레지스터

엔지니어인 크리스토퍼는 새로운 형태의 CPU를 설계하고 있다.

이 CPU는 m개의 서로 다른 b 비트 메모리 셀을 가지고 있는데, ($m = 100$, $b = 2000$) 이를 **레지스터**라고 부르며, 0부터 $m-1$로 번호가 매겨져 있다. 레지스터들을 차례로 $r[0], r[1], \ldots, r[m-1]$라고 부르자. 각각의 레지스터는 b 비트 배열이며, 가장 오른쪽 비트를 0으로, 가장 왼쪽 비트를 $b-1$로 순서대로 번호가 매겨진다. 각각의 i $(0 \le i \le m-1)$와 j $(0 \le j \le b-1)$에 대해서, i번 레지스터의 j번째 비트를 $r[i][j]$로 표현한다.

임의의 길이 l의 비트 서열 $d_0, d_1, \ldots, d_{l-1}$이 주어졌을 때, 이 서열의 **정수값**은 $2^0 \cdot d_0 + 2^1 \cdot d_1 + \ldots + 2^{l-1} \cdot d_{l-1}$이다. **레지스터** i에 저장된 **정수값**은 이 레지스터에 저장된 비트 서열의 정수값으로, $2^0 \cdot r[i][0] + 2^1 \cdot r[i][1] + \ldots + 2^{b-1} \cdot r[i][b-1]$이다.

이 CPU는 9가지의 **명령어**를 가지고 레지스터에 저장된 비트를 조작할 수 있다. 각 명령어는 하나 또는 그 이상의 레지스터에 동작하며, 하나의 레지스터에 명령 수행 결과를 저장한다. 앞으로, $x := y$는 x에 저장된 값을 y의 값과 같게 바꾸는 것으로 하자. 각 명령어에 따라 수행될 작업은 다음과 같다.

- $move(t, y)$: 레지스터 y에 저장된 비트를 레지스터 t에 복사한다. 각각 j $(0 \le j \le b-1)$에 대해서, $r[t][j] := r[y][j]$이다.
- $store(t, v)$: 레지스터 t의 값이 v가 되게 한다. v는 b 비트로 된 배열이다. 각각 j $(0 \le j \le b-1)$에 대해서, $r[t][j] := v[j]$이다.
- $and(t, x, y)$: 두 레지스터 x와 y에 대해서 비트대 비트로 AND 연산을 하고, 결과를 레지스터 t에 저장한다. 각각 j $(0 \le j \le b-1)$에 대해서, 만약 $r[x][j]$ 와 $r[y][j]$가 **모두** 1이면 $r[t][j] := 1$이고, 그렇지 않으면 $r[t][j] := 0$ 이다.
- $or(t, x, y)$: 두 레지스터 x와 y에 대해서 비트대 비트로 OR 연산을 하고, 결과를 레지스터 t에 저장한다. 각각 j $(0 \le j \le b-1)$에 대해서, 만약 $r[x][j]$ 와 $r[y][j]$ **둘 중 적어도 하나가** 1이면 $r[t][j] := 1$이고, 그렇지 않으면 $r[t][j] := 0$ 이다.
- $xor(t, x, y)$: 두 레지스터 x와 y에 대해서 비트대 비트로 XOR 연산을 하고, 결과를 레지스터 t에 저장한다. 각각 j $(0 \le j \le b-1)$에 대해서, 만약 $r[x][j]$ 와 $r[y][j]$ **둘 중 정확히 하나가** 1이면 $r[t][j] := 1$이고, 그렇지 않으면 $r[t][j] := 0$ 이다.
- $not(t, x)$: 레지스터 x에 대해서 비트 단위로 NOT 연산을 하고, 결과를 레지스터 t에 저장한다. 각각 j $(0 \le j \le b-1)$에 대해서, $r[t][j] := 1 - r[x][j]$이다.
- $left(t, x, p)$: 레지스터 x의 모든 비트를 왼쪽으로 p 만큼 이동한 값을 레지스터 t에 저장한다. 레지스터 x의 비트들을 왼쪽으로 p 만큼 이동한 값은 b 비트로 이루어진 배열 v이다. 각각 j $(0 \le j \le b-1)$에 대해서, 만약 $j \ge p$이면 $v[j] = r[x][j-p]$이고, 그렇지 않으면 $v[j] = 0$ 이다. 각각 j $(0 \le j \le b-1)$에 대해서, $r[t][j] := v[j]$이다.

- $right(t, x, p)$: 레지스터 x의 모든 비트를 오른쪽으로 p 만큼 이동한 값을 레지스터 t에 저장한다. 레지스터 x의 비트들을 오른쪽으로 p 만큼 이동한 값은 b 비트로 이루어진 배열 v이다. 각각 j $(0 \le j \le b-1)$에 대해서, 만약 $j \le b-1-p$이면 $v[j] = r[x][j+p]$이고, 그렇지 않으면 $v[j] = 0$이다. 각각 j $(0 \le j \le b-1)$에 대해서, $r[t][j] := v[j]$이다.
- $add(t, x, y)$: 레지스터 x에 저장된 정수값과 레지스터 y에 저장된 정수값을 더하고, 그 결과값을 레지스터 t에 저장한다. 덧셈을 한 다음 2^b로 나눈 나머지를 저장한다. 구체적으로는, X가 레지스터 x에 저장된 정수값이고 Y가 레지스터 y에 저장된 정수값이라고 하자. T가 연산이 끝난 후 레지스터 t에 저장될 정수값이라고 하자. 만약 $X + Y < 2^b$이면, $T = X + Y$가 되도록 t의 비트를 설정한다. 그렇지 않으면, $T = X + Y - 2^b$가 되도록 t의 비트를 설정한다.

크리스토퍼는 당신이 새로운 CPU를 이용하여 두 가지 종류의 문제를 풀어줬으면 한다.문제의 종류는 정수 s로 표현된다. 두 종류 모두, 위에서 정의된 명령어들로 이루어진 **프로그램**을 만들어야 한다.

프로그램의 **입력**은 n개의 정수 $a[0], a[1], \ldots, a[n-1]$로 이루어지는데, 각각 k 비트이다. 즉, $a[i] < 2^k$ ($0 \le i \le n-1$)이다. 프로그램을 실행하기 전, 모든 입력은 차례대로 레지스터 0에 저장되어 있다. 각각의 i에 대해서 $(0 \le i \le n-1)$ k 비트 $r[0][i \cdot k], r[0][i \cdot k + 1], \ldots, r[0][(i+1) \cdot k - 1]$ 의 정수값은 $a[i]$와 같다. $n \cdot k \le b$임에 유의하라. 레지스터 0의 다른 비트들 (즉, $n \cdot k$ 이상 $b-1$ 이하 위치)과 다른 레지스터의 모든 비트는 0으로 초기화되어 있다.

프로그램을 실행하는 것은 명령어를 차례대로 수행하는 것으로 이루어진다. 마지막 명령어를 수행한 다음, 프로그램의 **출력**이 레지스터 0에 최종적으로 저장된 비트의 값에 따라 계산된다. 구체적으로는, 출력은 n개의 정수 $c[0], c[1], \ldots, c[n-1]$인데, 각각 i ($0 \le i \le n-1$)에 대해서, $c[i]$는 레지스터 0의 $i \cdot k$번비트부터 $(i+1) \cdot k - 1$ 비트에 저장된 비트들의 정수값이다. 프로그램을 실행한 후 레지스터 0의 나머지 비트와 다른 모든 레지스터의 비트는 임의의 값이 될 수 있음에 유의하라.

- 첫번째 문제 $(s = 0)$는 입력된 정수 $a[0], a[1], \ldots, a[n-1]$중 가장 작은 정수를 찾는 것이다. 구체적으로는, $c[0]$는 $a[0], a[1], \ldots, a[n-1]$ 중 가장 작은 값이어야 한다. $c[1], c[2], \ldots, c[n-1]$의 값은 무엇이 되더라도 상관없다.
- 두번째 문제 $(s = 1)$는 입력된 정수 $a[0], a[1], \ldots, a[n-1]$를 감소하지 않는 순서, 즉 오름차순인데 동점이 있는 순서로 정렬하는 것이다. 구체적으로는, 각각의 i에 대해서 ($0 \le i \le n-1$), $c[i]$는 $a[0], a[1], \ldots, a[n-1]$ 중 $1+i$ 번째로 작은 정수여야 한다. (즉, $c[0]$는 입력 중 가장 작은 정수이다)

크리스토퍼에게, 한 프로그램에 최대 q개의 명령어를 사용하여, 이 문제들을 푸는 프로그램을 짜 주자.

Implementation Details

다음 함수를 구현해야 한다.

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

- s: 문제의 종류
- n: 입력으로 주어지는 정수의 개수
- k: 입력된 정수 하나가 차지하는 비트 길이

- q: 최대 사용할 수 있는 명령어 개수
- 이 함수는 정확하게 한 번 호출되며, 주어진 문제를 풀 수 있는 명령어의 서열을 만들어야 한다.

이 함수는 명령어의 서열을 만들기 위해서, 다음 함수들 중 하나 또는 그 이상을 호출해야 한다.

```
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)
```

- 각 함수는 각각 $move(t, y)$ $store(t, v)$, $and(t, x, y)$, $or(t, x, y)$, $xor(t, x, y)$, $not(t, x)$, $left(t, x, p)$, $right(t, x, p)$, 혹은 $add(t, x, y)$ 명령을 프로그램에 추가한다.
- 각 명령어에 대해서, t, x, y는 0 이상 $m - 1$이하이다.
- 각 명령어에 대해서, t, x, y가 꼭 서로 달라야 하는 것은 아니다.
- $left$와 $right$ 명령에서는 p가 0 이상 b 이하여야 한다.
- $store$명령어에서 v의 길이는 b여야 한다.

당신의 답이 맞는지 테스트해보기 위해서 다음 함수를 호출할 수 있다.

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

- 이 함수는 당신의 답을 채점하는동안 무시된다.
- 샘플 그레이더에서, 이 함수는 $print(t)$ 명령을 프로그램에 추가한다.
- 샘플 그레이더가 $print(t)$ 명령을 프로그램을 수행하는 과정에서 만나면, n개의 k 비트 정수를 출력하는데, 이는 레지스터 t의 처음 $n \cdot k$ 비트를 나타낸다. ("Sample Grader" 부분에서 자세한 내용을 확인)
- t는 $0 \leq t \leq m - 1$이어야 한다.
- 이 함수의 호출은 만들어진 명령어 수를 늘리지 않는다.

마지막 명령어를 추가한 다음, `construct_instructions`는 리턴해야 한다. 프로그램은 여러 개의 테스트 케이스를 이용해서 평가되는데, 각각 테스트 케이스는 n개의 k 비트 정수 $a[0], a[1], \ldots, a[n-1]$로 이루어진다. 당신의 답은 주어진 테스트케이스에 대해서, 프로그램의 출력 $c[0], c[1], \ldots, c[n-1]$이 다음 조건을 만족하면 통과된다.

- $s = 0$이면, $c[0]$는 $a[0], a[1], \ldots, a[n-1]$ 중 가장 작은 값이어야 한다.
- $s = 1$이면, 각각의 i ($0 \leq i \leq n-1$)에 대해서, $c[i]$는 $a[0], a[1], \ldots, a[n-1]$중 $1 + i$ 번째로 작은 값이어야 한다.

당신의 답은 다음 중 하나의 에러 메시지를 만들 수 있다.

- `Invalid index`: 함수를 호출할 때 파라미터 t, x, y 중 하나에서 레지스터 번호를 잘못 주었다 (음의 값을 주었을 수도 있다.)

- `Value to store is not b bits long`: `append_store`에 주어진 v의 길이가 b가 아니다.
- `Invalid shift value`: `append_left` or `append_right` 에 주어진 p 값이 0 이상 b 이하가 아니다.
- `Too many instructions`: q개가 넘는 명령어를 프로그램에 추가하려고 했다.

Examples

Example 1

$s = 0,\ n = 2,\ k = 1,\ q = 1000$이라고 하자. 입력은 두 정수 $a[0]$와 $a[1]$인데, 각각 $k = 1$ 비트이다. 프로그램을 수행하기 전에, $r[0][0] = a[0]$이고 $r[0][1] = a[1]$이다. 다른 비트는 0으로 초기화되어 있다. 프로그램의 모든 명령을 수행한 후에, $c[0] = r[0][0] = \min(a[0], a[1])$, 즉 $a[0]$와 $a[1]$ 중 작은 쪽 값이어야 한다.

이 프로그램에 주어지는 입력은 4가지만 가능하다.

- 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: $a[0] = 1, a[1] = 1$

모든 네 가지 경우에 대해서, $\min(a[0], a[1])$는 $a[0]$와 $a[1]$를 비트 단위로 AND 연산한 값과 같다. 따라서, 가능한 답 중 하나는 다음 순서로 함수를 호출하여 프로그램을 만드는 것이다.

1. `append_move(1, 0)`, $r[0]$을 $r[1]$에 복사하는 명령을 추가한다.
2. `append_right(1, 1, 1)`, $r[1]$의 모든 비트를 오른쪽으로 1만큼 이동한 다음, 그 결과를 다시 $r[1]$에 저장하는 명령어를 추가한다. 각각의 정수의 길이가 1 비트이니까, 이 명령을 실행하면 $r[1][0]$이 $a[1]$과 같게 된다.
3. `append_and(0, 0, 1)`, $r[0]$과 $r[1]$을 비트 단위로 AND 연산하여 $r[0]$에 저장하는 명령을 추가한다. 이 명령을 실행한 다음, $r[0][0]$는 $r[0][0]$과 $r[1][0]$를 비트 단위로 AND 연산한 결과가 되는데, 이는 우리가 원하는 $a[0]$와 $a[1]$를 비트 단위로 AND 연산한 결과이다.

Example 2

$s = 1,\ n = 2,\ k = 1,\ q = 1000$이라고 하자. 앞 예제와 같이, 이 프로그램에 가능한 입력은 오직 4가지 뿐이다. 모든 4가지 경우에 대해서, $\min(a[0], a[1])$는 $a[0]$과 $a[1]$를 비트 단위로 AND 연산한 결과이며, $\max(a[0], a[1])$는 $a[0]$와 $a[1]$를 비트 단위로 OR 연산한 결과이다. 다음 함수 호출을 통해서 가능한 답 하나를 만들 수 있다.

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)`

이 명령들을 다 실행하고 나면, $c[0] = r[0][0]$의 값은 $\min(a[0], a[1])$이고 $c[1] = r[0][1]$의 값은 $\max(a[0], a[1])$이 되는데, 입력을 정렬한 결과가 된다.

Constraints

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

Subtasks

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

Sample Grader

샘플 그레이더는 다음 양식으로 입력을 받는다.

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

다음 여러 줄에 걸쳐서 테스트케이스가 주어진다. 한 줄마다 테스트케이스 하나씩을 나타낸다. 각 테스트케이스는 다음 양식으로 주어진다.

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

이는 입력이 n 개의 정수인 $a[0], a[1], ..., a[n-1]$이라는 뜻이다. 모든 테스트케이스들이 주어진 다음, -1만 포함하고 있는 한 줄이 주어진다.

샘플 그레이더는 먼저 `construct_instructions(s, n, k, q)`를 호출한다. 만약 이 호출에서 위에서 설명한 제약 조건을 어긴 경우가 발생한다면, 샘플 그레이더는 "Implementation Details"의 마지막에 언급한 에러 메시지 중 하나를 출력하고 종료한다. 그렇지 않으면, 샘플 그레이더는 먼저 `construct_instructions(s, n, k, q)`에 의해 추가된 명령어를 차례대로 출력한다. $store$ 명령에 대해서는, v는 0번 부터 $b - 1$ 번 순서로 출력한다.

다음, 샘플 그레이더는 차례대로 테스트케이스를 처리한다. 각 테스트케이스에 대해서, 만들어진 프로그램을 이용하여 테스트케이스의 입력을 처리한다.

각각 $print(t)$ 명령에 대해서, $d[0], d[1], \ldots, d[n-1]$가 정수의 서열이고, 각각 i에 대해서 ($0 \le i \le n - 1$), $d[i]$는 (명령이 수행되었을 때) 레지스터 t에 저장되어 있는 비트 $i \cdot k$부터

$(i+1) \cdot k - 1$까지의 정수값이다. 그레이더는 다음 양식으로 이 정보를 출력한다. `register` t: $d[0]$ $d[1]$ $\ldots$ $d[n-1]$.

일단 모든 명령을 수행한 다음, 샘플 그레이더는 프로그램의 결과를 출력한다.

만약 $s = 0$이면, 샘플 그레이더는 다음 양식으로 각각의 테스트 케이스의 결과를 출력한다.

- $c[0]$.

만약 $s = 1$이면, 샘플 그레이더는 다음 양식으로 각각의 테스트 케이스의 결과를 출력한다.

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

모든 테스트 케이스를 처리한 다음 그레이더는 `number of instructions:` X를 출력하는데 X는 당신의 프로그램의 명령어 수이다.