Bit Shift Registers

Christopher é um engenheiro que está a trabalhar num novo tipo de processador.

Um processador tem acesso a m diferentes células de memória, cada uma com b bits (onde $m = 100$ e $b = 2000$), que são chamadas de **registos** e são numeradas de 0 a $m - 1$. Denotamos os registos por $r[0], r[1], \ldots, r[m-1]$. Cada registo é um array de b bits, numerados de 0 (o bit mais à direita) a $b - 1$ (o bit mais à esquerda). Para cada i $(0 \leq i \leq m - 1)$ e cada j $(0 \leq j \leq b - 1)$, denotamos o j-ésimo bit do registo i por $r[i][j]$.

Para qualquer sequência de bits $d_0, d_1, \ldots, d_{l-1}$ (com tamanho arbitrário l), o **valor inteiro** da sequência é igual a $2^0 \cdot d_0 + 2^1 \cdot d_1 + \ldots + 2^{l-1} \cdot d_{l-1}$. Dizemos que o **valor inteiro armazenado no registo** i é o valor inteiro da sua sequência de bits, ou seja, é $2^0 \cdot r[i][0] + 2^1 \cdot r[i][1] + \ldots + 2^{b-1} \cdot r[i][b-1]$.

O processador tem 9 tipos de **instruções** que podem ser usadas para modificar os bits dos registos. Cada instrução opera num ou mais registos e guarda o output num dos registos. No que se segue, usamos $x := y$ para indicar uma operação de mudar o valor de x tal que passa a ser igual a y. As operações feitas por cada tipo de instrução são descritas a seguir.

- $move(t, y)$: Copia o array de bits do registo y para o registo t. Para cada j $(0 \leq j \leq b - 1)$ executa $r[t][j] := r[y][j]$.
- $store(t, v)$: Faz com que o registo t seja igual a v, onde v é um array de b bits. Para cada j $(0 \leq j \leq b - 1)$ executa $r[t][j] := v[j]$.
- $and(t, x, y)$: Faz um AND bit a bit entre os registos x e y, e armazena o resultado no registo t. Para cada j $(0 \leq j \leq b - 1)$ executa $r[t][j] := 1$ se **ambos** $r[x][j]$ e $r[y][j]$ são 1, ou executa $r[t][j] := 0$ no caso contrário.
- $or(t, x, y)$: Faz um OR bit a bit entre os registos x e y, e armazena o resultado no registo t. Para cada j $(0 \leq j \leq b - 1)$ executa $r[t][j] := 1$ se **pelo menos um** entre $r[x][j]$ e $r[y][j]$ for 1, ou executa $r[t][j] := 0$ no caso contrário.
- $xor(t, x, y)$: Faz um XOR bit a bit entre os registos x e y, e armazena o resultado no registo t. Para cada j $(0 \leq j \leq b - 1)$ executa $r[t][j] := 1$ se **exatamente um** entre $r[x][j]$ e $r[y][j]$ for 1, ou executa $r[t][j] := 0$ no caso contrário.
- $not(t, x)$: Faz um NOT bit a bit do registo x, e armazena o resultado no registo t. Para cada j $(0 \leq j \leq b - 1)$ executa $r[t][j] := 1 - r[x][j]$.
- $left(t, x, p)$: Faz um "shift" de p posições para a esquerda a todos os bits no registo x e armazena o resultado no registo t. O resultado do shift é um array v consistindo em b bits.

Para cada j $(0 \leq j \leq b-1)$, $v[j] = r[x][j+p]$ se $j \leq b-1-p$, e $v[j] = 0$ no caso contrário. Para cada j $(0 \leq j \leq b-1)$, executa $r[t][j] := v[j]$.

- $right(t, x, p)$: Faz um "shift" de p posições para a direita a todos os bits no registo x e armazena o resultado no registo t. O resultado do shift é um array v consistindo em b bits. Para cada j $(0 \leq j \leq b-1)$, $v[j] = r[x][j+p]$ se $j \leq b-1-p$, e $v[j] = 0$ no caso contrário. Para cada j $(0 \leq j \leq b-1)$, executa $r[t][j] := v[j]$.

- $add(t, x, y)$: Adiciona os valores inteiros armazenados nos registos x e y, e armazena o resultado no registo t. A soma é feito módulo 2^b. Formalmente, seja X o inteiro armazenado x e Y o inteiro armazenado no registo y antes da operação. Seja T o valor inteiro armazenado no registo t depois da operação. Se $X + Y < 2^b$, is bits de t devem ser mudados de tal maneira que $T = X + Y$. Caso contrário os bits de t devem ser de tal maneira que $T = X + Y - 2^b$.

Christopher quer que resolvas dois tipos de tarefas usando o novo processador. O tipo de tarefa é indicado por um inteiro s. Para ambos os tipos de tarefas, deves produzir um **programa**, isto é, uma sequência de instruções como atrás foram definidas.

O **input** para o programa consiste em n inteiros $a[0], a[1], \ldots, a[n-1]$, cada um tendo k bits, ou seja, $a[i] < 2^k$ ($0 \leq i \leq n-1$). Antes do programa ser executado, todos os números de input são armazenados sequencialmente no registo 0, tal que para cada i $(0 \leq i \leq n-1)$ o valor da sequência de k bits $r[0][i \cdot k], r[0][i \cdot k + 1], \ldots, r[0][(i+1) \cdot k - 1]$ é igual a $a[i]$. Nota que $n \cdot k \leq b$. Todos os outros bits do registo 0 (isto é, aqueles com índices entre $n \cdot k$ e $b-1$, inclusive) e todos os bits de todos os outros registos são inicializados a 0.

Executar um programa consiste em executar as suas instruções por ordem. Depois da última instrução ser executada, o **output** de programa é calculado com base no valor final dos bits no regisso 0. Especificamente, o output é uma sequência de n inteiros $c[0], c[1], \ldots, c[n-1]$, onde para cada i ($0 \leq i \leq n-1$), $c[i]$ é o valor inteiro da sequência dos bits $i \cdot k$ to $(i+1) \cdot k - 1$ do registo 0. Nota que depois de executar o programa, todos os outros bits do registo 0 (com indíces maiores ou iguais a $n \cdot k$) e todos os bits dos outros registos podem ser arbitrários.

- A primeira tarefa $(s = 0)$ é descobrir o menor inteiro entre os inteiros de input $a[0], a[1], \ldots, a[n-1]$. Especificamente, $c[0]$ deve ser o mínimo entre $a[0], a[1], \ldots, a[n-1]$. Os valores de $c[1], c[2], \ldots, c[n-1]$ podem ser arbitrários.

- A segunda tarefa $(s = 1)$ é ordenar os inteiros de input $a[0], a[1], \ldots, a[n-1]$ por ordem não decrescente. Especificamente, para cada i ($0 \leq i \leq n-1$), $c[i]$ deve ser igual ao $1+i$-ésimo inteiro mais pequeno entre $a[0], a[1], \ldots, a[n-1]$ (isto é, $c[0]$ é o mais pequeno inteiro entre todos os inteiros do input).

Deves criar programas para o Christopher, cada um deles consistindo num máximo de q instruções, que resolvam estas tarefas.

Detalhes de Implementação

Deves implementação a seguinte função:

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

- s: ttipo de tarefa.
- n: número de inteiros no input.
- k: número de bits em cada inteiro do input.
- q: número máximo de instruções permitido.
- Esta função é chamada exatamente uma vez e deve construir uma sequência de instruções para para fazer a tarefa pedida.

Esta função deve chamar uma ou mais das seguintes funções para construir a sequência de instruções:

```
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 função adiciona uma instrução $move(t, y)$ $store(t, v)$, $and(t, x, y)$, $or(t, x, y)$, $xor(t, x, y)$, $not(t, x)$, $left(t, x, p)$, $right(t, x, p)$ ou $add(t, x, y)$ ao programa, respetivamente.
- Para todas as instruções relevantes, t, x, y deve ser no mínimo 0 e no máximo $m - 1$.
- Para todas as instruções relevantes, t, x, y não são necessariamente distintos (par a par).
- Para as instruções $left$ e $right$, p deve ser no mínimo 0 e no máximo b.
- Para as instruções $store$, o tamanho de v deve ser b.

Podes também chamar a seguinte função para ajudar a testar a tua solução:

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

- Qualquer chamada a esta função será ignorada durante a avaliação da tua solução.
- No avaliador exemplo, esta função adiciona uma operação $print(t)$ ao programa.
- Quando o avaliador exemplo encontra uma instrução $print(t)$ durante a execução de um programa, ele imprime n inteiros de k bits formados pelos primeiros $n \cdot k$ bits do registo t (ver a secção do avaliador exemplo para mais detalhes).
- t tem de satisfazer $0 \leq t \leq m - 1$.
- Qualquer chamada a este procedimento não adiciona nada ao número de instruções construídas.

Depois de adicionar a última instrução, `construct_instructions` deve sair (da função). O programa será então avaliado num conjunto de casos de teste, cada um especificando um input

consistindo em n inteiros de k bits $a[0], a[1], \ldots, a[n-1]$. A tua solução passa um caso de teste se o output do programa $c[0], c[1], \ldots, c[n-1]$ para o input dado satisfizer as seguintes condições:

- Se $s = 0$, $c[0]$ deve ser o menor valor entre $a[0], a[1], \ldots, a[n-1]$.
- Se $s = 1$, para cada i ($0 \leq i \leq n-1$), $c[i]$ deve ser o $1+i$-ésimo inteiro mais pequeno entre $a[0], a[1], \ldots, a[n-1]$.

A avaliação da tua solução pode resultar numa das seguintes mensagens de erro:

- `Invalid index`: um índice de registo incorreto (possivelmente negativo) foi dado como parâmetro t, x ou y numa das chamadas a funções.
- `Value to store is not b bits long`: o tamanho de um v dado a `append_store` não é igual a b.
- `Invalid shift value`: o valor p dado a `append_left` ou `append_right` não está entre 0 e b inclusive.
- `Too many instructions`: a tua função tentou adicionar mais do que q instrucões.

Exemplos

Exemplo 1

Supõe que $s = 0$, $n = 2$, $k = 1$, $q = 1000$. Existem dois inteiros de input $a[0]$ e $a[1]$, cada um tendo $k = 1$ bit. Antes do programa ser executado, $r[0][0] = a[0]$ e $r[0][1] = a[1]$. Todos os outros bits do processador estão colocados a 0. Depois de todas as instruões do programa serem executadas, temos de ter $c[0] = r[0][0] = \min(a[0], a[1])$, que é o mínimo entre $a[0]$ e $a[1]$.

Existem apenas 4 possíveis inputs para o 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 ver que para todos os 4 casos, $\min(a[0], a[1])$ é igual ao AND entre $a[0]$ e $a[1]$. Deste modo, uma possível solução seria construir um programa fazendo as seguintes chamadas:

1. `append_move(1, 0)`, que adiciona uma instrução para copiar $r[0]$ para $r[1]$.
2. `append_right(1, 1, 1)`, que adiciona uma instrução que pega em todos os bits de $r[1]$, aplica-lhes um shif para a direita de 1 posição, e depois armazena o resultado novamente em $r[1]$. Como cada inteiro tem 1 bit de comprimento, isto resulta em $r[1][0]$ igual $a[1]$.
3. `append_and(0, 0, 1)`, que adiciona uma instrução para fazer um AND a $r[0]$ e $r[1]$, e depois armazena o resultado em $r[0]$. Depois desta instrução ser executada, $r[0][0]$ passa a ser o AND de $r[0][0]$ e $r[1][0]$, que é igual ao AND entre $a[0]$ e $a[1]$, como desejado.

Exemplo 2

Supõe que $s = 1,\ n = 2,\ k = 1,\ q = 1000$. Tal como no exemplo anterior, só existem 4 possíveis inputs para o programa. Para todos os 4 casos, $\min(a[0], a[1])$ é o AND entre $a[0]$ e $a[1]$, e $\max(a[0], a[1])$ é o OR entre $a[0]$ e $a[1]$. Uma possível solução é fazer as seguintes chamadas:

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)`

Depois de executar estas instruções, $c[0] = r[0][0]$ contém $\min(a[0], a[1])$, e $c[1] = r[0][1]$ contém $\max(a[0], a[1])$, o que ordena o input.

Restrições

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

Subtarefas

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

Avaliador Exemplo

O avaliador exemplo lê o input no seguinte formato:

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

Isto é seguido de um certo número de linhas, cada uma descrevendo um único caso de teste. Cada caso de teste é dado no seguinte formato:

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

e descreve o caso de teste cujo input consiste em n inteiros $a[0], a[1], ..., a[n-1]$. A descrição de todos os casos de teste é seguida de uma linha contendo unicamente -1.

O avaliador exemplo chama primeiro `construct_instructions(s, n, k, q)`. Se esta chamada violar alguma das restrições descritas no enunciado, o avaliador exemplo escreve uma das mensagens de erro listadas no final da secção de detalhes de implementação e termina. Caso contrário, o avaliador exemplo primeiro escreve cada instrução adicionada por `construct_instructions(s, n, k, q)`, por ordem. Para as instruções $store,\ v$ é escrito do indíce 0 para o índice $b-1$.

Depois, o avaliador processa os casos de teste por ordem. Para cada caso de teste, o programa construído é executado com o input desse caso de teste.

Para cada instrução $print(t)$, seja $d[0], d[1], \ldots, d[n-1]$ uma sequência de inteiros, tal que para cada i ($0 \leq i \leq n-1$), $d[i]$ é o valor inteiro da sequência de bits $i \cdot k$ a $(i+1) \cdot k - 1$ do registo t (quando a instrução é executada). O avaliador escreve a sequência no seguinte formato: `register` t: $d[0]\ d[1]\ \ldots\ d[n-1]$.

Uma vez que todas as instruções tenham sido executadas, o avaliador exemplo escreve o output do programa.

Se $s = 0$, o output do avaliador exemplo para cada caso de teste está no seguinte formato:

- $c[0]$.

Se $s = 1$, o output do avaliador exemplo para cada caso de teste está no seguinte formato:

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

Depois de executar todos os casos de teste, o avaliador escreve `number of instructions:` X onde X é o número de instruções do teu programa.