Bit Shift Registers

Christopher the engineer is working on a new type of computer processor.

The processor has access to m different b-bit memory cells (where $m = 100$ and $b = 2000$), which are called **registers**, and are numbered from 0 to $m - 1$. We denote the registers by $r[0], r[1], \ldots, r[m-1]$. Each register is an array of b bits, numbered from 0 (the rightmost bit) to $b - 1$ (the leftmost bit). For each i $(0 \leq i \leq m - 1)$ and each j $(0 \leq j \leq b - 1)$, we denote the j-th bit of register i by $r[i][j]$.

For any sequence of bits $d_0, d_1, \ldots, d_{l-1}$ (of arbitrary length l), the **integer value** of the sequence is equal to $2^0 \cdot d_0 + 2^1 \cdot d_1 + \ldots + 2^{l-1} \cdot d_{l-1}$. We say that the **integer value stored in a register** i is the integer value of the sequence of its bits, i.e., it is $2^0 \cdot r[i][0] + 2^1 \cdot r[i][1] + \ldots + 2^{b-1} \cdot r[i][b-1]$.

The processor has 9 types of **instructions** that can be used to modify the bits in the registers. Each instruction operates on one or more registers and stores the output in one of the registers. In the following, we use $x := y$ to denote an operation of changing the value of x such that it becomes equal to y. The operations performed by each type of instruction are described below.

- $move(t, y)$: Copy the array of bits in register y to register t. For each j $(0 \leq j \leq b - 1)$, set $r[t][j] := r[y][j]$.
- $store(t, v)$: Set register t to be equal to v, where v is an array of b bits. For each j $(0 \leq j \leq b - 1)$, set $r[t][j] := v[j]$.
- $and(t, x, y)$: Take the bitwise-AND of registers x and y, and store the result in register t. For each j $(0 \leq j \leq b - 1)$, set $r[t][j] := 1$ if **both** $r[x][j]$ and $r[y][j]$ are 1, and set $r[t][j] := 0$ otherwise.
- $or(t, x, y)$: Take the bitwise-OR of registers x and y, and store the result in register t. For each j $(0 \leq j \leq b - 1)$, set $r[t][j] := 1$ if **at least one** of $r[x][j]$ and $r[y][j]$ are 1, and set $r[t][j] := 0$ otherwise.
- $xor(t, x, y)$: Take the bitwise-XOR of registers x and y, and store the result in register t. For each j $(0 \leq j \leq b - 1)$, set $r[t][j] := 1$ if **exactly one** of $r[x][j]$ and $r[y][j]$ is 1, and set $r[t][j] := 0$ otherwise.
- $not(t, x)$: Take the bitwise-NOT of register x, and store the result in register t. For each j $(0 \leq j \leq b - 1)$, set $r[t][j] := 1 - r[x][j]$.
- $left(t, x, p)$: Shift all bits in register x to the left by p, and store the result in register t. The result of shifting the bits in register x to the left by p is an array v consisting of b bits. For each

j $(0 \leq j \leq b-1)$, $v[j] = r[x][j-p]$ if $j \geq p$, and $v[j] = 0$ otherwise. For each j $(0 \leq j \leq b-1)$, set $r[t][j] := v[j]$.

- $right(t, x, p)$: Shift all bits in register x to the right by p, and store the result in register t. The result of shifting the bits in register x to the right by p is an array v consisting of b bits. For each j $(0 \leq j \leq b-1)$, $v[j] = r[x][j+p]$ if $j \leq b-1-p$, and $v[j] = 0$ otherwise. For each j $(0 \leq j \leq b-1)$, set $r[t][j] := v[j]$.

- $add(t, x, y)$: Add the integer values stored in register x and register y, and store the result in register t. The addition is carried out modulo 2^b. Formally, let X be the integer value stored in register x, and Y be the integer value stored in register y before the operation. Let T be the integer value stored in register t after the operation. If $X + Y < 2^b$, set the bits of t, such that $T = X + Y$. Otherwise, set the bits of t, such that $T = X + Y - 2^b$.

Christopher would like you to solve two types of tasks using the new processor. The type of a task is denoted by an integer s. For both types of tasks, you need to produce a **program**, that is a sequence of instructions defined above.

The **input** to the program consists of n integers $a[0], a[1], \ldots, a[n-1]$, each having k bits, i.e., $a[i] < 2^k$ ($0 \leq i \leq n-1$). Before the program is executed, all of the input numbers are stored sequentially in register 0, such that for each i $(0 \leq i \leq n-1)$ the integer value of the sequence of k bits $r[0][i \cdot k], r[0][i \cdot k + 1], \ldots, r[0][(i+1) \cdot k - 1]$ is equal to $a[i]$. Note that $n \cdot k \leq b$. All other bits in register 0 (i.e., those with indices between $n \cdot k$ and $b-1$, inclusive) and all bits in all other registers are initialized to 0.

Running a program consists in executing its instructions in order. After the last instruction is executed, the **output** of the program is computed based on the final value of bits in register 0. Specifically, the output is a sequence of n integers $c[0], c[1], \ldots, c[n-1]$, where for each i ($0 \leq i \leq n-1$), $c[i]$ is the integer value of a sequence consisting of bits $i \cdot k$ to $(i+1) \cdot k - 1$ of register 0. Note that after running the program the remaining bits of register 0 (with indices at least $n \cdot k$) and all bits of all other registers can be arbitrary.

- The first task $(s = 0)$ is to find the smallest integer among the input integers $a[0], a[1], \ldots, a[n-1]$. Specifically, $c[0]$ must be the minimum of $a[0], a[1], \ldots, a[n-1]$. The values of $c[1], c[2], \ldots, c[n-1]$ can be arbitrary.

- The second task $(s = 1)$ is to sort the input integers $a[0], a[1], \ldots, a[n-1]$ in nondecreasing order. Specifically, for each i ($0 \leq i \leq n-1$), $c[i]$ should be equal to the $1+i$-th smallest integer among $a[0], a[1], \ldots, a[n-1]$ (i.e., $c[0]$ is the smallest integer among the input integers).

Provide Christopher with programs, consisting of at most q instructions each, that can solve these tasks.

Implementation Details

You should implement the following procedure:

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

- s: type of task.
- n: number of integers in the input.
- k: number of bits in each input integer.
- q: maximum number of instructions allowed.
- This procedure is called exactly once and should construct a sequence of instructions to perform the required task.

This procedure should call one or more of the following procedures to construct a sequence of instructions:

```
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)
```

- Each procedure appends a $move(t, y)$ $store(t, v)$, $and(t, x, y)$, $or(t, x, y)$, $xor(t, x, y)$, $not(t, x)$, $left(t, x, p)$, $right(t, x, p)$ or $add(t, x, y)$ instruction to the program, respectively.
- For all relevant instructions, t, x, y must be at least 0 and at most $m - 1$.
- For all relevant instructions, t, x, y are not necessarily pairwise distinct.
- For $left$ and $right$ instructions, p must be at least 0 and at most b.
- For $store$ instructions, the length of v must be b.

You may also call the following procedure to help you in testing your solution:

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

- Any call to this procedure will be ignored during the grading of your solution.
- In the sample grader, this procedure appends a $print(t)$ operation to the program.
- When the sample grader encounters a $print(t)$ operation during the execution of a program, it prints n k-bit integers formed by the first $n \cdot k$ bits of register t (see "Sample Grader" section for details).
- t must satisfy $0 \leq t \leq m - 1$.
- Any call to this procedure does not add to the number of constructed instructions.

After appending the last instruction, `construct_instructions` should return. The program is then evaluated on some number of test cases, each specifying an input consisting of n k-bit integers

$a[0], a[1], \ldots, a[n-1]$. Your solution passes a given test case if the output of the program $c[0], c[1], \ldots, c[n-1]$ for the provided input satisfies the following conditions:

- If $s = 0$, $c[0]$ should be the smallest value among $a[0], a[1], \ldots, a[n-1]$.
- If $s = 1$, for each i ($0 \leq i \leq n-1$), $c[i]$ should be the $1+i$-th smallest integer among $a[0], a[1], \ldots, a[n-1]$.

The grading of your solution may result in one of the following error messages:

- `Invalid index`: an incorrect (possibly negative) register index was provided as parameter t, x or y for some call of one of the procedures.
- `Value to store is not b bits long`: the length of v given to `append_store` is not equal to b.
- `Invalid shift value`: the value of p given to `append_left` or `append_right` is not between 0 and b inclusive.
- `Too many instructions`: your procedure attempted to append more than q instructions.

Examples

Example 1

Suppose $s = 0$, $n = 2$, $k = 1$, $q = 1000$. There are two input integers $a[0]$ and $a[1]$, each having $k = 1$ bit. Before the program is executed, $r[0][0] = a[0]$ and $r[0][1] = a[1]$. All other bits in the processor are set to 0. After all the instructions in the program are executed, we need to have $c[0] = r[0][0] = \min(a[0], a[1])$, which is the minimum of $a[0]$ and $a[1]$.

There are only 4 possible inputs to the program:

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

We can notice that for all 4 cases, $\min(a[0], a[1])$ is equal to the bitwise-AND of $a[0]$ and $a[1]$. Therefore, a possible solution is to construct a program by making the following calls:

1. `append_move(1, 0)`, which appends an instruction to copy $r[0]$ to $r[1]$.
2. `append_right(1, 1, 1)`, which appends an instruction that takes all bits in $r[1]$, shifts them to the right by 1 bit, and then stores the result back in $r[1]$. Since each integer is 1-bit long, this results in $r[1][0]$ being equal to $a[1]$.
3. `append_and(0, 0, 1)`, which appends an instruction to take the bitwise-AND of $r[0]$ and $r[1]$, then store the result in $r[0]$. After this instruction is executed, $r[0][0]$ is set to the bitwise-AND of $r[0][0]$ and $r[1][0]$, which is equal to the bitwise-AND of $a[0]$ and $a[1]$, as desired.

Example 2

Suppose $s = 1,\ n = 2,\ k = 1,\ q = 1000$. As with the earlier example, there are only 4 possible inputs to the program. For all 4 cases, $\min(a[0], a[1])$ is the bitwise-AND of $a[0]$ and $a[1]$, and $\max(a[0], a[1])$ is the bitwise-OR of $a[0]$ and $a[1]$. A possible solution is to make the following calls:

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)`

After executing these instructions, $c[0] = r[0][0]$ contains $\min(a[0], a[1])$, and $c[1] = r[0][1]$ contains $\max(a[0], a[1])$, which sorts the input.

Constraints

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

Subtasks

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

Sample Grader

The sample grader reads the input in the following format:

- line 1 : $s\ n\ k\ q$

This is followed by some number of lines, each describing a single test case. Each test case is provided in the following format:

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

and describes a test case whose input consists of n integers $a[0], a[1], \ldots, a[n-1]$. The description of all test cases is followed by a single line containing solely -1.

The sample grader first calls `construct_instructions(s, n, k, q)`. If this call violates some constraint described in the problem statement, the sample grader prints one of the error messages listed at the end of the "Implementation Details" section and exits. Otherwise, the sample grader first prints each instruction appended by `construct_instructions(s, n, k, q)` in order. For $store$ instructions, v is printed from index 0 to index $b-1$.

Then, the sample grader processes test cases in order. For each test case, it runs the constructed program on the input of the test case.

For each $print(t)$ operation, let $d[0], d[1], \ldots, d[n-1]$ be a sequence of integers, such that for each i ($0 \le i \le n-1$), $d[i]$ is the integer value of the sequence of bits $i \cdot k$ to $(i+1) \cdot k - 1$ of register t (when the operation is executed). The grader prints this sequence in the following format: `register` t: $d[0]\ d[1]\ \ldots\ d[n-1]$.

Once all instructions have been executed, the sample grader prints the output of the program.

If $s = 0$, the output of the sample grader for each test case is in the following format:

- $c[0]$.

If $s = 1$, the output of the sample grader for each test case is in the following format:

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

After executing all test cases, the grader prints `number of instructions:` X where X is the number of instructions in your program.