Registres à Décalage de Bits

Christopher, ingénieur, est en train de travailler sur un nouveau type de processeur pour ordinateurs.

Ce processeur a accès à m cellules différentes contenant chacune b bits (avec $m = 100$ et $b = 2\,000$) qui sont appelées des **registres** et sont numérotées de 0 à $m-1$. On note les registres $r[0], r[1], \ldots, r[m-1]$. Chaque registre est un tableau de b bits, numérotés de 0 (le bit le plus à droite) à $b-1$ (le bit le plus à gauche). Pour chaque i $(0 \leq i \leq m-1)$ et chaque j $(0 \leq j \leq b-1)$, $r[i][j]$ représente le j-ième bit du registre i.

Étant donnée une séquence de bits $d_0, \ldots, d_{l-1}$ (pour un quelconque l) **l'entier associé** à la séquence est l'entier $2^0 \cdot d_0 + 2^1 \cdot d_1 + \ldots + 2^{l-1} \cdot d_{l-1}$. **L'entier associé au registre** i est l'entier associé à la séquence des bits de ce registre, et donc $2^0 \cdot r[i][0] + 2^1 \cdot r[i][1] + \ldots + 2^{b-1} \cdot r[i][b-1]$.

Le processeur a 9 types **d'instructions** qui peuvent être utilisées pour modifier les bits dans les registres. Chaque instruction opère sur un ou plusieurs registres et stocke son résultat dans un des registres. Dans la suite, on note $x := y$ pour signifier que l'on change la valeur de x pour la rendre égale à y. Les différentes opérations faites par chaque instruction sont décrites ci-dessous.

- $move(t, y)$: Copie le tableau de bits du registre y vers le registre t. Pour chaque j $(0 \leq j \leq b-1)$, on effectue $r[t][j] := r[y][j]$.
- $store(t, v)$: Modifie le registre t de sorte qu'il soit égal à v, où v est un tableau de b bits. Pour chaque j $(0 \leq j \leq b-1)$, on effectue $r[t][j] := v[j]$.
- $and(t, x, y)$: Calcule le ET bit-à-bit des registres x et y, puis stocke le résultat dans t. Pour chaque j $(0 \leq j \leq b-1)$, on effectue $r[t][j] := 1$ si les **deux** $r[x][j]$ et $r[y][j]$ sont 1, et sinon on effectue $r[t][j] := 0$.
- $or(t, x, y)$: Calcule le OU bit-à-bit des registres x et y, puis stocke le résultat dans le registre t. Pour chaque j $(0 \leq j \leq b-1)$, on effectue $r[t][j] := 1$ si **au moins un** de $r[x][j]$ ou de $r[y][j]$ est 1, et sinon on effectue $r[t][j] := 0$.
- $xor(t, x, y)$: Calcule XOR bit-à-bit des registres x and y, puis stocke le résultat dans le registre t. Pour chaque j $(0 \leq j \leq b-1)$, on effectue $r[t][j] := 1$ si **exactement un** de $r[x][j]$ ou de $r[y][j]$ est 1, et sinon on effectue $r[t][j] := 0$.
- $not(t, x)$: Calcule le NON bit à bit du registre x, puis stocke le résultat dans le registre t. Pour chaque j $(0 \leq j \leq b-1)$, on effectue $r[t][j] := 1 - r[x][j]$.
- $left(t, x, p)$: Décale de p bits vers la gauche (left), tous les bits du registre x, puis stocke le résultat dans t. Le résultat du décalage de p bits vers la gauche des bits du registre x est un

tableau v contenant b bits. Pour chaque j $(0 \leq j \leq b-1)$, $v[j] = r[x][j-p]$ si $j \geq p$, et $v[j] = 0$ sinon. Pour chaque j $(0 \leq j \leq b-1)$, on effectue $r[t][j] := v[j]$.

- $right(t, x, p)$: Décale de p bits vers la droite (right) tous les bits du registre x, puis stocke le résultat dans t. Le résultat du décalage de p bits vers la droite des bits du registre x est un tableau v contenant b bits. Pour chaque j $(0 \leq j \leq b-1)$, $v[j] = r[x][j+p]$ si $j \leq b-1-p$, et $v[j] = 0$ sinon. Pour chaque j $(0 \leq j \leq b-1)$, on effectue $r[t][j] := v[j]$.

- $add(t, x, y)$: Ajoute les entiers associés aux registres x et y, puis stocke le résultat dans le registre t. La résultat de l'addition est calculée modulo 2^b. Formellement, on pose X l'entier associé au registre x, et Y l'entier associé au registre y avant l'opération et T l'entier associé à t après l'opération. Si $X + Y < 2^b$, on modifie t de sorte que $T = X + Y$. Sinon, on modifie t de sorte que $T = X + Y - 2^b$.

Christopher aimerait résoudre deux types de tâches avec son nouveau processeur. Le type d'une tâche est représenté par un entier s. Pour les deux types de tâches, vous devez produire un **programme**, c'est à dire une séquence d'instructions telles que définis ci-haut.

L'entrée du programme est constituée de n entiers $a[0], a[1], \ldots, a[n-1]$, chacun ayant k bits, c'est à dire, $a[i] < 2^k$ ($0 \leq i \leq n-1$). Avant que le programme ne soit exécuté, tous les nombres de l'entrée sont stockés de façon séquentielle dans le registre 0, de sorte que pour chaque i $(0 \leq i \leq n-1)$ l'entier associé à la séquence de k bits $r[0][i \cdot k], r[0][i \cdot k + 1], \ldots, r[0][(i+1) \cdot k - 1]$ soit égal à $a[i]$. On remarque que $n \cdot k \leq b$. Tous les bits du registre 0 (et donc ceux aux indices $n \cdot k$ à $b-1$ inclus) et tous les bits des autres registres sont initialisés à 0.

Exécuter un programme consiste à exécuter ses instructions dans l'ordre. Une fois que la dernière instruction du programme est exécutée, la **sortie** du programme est calculée à partir de la valeur finale des bits du registre 0. Plus précisément, la sortie est constituée des n entiers $c[0], c[1], \ldots, c[n-1]$, avec pour chaque i ($0 \leq i \leq n-1$), $c[i]$ est l'entier associé à la séquence constituée des bits $i \cdot k$ à $(i+1) \cdot k - 1$ du registre 0. La valeur des autres bits à la fin de l'exécution du programme ne sont pas prises en compte (ni pour les bits du registre 0 aux indices supérieurs à $n \cdot k$ ni pour les bits des autres registres).

- La première tâche $(s = 0)$ demande de calculer le plus petit entier parmi les entiers $a[0], a[1], \ldots, a[n-1]$. Plus précisément, $c[0]$ doit être le minimum de $a[0], a[1], \ldots, a[n-1]$. Les valeurs $c[1], c[2], \ldots, c[n-1]$ ne sont pas prises en compte.

- La seconde tâche $(s = 1)$ demande de trier les entiers $a[0], a[1], \ldots, a[n-1]$ par ordre croissant. Plus précisément, pour chaque i ($0 \leq i \leq n-1$), $c[i]$ doit être égal au $1+i$-ième plus petit entier parmi $a[0], a[1], \ldots, a[n-1]$ (et donc $c[0]$ est le plus petit entier de l'entrée).

Vous devez fournir à Christopher des programmes constitués d'au plus q instructions chacun, et qui résoud ces tâches.

Détails d'implémentation

Vous devez implémenter la fonction suivante :

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

- s : est le type de tâche.
- n : le nombre d'entiers de l'entrée.
- k : le nombre de bits de chaque entier de l'entrée.
- q : le nombre maximal d'instructions autorisé.
- Cette fonction est appelée exactement une fois et doit construire une séquence d'instructions capable d'effectuer la tâche demandée.

Cette fonction doit donc appeler l'une ou plusieurs des fonctions suivantes pour construire une séquence d'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)
```

- Chacune de ces fonctions ajoute une instruction $move(t, y)$, $store(t, v)$, $and(t, x, y)$, $or(t, x, y)$, $xor(t, x, y)$, $not(t, x)$, $left(t, x, p)$, $right(t, x, p)$ ou $add(t, x, y)$ à la fin du programme courant.
- Pour toutes les instructions concernées, t, x, y doivent être au moins 0 et au plus $m - 1$.
- Pour toutes les instructions concernées, t, x, y ne sont pas nécessairement distincts deux à deux.
- Pour les instructions $left$ et $right$, p doit être au moins 0 et au plus b.
- Pour les instructions $store$, la taille de v doit être b.

Vous pouvez appeler la fonction suivante afin de tester votre solution :

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

- Tout appel à cette fonction durant l'évaluation sur le juge sera ignoré.
- Dans l'évaluateur d'exemple, la fonction ajoute une opération $print(t)$ à la fin du programme courant.
- Quand l'évaluateur d'exemple rencontre une opération $print(t)$ pendant l'exécution, il affiche n entiers de k bits formés des $n \cdot k$ bits du registre t (voir la section "Évaluateur d'exemple" pour plus de détails).
- t doit être tel que $0 \leq t \leq m - 1$.
- Les appels à cette fonction ne comptent pas dans la limite sur le nombre d'instructions.

Une fois que vous avez ajouté la dernière instruction au programme, l'appel `construct_instructions` doit terminer. Le programme est évalué sur un certain nombre de cas d'exemple, chacun spécifié par une entrée constituée de n entiers de k bits : $a[0], a[1], \ldots, a[n-1]$. Votre solution passe un cas d'exemple si la sortie $c[0], c[1], \ldots, c[n-1]$ du programme pour les entiers fournis satisfait les conditions suivantes :

- Si $s = 0$, $c[0]$ doit être la plus petite valeur parmi $a[0], a[1], \ldots, a[n-1]$.
- Si $s = 1$, pour chaque i ($0 \leq i \leq n-1$), $c[i]$ doit être le $1 + i$-th plus petit entier parmi $a[0], a[1], \ldots, a[n-1]$.

L'évaluation de votre solution peut produire l'un des messages d'erreurs suivants :

- `Invalid index` : un index de registre invalide (éventuellement négatif) a été fourni comme paramètre t, x or y à l'un des appels d'une des fonctions.
- `Value to store is not b bits long`: la longueur de v donnée à la fonction `append_store` n'est pas égale à b.
- `Invalid shift value` : la value de p donnée `append_left` ou `append_right` n'est pas entre 0 et b inclus.
- `Too many instructions` : votre fonction a tenté d'ajouter plus de q instructions.

Exemples

Exemple 1

Supposons que $s = 0$, $n = 2$, $k = 1$, $q = 1000$. Il y a deux entiers $a[0]$ et $a[1]$, chacun ayant $k = 1$ bit. Avant que le programme ne soit exécuté, $r[0][0] = a[0]$ et $r[0][1] = a[1]$. Tous les autres bits du processeur sont à 0. À la fin de l'exécution du programme, on doit avoir $c[0] = r[0][0] = \min(a[0], a[1])$, qui est le minimum de $a[0]$ et $a[1]$.

Il y a 4 cas possibles d'entrée au programme :

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

On peut remarquer que pour chacun des 4 cas, $\min(a[0], a[1])$ est égal au ET bit-à-bit de $a[0]$ et $a[1]$. Une solution possible est donc de construire un programme en effectuant les appels suivants :

1. `append_move(1, 0)`, qui ajoute une instruction copiant $r[0]$ dans $r[1]$.
2. `append_right(1, 1, 1)`, qui ajoute une instruction qui prend tous les bits de $r[1]$, les décale vers la droite de 1 bit, puis stocke le résultat dans $r[1]$. Comme chaque entier prend 1 bit, cela a pour conséquence que $r[1][0]$ est égal à $a[1]$.
3. `append_and(0, 0, 1)`, qui ajoute une instruction qui prend le ET bit-à-bit de $r[0]$ et $r[1]$, puis stocke le résultat dans $r[0]$. Une fois que cette instruction est exécutée, $r[0][0]$ correspond, comme ce que nous voulions, au résultat bit-à-bit de $r[0][0]$ et $r[1][0]$, ce qui est égal au ET bit-à-bit de $a[0]$ et $a[1]$.

Exemple 2

Supposons que $s = 1,\ n = 2,\ k = 1,\ q = 1000$. De la même façon que pour l'exemple précédent il n'y a que 4 entrées différentes possible pour le programme. Dans les 4 cas, $\min(a[0], a[1])$ correspond au ET bit-à-bit de $a[0]$ et $a[1]$, et $\max(a[0], a[1])$ est le OU bit-à-bit de $a[0]$ et $a[1]$. Une solution possible est de faire les appels suivants :

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)`

À la fin de l'exécution de ces instructions, $c[0] = r[0][0]$ contient $\min(a[0], a[1])$, et $c[1] = r[0][1]$ contient $\max(a[0], a[1])$, ce qui trie l'entrée.

Contraintes

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

Sous-tâches

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

Évaluateur d'exemple

L'évaluateur d'exemple lit l'entrée au format suivant :

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

Cette ligne est suivie de plusieurs lignes, chacune décrivant un test pour votre programme généré. Chacun de ces tests doit être fourni au format suivant :

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

et cela décrit un test de votre programme avec l'entrée constitué des n entiers $a[0], a[1], ..., a[n-1]$. La description de ces tests est suivie d'un ligne contenant uniquement -1.

L'évaluateur d'exemple appelle d'abord `construct_instructions(s, n, k, q)`. Si cet appel viole une des contraintes du sujet, l'évaluateur d'exemple affiche un des messages d'erreurs donnés à la fin de la section "Détails d'implémentation" et termine. Sinon, l'évaluateur d'exemple affiche les instructions ajoutées par `construct_instructions(s, n, k, q)` dans l'ordre d'ajout. Pour les instructions $store,\ v$ est affiché de l'index 0 à l'index index $b-1$.

Ensuite l'évaluateur d'exemple traite les tests pour le programme généré dans l'ordre où ils apparaissent. Pour chaque test, il exécute le programme généré sur le test fourni.

Pour chaque instruction $print(t)$, on pose $d[0], d[1], \ldots, d[n-1]$ la séquence d'entiers telle que pour chaque i ($0 \le i \le n-1$), $d[i]$ est l'entier associé à la séquence de bits $i \cdot k$ to $(i+1) \cdot k - 1$ du registre t (au moment où l'instruction est exécutée). L'évaluateur d'exemple affiche alors cette séquence au format suivant : `register` t: $d[0]\ d[1]\ \ldots\ d[n-1]$.

Une fois que toutes les instructions ont été exécutées, l'évaluateur d'exemple affiche la sortie du programme.

Si $s = 0$, la sortie de l'évaluateur d'exemple pour chaque test est au format suivant :

- $c[0]$.

Si $s = 1$, la sortie de l'évaluateur pour chaque test est au format suivant :

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

Une fois que tous les tests ont été exécutés, l'évaluateur affiche `number of instructions:` X où X est le nombre d'instructions de votre programme.