Registros de recorrimientos de bits

Christopher el ingeniero está trabajando en nuevo tipo de procesador de computadora.

El procesador tiene acceso a m distintas celdas de memoria de b bits cada una (donde $m = 100$ y $b = 2000$), que son llamadas **registros** y están numeradas de 0 a $m - 1$.

Denotamos los registros como $r[0], r[1], \ldots, r[m-1]$.

Cada registro es un arreglo de b bits, numerado de 0 (bit más a la derecha) a $b - 1$ (bit más a la izquierda). Para cada i $(0 \leq i \leq m - 1)$ y para cada j $(0 \leq j \leq b - 1)$, denotamos el j-ésimo bit del registro i como $r[i][j]$.

Para cualquier subsecuencia de bits $d_0, d_1, \ldots, d_{l-1}$ (de 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 el valor entero de la secuencia de bits, i.e., 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 **instruciones** que se pueden usar para modificar los bits del registro. Cada insrtuccion opera en uno o más registros y guarda la salida en uno de los registros. A continuación, usamos $x := y$ para denotar una operación que cambia el valor de x de tal forma que se vuelve igual a y. Las operaciones realizadas por cada tipo de instrucción son descritas a continuación:

- $move(t, y)$: Copia el registro de bits del arreglo y al arreglo t. Para cada j $(0 \leq j \leq b - 1)$, asigna $r[t][j] := r[y][j]$.
- $store(t, v)$: Asigna el registro t igual a v, donde v es un arrerglo de b bits. Para cada j $(0 \leq j \leq b - 1)$, asigna $r[t][j] := v[j]$.
- $and(t, x, y)$: Toma la operación AND (bit por bit) de los registros x y y, y lo guarda en el registro t. Para cada j $(0 \leq j \leq b - 1)$, asigna $r[t][j] := 1$ si **ambos** $r[x][j]$ y $r[y][j]$ son 1, y asigna $r[t][j] := 0$ de lo contrario.
- $or(t, x, y)$: Toma la operación OR (bit por bit) de los registros x y y, y lo guarda en el registro t. Para cada j $(0 \leq j \leq b - 1)$, asigna $r[t][j] := 1$ si **al menos uno** de $r[x][j]$ y $r[y][j]$ es 1, y asigna $r[t][j] := 0$ de lo contrario.
- $xor(t, x, y)$:Toma la operación XOR (bit por bit) de los registros x y y, y lo guarda en el registro t. Para cada j $(0 \leq j \leq b - 1)$, asigna $r[t][j] := 1$ si **exactamente uno** de $r[x][j]$ y $r[y][j]$ es 1, y asigna $r[t][j] := 0$ de lo contrario.

- $not(t, x)$: Toma la operación NOT (bit por bit) del registro x, y guarda el resultado en el registo t. Para cada j $(0 \leq j \leq b-1)$, asigna $r[t][j] := 1 - r[x][j]$.
- $left(t, x, p)$: Recorre todos los bits en el registro x a la izquierda p lugares, y guarda el resultado en el registro t. El resultado de recorrer los bits del registro x a la izquierda p posiciones es en 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 $(0 \leq j \leq b-1)$, asigna $r[t][j] := v[j]$.
- $right(t, x, p)$: Recorre todos los bits en el registro x a la derecha p lugares, y guarda el resultado en el registro t. El resultado de recorrer los bits del registro x a la derecha p posiciones es en 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 $(0 \leq j \leq b-1)$, asigna $r[t][j] := v[j]$.
- $add(t, x, y)$: Suma los valores guardados en el registro x y el registro y, y guarda el resultado en el registro t. La suma se realiza 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 después de la operación. Si $X + Y < 2^b$, asigna los bits de t, de tal forma que $T = X + Y$. De lo contrario, asigna los bits de t, de tal forma que $T = X + Y - 2^b$.

Christopher quiere que tú resuelvas dos tipos de tareas usando el nuevo procesado. El tipo de tareas es denotado por un entero s. Para ambos tipos de tareas, tienes que producir un **progama**, es decir una secuencia de instrucciones definidas a continuación.

La **entrada** de un programa consiste en n enteros $a[0], a[1], \ldots, a[n-1]$, donde cada uno tiene k -bits, i.e., $a[i] < 2^k$ ($0 \leq i \leq n-1$).

Antes de que el programa sea ejecutado, todos los números de la entrada son guardados secuencialmente en el registro 0, de tal forma 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 $n \cdot k \leq b$.

Todos los otros bits en el registro 0 (i.e., los que tienen índices entre $n \cdot k$ y $b-1$, inclusive) y todos los bits en los otros registros son inicializados como 0.

Correr un programa consiste en ejecutar sus instrucciones en orden. Después de que la última intrucción es ejecutada, la **salida** del programa es calculada basada en los valores finales de los bits del registo 0. Especificamente, 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 en los bits $i \cdot k$ a $(i+1) \cdot k - 1$ del registro 0. Nota que después de correr el programa, los bits restantes del registro 0 (con índices al menos $n \cdot k$) y todos los demás bits de los otros registros pueden tener valor arbitrario.

- La primera tarea $(s = 0)$ consiste en enontrar el entero más chico entre los enteros de la entrada $a[0], a[1], \ldots, a[n-1]$. Especificamente, , $c[0]$ 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 $(s = 1)$ consiste en ordenar los enteros de la entrada $a[0], a[1], \ldots, a[n-1]$ en orden no creciente. Especificamete, para cada i ($0 \leq i \leq n-1$), $c[i]$ debe ser igual al $1+i$-ésimo entero más chico entre $a[0], a[1], \ldots, a[n-1]$ (i.e., $c[0]$ es el entero más chico entre los enteros de la entrada).

Dale a Christopher programas que consistan de a lo más q instrucciones cada uno, de tal forma que puedan resolver las tareas.

Detalles de implementación

Debes implementar el siguiente procedimiento:

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

- s: tipo de tarea.
- n: número de enteros en la entrada.
- k: número de bits de cada entero de la entrada.
- q: cantidad máxima de instrucciones permitidas.
- Este pocedimiento debe ser llamado exactamente una vez y debe constuir una secuencia de instrucciones que realicen la tarea solicitada.

El procedimiento debe llamar uno o más de los siguientes métodos 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 procedimiento 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 programa, respectivamente.
- Para todas las instrucciones relevantes, t, x, y debe ser al menos 0 y a lo más $m-1$.
- Para todas las instrucciones relevantes, t, x, y no son necesariamente distintos entre ellos.
- Para las instrucciones $left$ y $right$, p debe ser al menos 0 y a lo más b.
- Para las instrucciones $store$, la longitud de v debe ser b.

Puedes llamar el siguiente procedimiento para ayudar a probar tu solución:

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

- Cualquier llamada a este procedimiento será ignorada al evaluar tu solución.
- En el evaluador de ejemplo, este procedimiento agrega la operación $print(t)$ al programa.
- Cuando el evaluador de ejemplo encuentra la operación $print(t)$ durante la ejecución del programa, imprime n k-bit 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 la cantidad de llamadas de tu programa.

Después de agregar la última instrucción, `construct_instructions` deber regresar. El programa es después evaluado con algunos casos de prueba, cada uno especificando una entrada de n enteros de k bits: $a[0], a[1], \ldots, a[n-1]$.

Tu solución resuelve el caso de prueba, si la salida del programa $c[0], c[1], \ldots, c[n-1]$ para la entrada asignada, satisface las siguientes condicions.:

- Si $s = 0$, $c[0]$ es el valor más chico 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 l $1 + i$-ésimo entero más chico entre $a[0], a[1], \ldots, a[n-1]$.

La evaluación de tu programa puede resultar en alguno de los siguientes mensajes de error:

- `Invalid index`: un índice de registro (posiblemente negativo) fue asignado como parámetro t, x o y en alguno de los procedimientos.
- `Value to store is not b bits long`: la longitud de v asignada en `append_store` es distinta de b.
- `Invalid shift value`: el valor de p asignado en `append_left` o `append_right` no se encuentra entre 0 y b inclusive.
- `Too many instructions`: tu procedimiento intentó asignar 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 $k = 1$ bits. Antes de que el programa sea ejecutado, $r[0][0] = a[0]$ y $r[0][1] = a[1]$. Todos los otros bits en el procesador son asignados a 0. Después de que todas las instrucciones en el programa sean ejecutadas, tenemos que 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: $a[0] = 1, a[1] = 1$

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

1. `append_move(1, 0)`, que agrega la instrucción para copiar $r[0]$ a $r[1]$.
2. `append_right(1, 1, 1)`, que agrega una instrucción donde agarra $r[1]$, y lo recorre a la derecha 1 bit, y pone el resultado de regreso en $r[1]$. Como cada entero es de longitud de 1-bit, el resultado en $r[1][0]$ se vuelve igual a $a[1]$.
3. `append_and(0, 0, 1)`, que agrega la instrucción que toma el AND (bit por bit) de $r[0]$ y $r[1]$, y guarda el resultado en $r[0]$. Después de que esta intrucción es ejecutada, $r[0][0]$ es asignado a la operación AND (bit por bit) de $r[0][0]$ y $r[1][0]$, que es igual a la operación AND (bit por bit) de $a[0]$ y $a[1]$, como era deseado.

Ejemplo 2

Supongamos $s = 1,\ n = 2,\ k = 1,\ q = 1000$. Como con el ejemplo anterior, sólo hay 4 posibles entradas para el programa. para los 4 casos, $\min(a[0], a[1])$ es el AND (bit por bit) de $a[0]$ y $a[1]$, y $\max(a[0], a[1])$ es el OR (bit por bit) de $a[0]$ y $a[1]$.Una posible solución es hacer 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 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 la entrada ordenada.

Restricciones

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

Subtareas

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

Evaluador de ejemplo

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

- línea 1 : $s\ n\ k\ q$

Esto es seguido por alún número de líneas, cada una describiendo un solo caso de prueba. Cada caso de prueba se da en el siguiente formato:

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

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

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

Luego, el evaluador de ejemplo procesa los casos de prueba en orden. Para cada caso de prueba, se corre el programa construido en la entrada de cada 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 ($0 \le i \le n-1$), $d[i]$ es el valor entero de la secuencia de bits $i \cdot k$ a $(i+1) \cdot k - 1$ del registro t (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 han sido ejecutadas, el evaluador de ejemplo imprime la salida del programa.

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

- $c[0]$.

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

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

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