés que crear un **programa**, es decir una secuencia de instrucciones como las definidas arriba.

La **input** al programa consiste en n enteros $a[0], a[1], \ldots, a[n-1]$, cada uno con k bits, es decir $a[i] < 2^k$ ($0 \leq i \leq n-1$). Entes de la ejecución del programa, todos los números de la input se guardan en el registro 0 de manera secuencial, es decir que para cada i $(0 \leq i \leq n-1)$ el valor entero de la secuencia $r[0][i \cdot k], r[0][i \cdot k + 1], \ldots, r[0][(i+1) \cdot k - 1]$ de k bits es $a[i]$. Notá que $n \cdot k \leq b$. Todos los demás bits del registro 0 (aquellos con índices entre $n \cdot k$ y $b-1$ inclusive) y todos los bits de todos los otros registros son inicialmente 0.

Correr un programa consiste en ejecutar en orden sus instrucciones. Luego de la ejecución de la última instrucción, la **output** de tu programa se extrae de los valores de los bits del registro 0. Más específicamente, la output 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 $i \cdot k$ to $(i+1) \cdot k - 1$ de bits del registro 0.

Notá que después de correr el programa, los bits restantes del registro 0 (los que tienen índices mayores o iguales a $n \cdot k$) y todos los bits de todos los otros registros pueden tener valores arbitrarios.

- El primer tipo de tarea $(s = 0)$ es encontrar el menor de los enteros de la input $a[0], a[1], \ldots, a[n-1]$. Especificamente, $c[0]$ debe ser el mínimo de entre $a[0], a[1], \ldots, a[n-1]$. Los valores $c[1], c[2], \ldots, c[n-1]$ pueden ser arbitrarios.

- El segundo tipo de tarea $(s = 1)$ consiste en ordenar los enteros de la input $a[0], a[1], \ldots, a[n-1]$ en orden no decreciente. Específicamente, para cada i ($0 \leq i \leq n-1$), $c[i]$ debe ser el $1+i$-ésimo entero más chico de entre $a[0], a[1], \ldots, a[n-1]$, es decir $c[0]$ es el más chico de los enteros de la input.

Dale a Cristóbal programas de a lo sumo q instrucciones cada uno que resuelvan estas tareas.

Detalles de implementación

Tenés que implementar la siguiente función:

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

- s: el tipo de la tarea.
- n: la cantidad de enteros en la input.
- k: cantidad de bits en cada entero de la input.
- q: cantidad máxima de instrucciones permitidas.
- Esta función se va a llamar solo una vez, y debe construir la secuencia de instrucciones que resuelvan la tarea correspondiente.

Para construir la secuencia de instrucciones, tu función debe llamar una o más de las siguientes funciones:

```
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 la 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 final de tu programa, respectivamente.
- Para las instrucciones que corresponda, t, x, y deben ser al menos 0 y a lo sumo $m - 1$.
- Para las instrucciones que corresponda, t, x, y no necesariamente deben ser distintos dos-a-dos.
- Para las instrucciones $left$ y $right$, p debe ser al menos 0 y a lo sumo b.
- Para la instrucción $store$, la longitud de v debe ser b.

Para ayudarte a testear tu función, podés llamar también a la siguiente función:

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

- Llamadas a esta función van a ser ignoradas por el juez.
- En el evaluador local, esta función agrega la instrucción $print(t)$ al final del programa.
- Cuando el evaluador local trate de ejecutar la instrucción $print(t)$, escribe n enteros de k bits formados por los primeros $n \cdot k$ bits del registro t (más detalles en la sección "Evaluador local").
- t debe cumplir $0 \leq t \leq m - 1$.

- Esta instrucción no cuenta a la hora de determinar la cantidad de instrucciones producidas.

Luego de agregar la última instrucción al programa, `construct_instructions` debe retornar. Después de esto, el programa que construiste se pone a prueba con algunos casos de prueba, donde cada uno corresponde a una input que consiste en n enteros de k bits $a[0], a[1], \ldots, a[n-1]$. Para que tu programa pase cierto caso de prueba, la output $c[0], c[1], \ldots, c[n-1]$ del programa para la input $a[0], a[1], \ldots, a[n-1]$ debe satisfacer:

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

Cuando juzguemos tu solución se pueden producir algunos de los siguientes mensajes de error:

- `Invalid index` (Índice inválido): un índice incorrecto (quizás negativo) para un registro se pasó como parámetro t, x o y a alguna llamada de las funciones provistas.
- `Value to store is not b bits long` (Valor a guardar no tiene b bits): la longitud del arreglo v de bits pasado a `append_store` no es b.
- `Invalid shift value` (Valor inválido de shifteo): el valor p pasado a `append_left` o `append_right` no está entre 0 y b inclusive.
- `Too many instructions` (Muchas instrucciones): tu función intentó agregar más de q instrucciones al programa.

Ejemplos

Ejemplo 1

Suponé que $s = 0$, $n = 2$, $k = 1$, $q = 1000$. Hay dos enteros $a[0]$ y $a[1]$ en la input, cada uno con $k = 1$ bits. Antes de la ejecución del programa, $r[0][0] = a[0]$ y $r[0][1] = a[1]$. Todos los otros bits del procesador son 0. Luego de la ejecución de todas las instrucciones del programa, necesitamos que $c[0] = r[0][0] = \min(a[0], a[1])$, que es el mínimo entre $a[0]$ y $a[1]$.

Hay solo 4 posibles inputs 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: $a[0] = 1, a[1] = 1$

Se puede ver que para los 4 casos, $\min(a[0], a[1])$ es igual al and bit-a-bit entre $a[0]$ y $a[1]$. Una solución prosible es, entonces, construir el programa con las siguientes llamadas:

1. `append_move(1, 0)`, que agrega una instrucción que copia $r[0]$ a $r[1]$.
2. `append_right(1, 1, 1)`, que agrega una instrucción que agarra los bits de $r[1]$, y los corre a la derecha 1 posición, y guarda el resultado de nuevo en $r[1]$. Como cada entero tiene un bit, esto hace que $r[1][0]$ termine siendo $a[1]$.
3. `append_and(0, 0, 1)`, que agrega una instrucción que calcula el and bit-a-bit entre $r[0]$ y $r[1]$, y guarda el resultado en $r[0]$. Después de la ejecución de esta instrucción, $r[0][0]$ es el

and bit-a-bit entre $r[0][0]$ y $r[1][0]$, que es el and bit-a-bit entre $a[0]$ y $a[1]$, como queríamos.

Ejemplo 2

Suponé que $s = 1,\ n = 2,\ k = 1,\ q = 1000$. Como en el caso anterior, hay solo 4 inputs posibles para el programa. Para los 4 casos, $\min(a[0], a[1])$ es el and bit-a-bit entre $a[0]$ y $a[1]$, y $\max(a[0], a[1])$ es el or bit-a-bit de $a[0]$ y $a[1]$. Una posible solución consiste en la siguiente sucesión de 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 las 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 un ordenamiento de la input.

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 cada $0 \leq i \leq n - 1$)

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 Local

El evaluador local lee la entrada en el siguiente formato:

- línea 1 : $s\ n\ k\ q$

A esto le siguen cierta cantidad de líneas, cada una describiendo un caso de prueba. Cada una de esas líneas describe la input del caso de prueba en el siguiente formato:

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

y describe un caso de prueba donde la input consiste en los n enteros $a[0], a[1], ..., a[n-1]$. Luego de la descripción de todos los casos, va una línea que contiene únicamente un -1.

El evaluador local llama primero a la función `construct_instructions(s, n, k, q)`. Si esta llamada rompe alguna restricción mencionada en la descripción del problema, se imprime alguno de los mensajes de error listados en la sección de "Detalles de Implementación", y luego termina inmediatamente.

Caso constrario, el evaluador local imprime primero la lista de instrucciones generada por `construct_instructions(s, n, k, q)`, en el orden en el que fueron generadas. Para la instrucción $store,\ v$ se imprime desde el índice 0 hasta el índice $b-1$.

Después de esto, el evaluador local procesa los casos de prueba en orden, uno por uno. Por cada caso de prueba, corre el programa que construiste con la input de ese caso.

Para cada operación $print(t)$, sea $d[0], d[1], \ldots, d[n-1]$ la secuencia de enteros tales que para cada i ($0 \le i \le n-1$), $d[i]$ es el valor entero de la secuencia de bits $i \cdot k$ a $(i+1) \cdot k - 1$ del registro t, en el momento en el que la instrucción se ejecuta. El evaluador local imprime esta secuencia en el formato: `register` t: $d[0]\ d[1]\ \ldots\ d[n-1]$.

Luego de que se ejecutan todas las instrucciones, el evaluador local imprime la output del programa.

Si $s = 0$, Para cada caso de prueba el evaluador local imprime la output del programa en el formato:

- $c[0]$.

Si $s = 1$, la output se imprime en el formato:

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

Luego de la ejecución de todos los casos de prueba, el evaluador local imprime `number of instructions:` X donde X es la cantidad de instrucciones en tu programa.