Schieberegister

Christopher der Erfinder arbeitet an einer neuen Art von Computerprozessoren.

Der Prozessor hat Zugriff auf m verschiedene Speicherzellen aus b Bits (wobei $m = 100$ und $b = 2000$), sogenannte **Register**, welche nummeriert sind von 0 bis $m-1$. Wir bezeichnen die Register als $r[0], r[1], \ldots, r[m-1]$. Jedes Register ist ein Array von b Bits, nummeriert von 0 (das Bit ganz rechts) bis $b-1$ (das Bit ganz links). Für i $(0 \leq i \leq m-1)$ und j $(0 \leq j \leq b-1)$ bezeichnen wir das j-te Bit von Register i mit $r[i][j]$.

Für eine Bitsequenz $d_0, d_1, \ldots, d_{l-1}$ (von beliebiger Länge l) ist der **Zahlenwert** dieser Sequenz definiert als $2^0 \cdot d_0 + 2^1 \cdot d_1 + \ldots + 2^{l-1} \cdot d_{l-1}$. Der **Zahlenwert eines Registers** i ist der Zahlenwert seiner Bitsequenz, d.h. $2^0 \cdot r[i][0] + 2^1 \cdot r[i][1] + \ldots + 2^{b-1} \cdot r[i][b-1]$.

Der Prozessor kennt 9 verschiedene **Instruktionen**, mit denen sich die Bits in den Registern verändern lassen. Jede Instruktion nimmt ein odere mehrere Register und speichert das Resultat in einem weiteren Register ab. Im Folgenden meinen wir mit $x := y$, dass der Wert von x auf den Wert von y gesetzt wird. Es folgt eine Definition aller Instruktionen.

- $move(t, y)$: Kopiere das Bit-Array in Register y nach Register t. Für jedes j $(0 \leq j \leq b-1)$, setze $r[t][j] := r[y][j]$.
- $store(t, v)$: Setze Register t auf den Wert v, wobei v ein Bit-Array der Länge b ist. Für jedes j $(0 \leq j \leq b-1)$, setze $r[t][j] := v[j]$.
- $and(t, x, y)$: Berechne das bitweise AND der Register x und y und speichere das Resultat in Register t ab. Für jedes j $(0 \leq j \leq b-1)$, setze $r[t][j] := 1$, falls **sowohl** $r[x][j]$ **als auch** $r[y][j]$ den Wert 1 haben, und ansonsten setze $r[t][j] := 0$.
- $or(t, x, y)$: Berechne das bitweise OR der Register x und y und speichere das Resultat in Register t ab. Für jedes j $(0 \leq j \leq b-1)$, setze $r[t][j] := 1$ falls **mindestens eines** der Bits $r[x][j]$ und $r[y][j]$ den Wert 1 haben, und ansonsten setze $r[t][j] := 0$.
- $xor(t, x, y)$: Berechne das bitweise XOR der Register x und y und speichere das Resultat in Register t ab. Für jedes j $(0 \leq j \leq b-1)$, setze $r[t][j] := 1$ falls **genau einer** der Bits $r[x][j]$ und $r[y][j]$ den Wert 1 haben, und ansonsten setze $r[t][j] := 0$.
- $not(t, x)$: Berechne das bitweise NOT von Register x und speichere das Resultat in Register t. Für jedes j $(0 \leq j \leq b-1)$, setze $r[t][j] := 1 - r[x][j]$.
- $left(t, x, p)$: Verschiebe die Bits in Register x um p nach links, und speichere das Resultat in Register t ab. Das Ergebnis einer Verschiebung um p nach links von Register x ist ein Bit-Array v von Länge b. Für jedes j $(0 \leq j \leq b-1)$ sei $v[j] = r[x][j-p]$, falls $j \geq p$, und sonst $v[j] = 0$. Für jedes j $(0 \leq j \leq b-1)$, setze $r[t][j] := v[j]$.

- $right(t, x, p)$: Verschiebe die Bits in Register x um p nach rechts und speichere das Resultat in t ab. Das Ergebnis einer Verschiebung um p nach rechts von Register x ist ein Bit-Array v der Länge b. Für jedes j $(0 \leq j \leq b-1)$, sei $v[j] = r[x][j+p]$ falls $j \leq b-1-p$ und sonst $v[j] = 0$. Für jedes j $(0 \leq j \leq b-1)$, setze $r[t][j] := v[j]$.
- $add(t, x, y)$: Addiere die Zahlenwerte von Register x und Register y und speichere das Resultat in Register t ab. Die Addition wird modulo 2^b berechnet. Sei X der Zahlenwert in Register x und sei Y der Zahlenwert in y vor der Operation. Sei T der Zahlenwert in Register t nach der Operation. Falls $X + Y < 2^b$, setze die Bits von t so, dass $T = X + Y$ gilt. Ansonsten, setze die Bits von t so, dass $T = X + Y - 2^b$ gilt.

Christopher möchte, dass du zwei Aufgabentypen mit dem neuen Prozessor löst. Der Aufgabentyp ist gegeben durch die Zahl s. Für beide Aufgabentypen sollst du ein **Programm** generieren. Ein Programm ist eine Liste an Instruktionen, die oben beschrieben wurden.

Die **Eingabe** deines Programms besteht aus n Ganzzahlen $a[0], a[1], \ldots, a[n-1]$, jeweils bestehend aus k Bits, d.h. $a[i] < 2^k$ ($0 \leq i \leq n-1$). Bevor dein Programm ausgeführt wird, werden alle Zahlen in der Eingabe hintereinander in Register 0 abgelegt, so dass für jedes i $(0 \leq i \leq n-1)$ der Zahlenwert der Bitsequenz $r[0][i \cdot k], r[0][i \cdot k + 1], \ldots, r[0][(i+1) \cdot k - 1]$ gleich $a[i]$ ist. Bemerke, dass $n \cdot k \leq b$. Alle anderen Bits in Register 0 (d.h. alle mit Indizes zwischen $n \cdot k$ und einschließlich $b-1$) und alle Bits der restlichen Registern werden mit 0 initialisiert.

Ein Programm auszuführen, bedeutet, seine Instruktionen in der gegebenen Reihenfolge abzuarbeiten. Nachdem die letzte Instruktion ausgeführt wurde, wird die **Ausgabe** deines Programms von dem Wert der Bits in Register 0 bestimmt. Konkret ist die Ausgabe eine Liste von n Ganzzahlen $c[0], c[1], \ldots, c[n-1]$, wobei für jedes i ($0 \leq i \leq n-1$), $c[i]$ dem Zahlenwert der Bitsequenz von $i \cdot k$ bis $(i+1) \cdot k - 1$ von Register 0 entspricht. Bemerke, dass alle anderen Bits von Register 0 (diejenigen mit Indizes ab $n \cdot k$) und alle anderen Bits der restlichen Register beliebig sein dürfen.

- Der erste Aufgabentyp $(s = 0)$ ist, die kleinste Zahl unter den Eingabezahlen $a[0], a[1], \ldots, a[n-1]$ zu finden. Formal soll $c[0]$ das Minimum von $a[0], a[1], \ldots, a[n-1]$ sein. Die Werte von $c[1], c[2], \ldots, c[n-1]$ dürfen beliebig sein.
- Der zweite Aufgabentyp $(s = 1)$ ist, die Eingabezahlen $a[0], a[1], \ldots, a[n-1]$ in aufsteigender Reihenfolge anzuordnen. Formal soll $c[0], c[1], \ldots, c[n-1]$ eine Permutation der Eingabewerte $a[0], a[1], \ldots, a[n-1]$ sein, für die gilt, dass $c[0] \leq c[1] \leq \cdots \leq c[n-1]$.

Hilf Christopher, jeweils ein Programm zu finden, welches den entsprechenden Aufgabentyp lösen kann und höchstens q Instruktionen lang ist.

Implementierungsdetails

Du sollst die folgende Funktion implementieren:

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

- s: Aufgabentyp.
- n: Anzahl an Zahlen in der Eingabe.
- k: Anzahl an Bits, die jede Zahl in der Eingabe hat.
- q: Maximale Anzahl an generierten Instruktionen.
- Diese Funktion wird genau einmal aufgerufen und sollte eine Sequenz von Instruktionen generieren, welche den gefragten Aufgabentyp löst.

Die Funktion sollte ein oder mehrere Male eine der folgenden Funktionen aufrufen, um eine Sequenz von Instuktionen zu generiern:

```
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)
```

- Jeder Funktionsaufruf hängt die entsprechende Instruktion dem Programm an: $move(t, y)$, $store(t, v)$, $and(t, x, y)$, $or(t, x, y)$, $xor(t, x, y)$, $not(t, x)$, $left(t, x, p)$, $right(t, x, p)$ oder $add(t, x, y)$.
- Für alle passenden Instruktionen sollen t, x, y mindestens 0 und höchstens $m - 1$ sein.
- Für alle passenden Instruktionen müssen t, x, y nicht notwendigerweise paarweise verschieden sein.
- Bei den Instruktionen $left$ und $right$ soll p mindestens 0 und höchstens b sein.
- Bei der $store$-Instruktion muss die Länge von v genau b sein.

Du kannst ausserdem die folgende Funktion aufrufen, die dir helfen kann, deine Lösung zu testen:

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

- Jeder Aufruf dieser Funktion wird während der Bewertung deiner Lösung ignoriert.
- Im Beispiel-Grader fügt diese Funktion eine $print(t)$-Instruktion dem Programm an.
- Wenn der Beispiel-Grader während der Ausführung eines Programs eine $print(t)$-Instruktion antrifft, gibt er n k-Bit Zahlen aus, welches die ersten $n \cdot k$ Bits von Register t sind. (Siehe Abschnitt "Beispiel-Grader" für Details).
- t muss $0 \leq t \leq m - 1$ erfüllen.
- Ein Aufruf dieser Funktion wird nicht zur Anzahl an generierten Instruktionen dazugezählt.

Nachdem das Generieren der Instruktionen abgeschlossen ist, soll sich die `construct_instructions`-Funktion beenden. Das generierte Program anschliessend anhand einer

Reihe von Testfällen bewertet, welche jeweils aus einer Eingabe von n k-Bit Zahlen $a[0], a[1], \ldots, a[n-1]$ bestehen. Deine Lösung löst einen Testfall, sofern für die gegebene Eingabe die Ausgabe des Programs $c[0], c[1], \ldots, c[n-1]$ folgende Bedingungen erfüllt:

- Für $s = 0$ soll $c[0]$ den kleinsten Wert aus $a[0], a[1], \ldots, a[n-1]$ enthalten.
- Für $s = 1$ soll für jedes i ($0 \leq i \leq n-1$), $c[i]$ die $1+i$-th kleinste Zahl aus $a[0], a[1], \ldots, a[n-1]$ enthalten.

Bei der Bewertung deiner Lösung kannst du eine der folgenden Fehlermeldungen erhalten:

- `Invalid index`: Du hast einen ungültigen (möglicherweise negativen) Registerindex als Argument von t, x oder y übergeben, beim Generieren einer Instruktion.
- `Value to store is not b bits long`: Die Länge von v, welches an `append_store` als Argument übergeben wurde, ist nicht b.
- `Invalid shift value`: Der Wert von p, welcher an `append_left` oder `append_right` als Argument übergeben wurde, ist nicht zwischen 0 und (einschließlich) b.
- `Too many instructions`: Das generierte Programm hat mehr als q Instruktionen.

Beispiele

Beispiel 1

Sei $s = 0$, $n = 2$, $k = 1$, $q = 1000$. Es gibt zwei Zahlen in der Eingabe $a[0]$ und $a[1]$; beide haben $k = 1$ Bit. Bevor das Programm ausgeführt wird, ist $r[0][0] = a[0]$ und $r[0][1] = a[1]$. Alle anderen Bits im Prozessor sind auf 0 gesetzt. Nachdem alle Instruktionen im Programm ausfgeführt wurden, muss $c[0] = r[0][0] = \min(a[0], a[1])$ sein, was dem Minimum aus $a[0]$ und $a[1]$ entspricht.

Es gibt nur 4 mögliche Eingaben für das Programm:

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

Wir beobachten, dass in allen 4 Fällen $\min(a[0], a[1])$ den selben Wert hat, wie das bitweise AND von $a[0]$ und $a[1]$. Somit ist es eine mögliche Lösung, ein Programm zu generieren, indem man die folgenden Funktionsaufrufe tätigt:

1. `append_move(1, 0)`, was eine Instruktion hinzufügt, um $r[0]$ nach $r[1]$ zu kopieren.
2. `append_right(1, 1, 1)`, was eine Instruktion hinzufügt, welche alle Bits in $r[1]$ um 1 Bit nach rechts schiebt und das Ergebnis in $r[1]$ speichert. Da alle Zahlen 1 Bit lang sind, entspricht der Wert in $r[1][0]$ dem Wert von $a[1]$.
3. `append_and(0, 0, 1)`, was eine Instruktion hinzufügt, welche das bitweise AND von $r[0]$ und $r[1]$ berechnet und das Ergebnis in $r[0]$ speichert. Nach dem Ausführen dieser Instruktion ist $r[0][0]$ auf den Wert vom bitweisen AND von $r[0][0]$ und $r[1][0]$ gesetzt, was, wie gewünscht, dem Wert vom bitweisen AND von $a[0]$ und $a[1]$ entspricht.

Beispiel 2

Sei $s = 1,\ n = 2,\ k = 1,\ q = 1000$. Wie im vorherigen Beispiel gibt es nur 4 mögliche Eingaben für das Programm. In allen Fällen hat $\min(a[0], a[1])$ den selben Wert wie das bitweise AND von $a[0]$ und $a[1]$, und $\max(a[0], a[1])$ hat den selben Wert wie das bitweise OR von $a[0]$ und $a[1]$. Eine mögliche Lösung könnte folgende Funktionsaufrufe tätigen:

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)`

Nach der Ausführung dieser Instruktionen ist $c[0] = r[0][0]$ auf $\min(a[0], a[1])$ gesetzt, und $c[1] = r[0][1]$ auf $\max(a[0], a[1])$ gesetzt, womit die Eingabe sortiert ist.

Beschränkungen

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

Teilaufgaben

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

Beispiel-Grader

Der Beispiel-Grader liest die Eingabe im folgenden Format:

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

Es folgen mehrere Zeilen, wovon jede einen einzelnen Testfall beschreibt. Jede dieser Zeilen hat das folgende Format:

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

und beschreibt einen Testfall, in welchem die Eingabe aus n Zahlen $a[0], a[1], \ldots, a[n-1]$ besteht. Am Ende der Beschreibung der Testfällen folgt eine einzelne Zeile mit -1.

Der Beispiel-Grader ruft zuerst `construct_instructions(s, n, k, q)` auf. Falls dies eine der obigen Bedingungen verletzt, gibt der Beispiel-Grader eine unter "Implementierungsdetails" (am Ende) beschriebenen Fehlermeldungen aus und beendet die Ausführung des Programms. Ansonsten gibt der Beispiel-Grader zuerst alle von `construct_instructions(s, n, k, q)` generierten Instruktionen aus. Für $store$-Instruktionen wird v von Index 0 bis Index $b-1$ ausgegeben.

Danach arbeitet der Beispiel-Grader die Testfälle der Reihe nach ab. Für jeden Testfall lässt er das generierte Programm mit der Eingabe des Testfalls laufen.

Für jede $print(t)$-Instruktion sei $d[0], d[1], \ldots, d[n-1]$ eine Zahlenfolge, so dass für jedes i ($0 \leq i \leq n-1$), $d[i]$ den Zahlenwert der Bitsequenz $i \cdot k$ bis $(i+1) \cdot k - 1$ von Register t hat (zum Zeitpunkt, in dem $print$ ausgeführt wird). Der Beispiel-Grader gibt die Bitsequenz in folgendem Format aus: `register` t: $d[0]\ d[1]\ \ldots\ d[n-1]$.

Sobald alle Instruktionen ausgeführt wurden, gibt der Beispiel-Grader die Ausgabe des Programmes aus.

Falls $s = 0$, hat die Ausgabe des Beispiel-Graders für jeden Testfall folgendes Format:

- $c[0]$.

Falls $s = 1$, hat die Ausgabe des Beispiel-Graders für jeden Testfall das folgende Format:

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

Nachdem alle Testfälle ausgeführt wurden, gibt der Beispiel-Grader folgendes aus:

`number of instructions:` X

wobei X der Anzahl Instruktionen in deinem Programm enspricht.