Rejestry bitowe (Bit Shift Registers)

Krzysztof jest inżynierem pracującym nad nowym typem procesora.

Procesor ma dostęp do m różnych b-bitowych komórek pamięci (gdzie $m = 100$ oraz $b = 2000$), zwanych **rejestrami** i numerowanymi od 0 do $m - 1$. Rejestry oznaczamy przez $r[0], r[1], \ldots, r[m-1]$. Każdy rejestr jest b-bitową tablicą. Bity numerowane są od 0 (skrajny bit z prawej) do $b - 1$ (skrajny bit z lewej). Dla każdego i $(0 \leq i \leq m - 1)$ i każdego j $(0 \leq j \leq b - 1)$, oznaczamy j-ty bit rejestru i przez $r[i][j]$.

Dla każdego ciągu bitów $d_0, d_1, \ldots, d_{l-1}$ (dowolnej długości l), jego **całkowita wartość** to $2^0 \cdot d_0 + 2^1 \cdot d_1 + \ldots + 2^{l-1} \cdot d_{l-1}$. Zdefiniujmy **wartość całkowitą w rejestrze** i jako całkowitą wartość ciągu jego bitów, czyli $2^0 \cdot r[i][0] + 2^1 \cdot r[i][1] + \ldots + 2^{b-1} \cdot r[i][b-1]$.

Procesor ma 9 rodzajów **instrukcji**, które mogą modyfikować jego zawartość. Każda instrukcja działa na jednym lub więcej rejestrach i zapamiętuje wynik w jednym z nich. Zapis $x := y$ przedstawia zmianę wartości x na y. Operacje związane z każdym typem instrukcji są opisane poniżej.

- $move(t, y)$: Kopiuj tablicę bitów z rejestru y do rejestru t. Dla każdego j $(0 \leq j \leq b - 1)$, wykonaj $r[t][j] := r[y][j]$.
- $store(t, v)$: Zainicjalizuje rejestr t na v, gdzie v jest tablicą zawierającą wartości b bitów. Dla każdego j $(0 \leq j \leq b - 1)$, wykonaj $r[t][j] := v[j]$.
- $and(t, x, y)$: Wykonaj bitową koniunkcję (AND) rejestrów x oraz y, a wynik zapamiętaj w rejestrze t. Dla każdego j $(0 \leq j \leq b - 1)$, wykonaj $r[t][j] := 1$ jeśli **oba** $r[x][j]$ oraz $r[y][j]$ są równe 1, zaś $r[t][j] := 0$ w przeciwnym razie.
- $or(t, x, y)$: Wykonaj bitową alternatywę (OR) rejestrów x oraz y, a wynik zapamiętaj w rejestrze t. Dla każdego j $(0 \leq j \leq b - 1)$, wykonaj $r[t][j] := 1$ jeśli **co najmniej jeden z** $r[x][j]$ oraz $r[y][j]$ jest równy 1, zaś $r[t][j] := 0$ w przeciwnym razie.
- $xor(t, x, y)$: Wykonaj bitową alternatywę wyłączającą (XOR) rejestrów x oraz y, a wynik zapamiętaj w rejestrze t. Dla każdego j $(0 \leq j \leq b - 1)$, wykonaj $r[t][j] := 1$ jeśli **dokładnie jeden z** $r[x][j]$ oraz $r[y][j]$ jest równy 1, zaś $r[t][j] := 0$ w przeciwnym razie.
- $not(t, x)$: Wykonaj negację każdego bitu (NOT) x, a wynik zapamiętaj w rejestrze t. Dla każdego j $(0 \leq j \leq b - 1)$, wykonaj $r[t][j] := 1 - r[x][j]$.
- $left(t, x, p)$: Przesuń wszystkie bity rejestru x w lewo o p pozycji i zapamiętaj wynik w rejestrze t. Wynik przesunięcia w lewo bitów rejestru x o p pozycji jest tablicą v składającą się z b bitów. Dla każdego j $(0 \leq j \leq b - 1)$, $v[j] = r[x][j - p]$ jeśli $j \geq p$, oraz $v[j] = 0$ w przeciwnym razie. Dla każdego j $(0 \leq j \leq b - 1)$, wykonaj $r[t][j] := v[j]$.

- $right(t, x, p)$: Przesuń wszystkie bity rejestru x w prawo o p pozycji i zapamiętaj wynik w rejestrze t. Wynik przesunięcia w prawo bitów rejestru x o p pozycji jest tablicą v składającą się z b bitów. Dla każdego j $(0 \leq j \leq b-1)$, $v[j] = r[x][j+p]$ jeśli $j \leq b-1-p$, oraz $v[j] = 0$ w przeciwnym razie. Dla każdego j $(0 \leq j \leq b-1)$, wykonaj $r[t][j] := v[j]$.
- $add(t, x, y)$: Dodaj całkowite wartości zapamiętane w rejestrach x i y, a wynik zapamiętaj w rejestrze t. Dodawanie wykonuje się modulo 2^b. Formalnie: niech X będzie całkowitą wartością zapamiętaną w rejestrze x, a Y w rejestrze y przed wykonaniem operacji. Niech T będzie całkowitą wartością zapamiętaną w rejestrze t po wykonaniu operacji. Jeśli $X + Y < 2^b$, to bity t będą spełniały $T = X + Y$. W przeciwnym razie bity t będą spełniały $T = X + Y - 2^b$.

Krzysztof chciałby rozwiązać dwa rodzaje zadań na nowym procesorze. Typ zadania jest kodowany przez liczbę całkowitą s. Rozwiązanie obu typów zadań polega na stworzeniu **programu**, który będzie ciągiem instrukcji podanych powyżej.

Wejście do programu składa się z n liczb całkowitych $a[0], a[1], \ldots, a[n-1]$, składających się z k bitów, czyli spełniających nierówność $a[i] < 2^k$ ($0 \leq i \leq n-1$). Zanim program zostanie wykonany, wszystkie liczby są zapamiętane kolejno w rejestrze 0 w taki sposób, że dla każdego i $(0 \leq i \leq n-1)$ wartość całkowita odpowiadająca k bitom $r[0][i \cdot k], r[0][i \cdot k + 1], \ldots, r[0][(i+1) \cdot k - 1]$ jest równa $a[i]$. Mamy zagwarantowane $n \cdot k \leq b$. Pozostałe bity rejestru 0 (te, których indeksy mieszczą się w zakresie od $n \cdot k$ do $b-1$, włącznie) i wszystkie bity pozostałych rejestrów są ustawione na 0.

Uruchomienie programu polega na wykonaniu jego instrukcji w podanej kolejności. Po wykonaniu ostatniej instrukcji programu jego **wyjście** jest określane przez końcową wartość rejestru 0. Konkretnie: wyjście jest ciągiem n liczb całkowitych $c[0], c[1], \ldots, c[n-1]$, gdzie dla każdego i ($0 \leq i \leq n-1$), $c[i]$ jest wartością całkowitą ciągu bitów od $i \cdot k$ do $(i+1) \cdot k - 1$ w rejestrze 0. Zakładamy, że po zakończeniu działania programu pozostałe bity rejestru 0 (z indeksami co najmniej $n \cdot k$) oraz wszystkie bity pozostałych rejestrów mogą być dowolne.

- Pierwsze zadanie $(s = 0)$ polega na wyznaczeniu najmniejszej liczby spośród $a[0], a[1], \ldots, a[n-1]$. Konkretnie, $c[0]$ musi być minimum z $a[0], a[1], \ldots, a[n-1]$. Wartości $c[1], c[2], \ldots, c[n-1]$ mogą być dowolne.
- Drugie zadanie $(s = 1)$ polega na posortowaniu wartości $a[0], a[1], \ldots, a[n-1]$ w kolejności niemalejącej. Konkretnie, dla każdego i ($0 \leq i \leq n-1$), $c[i]$ powinno być równe $(1+i)$-tej (w kolejności od najmniejszej) liczbie spośród $a[0], a[1], \ldots, a[n-1]$ (czyli w szczególności $c[0]$ powinno być najmniejszą ze wszystkich wartości).

Dostarcz Krzysztofowi programy, które za pomocą co najwyżej q instrukcji rozwiążą te problemy.

Szczegóły implementacyjne

Powinieneś zaimplementować następującą procedurę:

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

- s: typ zadania.
- n: liczba liczb całkowitych na wejściu.
- k: liczba bitów w każdej z liczb wejściowych.
- q: maksymalna liczba dozwolonych instrukcji w programie.
- Ta procedura będzie wywołana dokładnie raz i powinna wygenerować ciąg instrukcji rozwiązujących dane zadanie.
- Ta procedura powinna wywołać jedną lub więcej poniższych procedur tworząc stosowny ciąg instrukcji.

```
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)
```

- Wywołanie każdej z procedur $move(t, y)$ $store(t, v)$, $and(t, x, y)$, $or(t, x, y)$, $xor(t, x, y)$, $not(t, x)$, $left(t, x, p)$, $right(t, x, p)$ lub $add(t, x, y)$ powoduje dopisanie odpowiedniej instrukcji na końcu tworzonego programu.
- Dla każdej z instrukcji, wartości parametrów t, x, y muszą być równe co najmniej 0 i co najwyżej $m - 1$.
- We wszystkich instrukcjach, t, x, y nie muszą być parami różne.
- W przypadku instrukcji $left$ i $right$, p musi być równe co najmniej 0 i co najwyżej b.
- W przypadku instrukcji $store$ długość v musi być równa b.

Aby przetestować swoje rozwiązanie możesz użyć dodatkowej instrukcji

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

- Wszystkie wywołania tej procedury będą zignorowane w czasie oceniania Twojego rozwiązania.
- W przykładowej sprawdzaczce ta procedura dodaje operację $print(t)$ na końcu programu.
- Gdy przykładowa sprawdzaczka w casie wykonywania programu napotka na operację $print(t)$, wydrukuje n k-bitowych liczb całkowitych utworzonych z pierwszych $n \cdot k$ bitów rejestru t (zob. rozdział "Przykładowa Sprawdzaczka").
- t musi spełniać warunek $0 \leq t \leq m - 1$.
- Żadne z wykonań tej procedury nie zwiększa liczby użytych instrukcji.

Po dodaniu ostatniej instrukcji procedura `construct_instructions` powinna zakończyć działanie. Powstały program jest testowany na pewnej liczbie przypadków testowych, każdy z nich składa się z n k-bitowych liczb całkowitych $a[0], a[1], \ldots, a[n-1]$. Twoje rozwiązanie zalicza dany test, jeśli wyjście programu $c[0], c[1], \ldots, c[n-1]$ dla zadanego wejścia spełnia następujące warunki:

- Jeśli $s = 0$, to $c[0]$ powinno być najmniejszą spośród wartości $a[0], a[1], \ldots, a[n-1]$.

- Jeśli $s = 1$, to dla każdego i ($0 \leq i \leq n-1$), $c[i]$ powinno być $(1+i)$-tą w kolejności od najmniejszej liczbą spośród $a[0], a[1], \ldots, a[n-1]$.

Ocena Twojego rozwiązania może skutkować jednym z następujących błędów:

- `Invalid index`: niepoprawny indeks (w szczególności ujemny) rejestru podany jako parametr t, x lub y dla któregoś z wywołań procedur.
- `Value to store is not b bits long`: długość v przekazana `append_store` nie jest równa b.
- `Invalid shift value`: wartość p przekazana do `append_left` lub `append_right` nie mieści się w przedziale od 0 do b włącznie.
- `Too many instructions`: Twoja procedura wygenerowała więcej niż q instrukcji.

Przykłady

Przykład 1

Niech $s = 0$, $n = 2$, $k = 1$, $q = 1000$. Mamy dwie wartości na wejściu $a[0]$ i $a[1]$, każda ma $k = 1$ bit. Zanim program zacznie się wykonywać, $r[0][0] = a[0]$ i $r[0][1] = a[1]$. Wszystkie pozostałe bity procesora są ustawione na 0. Po wykonaniu wszystkich instrukcji programu, chcemy mieć $c[0] = r[0][0] = \min(a[0], a[1])$, co jest mniejszą z wartości $a[0]$ i $a[1]$.

Istnieją tylko 4 możliwe wejścia:

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

Można zauważyć, że we wszystkich przypadkach, $\min(a[0], a[1])$ to po prostu bitowa koniunkcja $a[0]$ i $a[1]$. Zatem rozwiązanie problemu polegać może na wykonaniu następujących wywołań:

1. `append_move(1, 0)`, która dodaje instrukcję kopiującą zawartość $r[0]$ do $r[1]$.
2. `append_right(1, 1, 1)`, która dodaje instrukcję biorącą wszystkie bity $r[1]$, przesuwającą je w prawo o 1 bit i zapamiętującą wynik w $r[1]$. Ponieważ każda liczba jest jednobitowa, więc w $r[1][0]$ mamy wartość $a[1]$.
3. `append_and(0, 0, 1)`, która dodaje instrukcję wykonującą bitową koniunkcję $r[0]$ i $r[1]$, i zapamiętującą wynik w $r[0]$. Po wykonaniu tej instrukcji, w $r[0][0]$ mamy wynik bitowej koniunkcji $r[0][0]$ i $r[1][0]$, co jest równe bitowej koniunkcji $a[0]$ i $a[1]$, o co nam chodziło.

Przykład 2

Niech $s = 1$, $n = 2$, $k = 1$, $q = 1000$. Podobnie jak w poprzednim przykładzie, są tylko 4 możliwe wejścia do programu. Dla wszystkich z nich $\min(a[0], a[1])$ jest bitową koniunkcją $a[0]$ i $a[1]$, zaś $\max(a[0], a[1])$ bitową alternatywą $a[0]$ i $a[1]$. Możliwym rozwiązaniem jest wygenerowanie następującego ciągu wywołań:

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)`

Po wykonaniu tych instrukcji, $c[0] = r[0][0]$ zawiera $\min(a[0], a[1])$, a $c[1] = r[0][1]$ zawiera $\max(a[0], a[1])$, co oznacza posortowanie wejścia.

Ograniczenia

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

Podzadania

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

Przykładowa sprawdzaczka

Przykładowa sprawdzaczka czyta wejście w następującym formacie:

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

po którym następuje pewna liczba wierszy, każdy opisujący pojedynczy przypadek testowy. Każdy przypadek testowy ma następujący format:

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

i opisuje przypadek testowy, którego wejście składa się z n liczb całkowitych $a[0], a[1], ..., a[n-1]$. Opis wszystkich przypadków testowych kończy się pojedynczym wierszem zawierającym jedynie -1.

Przykładowa sprawdzaczka najpierw wywołuje `construct_instructions(s, n, k, q)`. Jeśli to wywołanie powoduje błąd, przykładowa sprawdzaczka wypisuje informację o błędzie - jedną z opisanych w rozdziale "Szczegóły implementacyjne" - i kończy działanie.

W przeciwnym razie przykładowa sprawdzaczka najpierw wypisuje po kolei każdą z instrukcji dodanych przez `construct_instructions(s, n, k, q). W przypadku instrukcji $store$ wartość v jest wypisywana od indeksu 0 do indeksu $b-1$.

Następnie przykładowa sprawdzaczka uruchamia program dla kolejnych przypadków testowych uruchamiając je dla ich danych wejściowych.

Dla każdej operacji $print(t)$, niech $d[0], d[1], \ldots, d[n-1]$ będzie ciągiem liczb całkowitych takich, że dla każdego i ($0 \leq i \leq n-1$), $d[i]$ jest wartością całkowitą ciągu bitów od $i \cdot k$ do $(i+1) \cdot k - 1$ rejestru t (w momencie, w który ją wykonujemy). Sprawdzaczka wypisuje ten ciąg w następującym formacie: `register` t: $d[0]\ d[1]\ \ldots\ d[n-1]$.

Gdy wszystkie instrukcje zostaną wykonane, przykładowa sprawdzaczka drukuje wyjście z programu.

Jeśli $s = 0$, wyjście z przykładowej sprawdzaczki dla każdego testu ma format:

- $c[0]$.

Jeśli $s = 1$, wyjście z przykładowej sprawdzaczki dla każdego testu ma format:

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

Po wykonaniu wszystkich przypadków testowych, sprawdzaczka wypisuje `number of instructions:` X, gdzie X jest liczbą instrukcji Twojego programu.