Registri di bit

L'ingegnere Christopher sta lavorando ad una nuova architettura per il processore del suo computer.

Il processore possiede m celle di memoria di b bit ciascuna ($m = 100$ e $b = 2000$), chiamate **registri** e denotati come $r[0], r[1], \ldots, r[m-1]$. Ogni registro è un array di b bit, numerati da 0 (il bit più a destra) a $b-1$ (il bit più a sinistra). Per ogni i $(0 \leq i \leq m-1)$ e ogni j $(0 \leq j \leq b-1)$, denotiamo il j-esimo bit del registro i con $r[i][j]$.

Data una sequenza di bit $d_0, d_1, \ldots, d_{l-1}$ (di lunghezza arbitraria l), il **valore intero** della sequenza è uguale a $2^0 \cdot d_0 + 2^1 \cdot d_1 + \ldots + 2^{l-1} \cdot d_{l-1}$. Diciamo che il **valore intero memorizzato nel registro** i è il valore intero della sua sequenza di bit, cioè $2^0 \cdot r[i][0] + 2^1 \cdot r[i][1] + \ldots + 2^{b-1} \cdot r[i][b-1]$.

Il processore supporta 9 tipi di **istruzioni** che possono essere usate per modificare i bit nei registri. Ogni istruzione opera su uno o più registri e memorizza l'output in uno dei registri. Con la notazione $x := y$ indichiamo l'operazione che modifica il valore di x per farlo diventare y. Le operazioni supportate sono le seguenti:

- $move(t, y)$: copia l'array di bit dal registro y al registro t. Per ogni j $(0 \leq j \leq b-1)$, imposta $r[t][j] := r[y][j]$.
- $store(t, v)$: imposta il valore del registro t al valore v, dove v è un array di b bit. Per ogni j $(0 \leq j \leq b-1)$, imposta $r[t][j] := v[j]$.
- $and(t, x, y)$: esegui l'*and* bit-a-bit dei registri x e y, e scrivi il risultato nel registro t. Per ogni j $(0 \leq j \leq b-1)$, imposta $r[t][j] := 1$ se **entrambi** $r[x][j]$ e $r[y][j]$ sono 1, altrimenti $r[t][j] := 0$.
- $or(t, x, y)$: esegui l'*or* bit-a-bit dei registri x e y, e scrivi il risultato nel registro t. Per ogni j $(0 \leq j \leq b-1)$, imposta $r[t][j] := 1$ se **almeno uno** tra $r[x][j]$ e $r[y][j]$ è 1, altrimenti $r[t][j] := 0$.
- $xor(t, x, y)$: esegui lo *xor* bit-a-bit dei registri x e y, e scrivi il risultato nel registro t. Per ogni j $(0 \leq j \leq b-1)$, imposta $r[t][j] := 1$ se **esattamente uno** tra $r[x][j]$ e $r[y][j]$ è 1, altrimenti $r[t][j] := 0$.
- $not(t, x)$: esegui il *not* bit-a-bit del registro x, e scrivi il risultato nel registro t. Per ogni j $(0 \leq j \leq b-1)$, imposta $r[t][j] := 1 - r[x][j]$.
- $left(t, x, p)$: trasla tutti i bit del registro x verso sinistra di p posizioni ($0 \leq p \leq b$), e scrivi il risultato nel registro t. Più precisamente, per ogni j $(0 \leq j \leq b-1)$, $r[t][j] := r[x][j-p]$ se $j \geq p$, e $r[t][j] := 0$ altrimenti.

- $right(t, x, p)$: trasla tutti i bit del registro x verso destra di p posizioni ($0 \leq p \leq b$), e scrivi il risultato nel registro t. Più precisamente, per ogni j $(0 \leq j \leq b-1)$, $r[t][j] := r[x][j+p]$ se $j \leq b-1-p$, e $r[t][j] := 0$ altrimenti.
- $add(t, x, y)$: somma i valori interi nei registri x e y (modulo 2^b), e scrivi il risultato in t. Più precisamente, chiamiamo X il valore intero nel registro x, e Y quello in y; nel registro t viene scritto il valore $X+Y$ se $X+Y < 2^b$, altrimenti $X+Y-2^b$.

Christopher vuole risolvere con il suo nuovo processore due diversi problemi (indicati dall'intero s), e per ciascuno di questi devi produrre un **programma** (una sequenza di istruzioni).

L'**input** del programma è una sequenza di n interi $a[0], a[1], \ldots, a[n-1]$, ciascuno di k bit ($a[i] < 2^k$). All'inizio del programma, gli interi in input sono memorizzati sequenzialmente nel registro 0, cioè, per ogni i $(0 \leq i \leq n-1)$, il valore intero della sequenza di k bit $r[0][i \cdot k], r[0][i \cdot k + 1], \ldots, r[0][(i+1) \cdot k - 1]$ è uguale ad $a[i]$. Nota che $n \cdot k \leq b$. Tutti gli altri bit nel registro 0 (i bit da $n \cdot k$ a $b-1$ inclusi), e tutti i bit degli altri registri, sono inizializzati a zero.

Eseguire un programma consiste nell'eseguire in ordine tutte le sue istruzioni. Dopo aver eseguito l'ultima istruzione, l'**output** del programma è valutato rispetto ai bit del registro 0. In particolare, l'output è una sequenza di n interi $c[0], c[1], \ldots, c[n-1]$, dove per ogni i ($0 \leq i \leq n-1$), $c[i]$ è il valore intero della sequenza di k bit da $i \cdot k$ a $(i+1) \cdot k - 1$ (inclusi), nel registro 0. Nota che i bit rimanenti nel registro 0 (con indice almeno $n \cdot k$) e tutti i bit degli altri registri verranno ignorati.

- Il primo problema $(s = 0)$ è quello di trovare il minimo intero all'interno della sequenza in input $a[0], a[1], \ldots, a[n-1]$. In particolare, $c[0]$ deve essere il minimo tra $a[0], a[1], \ldots, a[n-1]$, mentre i valori di $c[1], c[2], \ldots, c[n-1]$ possono essere arbitrari.
- Il secondo problema $(s = 1)$ è quello di ordinare gli interi $a[0], a[1], \ldots, a[n-1]$ in ordine non decrescente. In particolare, $c[0]$ è l'intero più piccolo in input, mentre $c[n-1]$ è il più grande.

Fornisci a Christopher due programmi (di al più q istruzioni ciascuno), che risolvono questi problemi.

Note di implementazione

Devi implementare la seguente funzione:

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

- s: il tipo di problema.
- n: il numero di interi in input.
- k: il numero di bit di ciascun valore in input.
- q: il numero massimo di istruzioni ammesse.
- Questa funzione è chiamata esattamente una volta, e deve produrre una sequenza di istruzioni per risolvere il problema s.

Questa funzione dovrà chiamare una o più volte le seguenti funzioni per costruire una sequenza di istruzioni:

```
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)
```

- Ciascuna di queste funzioni aggiunge la corrispondente istruzione al programma.
- Per tutte le istruzioni interessate, t, x, y deve essere almeno 0 e al più $m - 1$.
- Per tutte le istruzioni interessate, t, x, y non devono essere necessariamente distinti.
- Per le istruzioni $left$ e $right$, p deve essere almeno 0 e al più b.
- Per l'istruzione $store$, la lunghezza di v deve essere b.

Puoi anche chiamare la seguente funzione per testare più agevolmente la tua soluzione:

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

- Ogni chiamata a questa funzione verrà ignorata durante la valutazione della soluzione.
- Nel grader di esempio questa funzione aggiunge un'istruzione $print(t)$ al programma.
- Quando il grader di esempio incontra una $print(t)$, stamperà n interi di k bit, formati dai primi $n \cdot k$ bit del registro t.
- t deve essere compreso tra 0 e $m - 1$ inclusi.
- Le istruzioni $print$ non vengono contate nel totale delle istruzioni usate.

La valutazione della soluzione può risultare in uno dei seguenti messaggi di errore:

- `Invalid index`: l'indice di un registro usato come parametro t, x o y non è valido (eventualmente negativo).
- `Value to store is not b bits long`: la lunghezza di v passata a `append_store` non è uguale a b.
- `Invalid shift value`: il valore di p passato a `append_left` o `append_right` non è tra 0 e b inclusi.
- `Too many instructions`: il programma prodotto contiene più di q istruzioni.

Esempi

Esempio 1

Supponi che $s = 0,\ n = 2,\ k = 1,\ q = 1000$. Sono presenti due interi in input, $a[0]$ e $a[1]$, ciascuno lungo $k = 1$ bit. All'inizio dell'esecuzione, $r[0][0] = a[0]$ e $r[0][1] = a[1]$, mentre tutti gli altri bit nel processore sono 0. Alla fine dell'esecuzione, deve valere $c[0] = \min(a[0], a[1])$.

Sono possibili solo 4 input al programma:

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

Possiamo osservare che, per tutti i 4 casi, $\min(a[0], a[1])$ è uguale all'*and* bit-a-bit di $a[0]$ e $a[1]$. Quindi una possibile soluzione è costruire un programma nel seguente modo:

1. `append_move(1, 0)`, che aggiunge un'istruzione che copia $r[0]$ in $r[1]$.
2. `append_right(1, 1, 1)`, che aggiunge un'istruzione che trasla tutti i bit di $r[1]$ a destra di 1, e scrive il risulato su $r[1]$. Visto che gli interi in input sono lunghi 1 bit, $r[1][0]$ sarà uguale a $a[1]$.
3. `append_and(0, 0, 1)`, che aggiunge un'istruzione che effettua l'*and* bit-a-bit di $r[0]$ e $r[1]$, e scrive il risultato in $r[0]$. Dopo questa istruzione, $r[0][0]$ è l'*and* bit-a-bit di $r[0][0]$ e $r[1][0]$, che corrisponde all'*and* bit-a-bit tra $a[0]$ e $a[1]$.

Esempio 2

Supponiamo che $s = 1,\ n = 2,\ k = 1,\ q = 1000$. Come nell'esempio precedente, ci sono solo 4 input possibili per un tale programma. In tutti e 4 i casi, $\min(a[0], a[1])$ è l'*and bit-a-bit* di $a[0]$ e $a[1]$, mentre $\max(a[0], a[1])$ è l'*or bit-a-bit* di $a[0]$ e $a[1]$. Una possibile soluzione è effettuare le seguenti chiamate:

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)`

Dopo aver eseguito queste istruzioni, $c[0] = r[0][0]$ conterrà $\min(a[0], a[1])$, mentre $c[1] = r[0][1]$ conterrà $\max(a[0], a[1])$, ordinando il vettore.

Assunzioni

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

Subtask

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

Grader di esempio

Il grader di esempio legge l'input nel seguente formato:

- riga 1 : s n k q

Seguono alcune righe, dove ciascuna descrive un caso di test nel seguente formato:

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

rappresentanti gli interi in input. Dopo la descrizione dei casi di test, deve seguire un'ultima riga con il solo intero -1.

Il grader di esempio chiama innanzitutto `construct_instructions(s, n, k, q)`. Se questa chiamata viola alcune delle assunzioni nel testo del problema, il grader stampa uno dei messaggi di errore elencati alla fine della sezione "Note di implementazione" e termina. Altrimenti, il grader di esempio innanzitutto stampa una descrizione delle istruzioni che sono state aggiunte al programma (in ordine); stampando anche il valore di v per le istruzioni $store$.

Poi, il grader di esempio processa ciascun caso di test in ordine, eseguendo il programma costruito sul corrispondente input. Per ogni istruzione $print(t)$, il valore corrente del registro t viene stampato come una sequenza di n interi di k bit, nel formato: `register` t:
$d[0]$ $d[1]$ $\ldots$ $d[n-1]$.

Dopo l'esecuzione di tutte le istruzioni, il grader di esempio stampa l'output finale del programma.

Se $s = 0$, l'output di ogni caso di test è nel formato:

- $c[0]$.

Se $s = 1$, l'output di ogni caso di test è nel formato:

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

Dopo l'esecuzione di tutti i test, il grader di esempio stampa `number of instructions:` X dove X è il numero di istruzioni nel tuo programma.