Slabě doleva vychýlené úseky - komentář a řešení

Připravíme si okno délky L, které budeme posouvat po posloupnosti odleva doprava. Průběžně si budeme pamatovat hodnotu maxiam a minima v okně a zároveň i jejich indexy. K tomu nám poslouží prioritní fronta. Minimální prioritní fronta bude obsahovat všechny hodnoty v okně s minimem na čele fronty, Maximální prioritní fronta bude obsahovat všechny hodnoty v okně s maximem na čele fronty. Při posunu okna o jedno pole doleva, odstraníme z obou front prvek nalézající se na doposud nejlevější pozici okna a přidáme do obou front prvek za doposud nejpravější pozicí okna.

Oproti standardním operacím ve frontě musíme implementovat možnost odebírání libovolného prvku fronty, nikoli pouze minima. V okamžiku, kdy nejlevější prvek opustí posouvající se okno, musí být z fronty odstraněn. Aby nebylo nutno procházet celou frontu, budeme si pozici každého prvku posloupnosti ve frontě pamatovat v dodatečném poli. Protože fronty jsou dvě, budeme mít dvě dodatečná pole o délce rovné délce zkoumané posloupnosti. Každá z front bude registrovat jak hodnotu prvku, tak i jeho index v posloupnosti, protože je nutno průběžně porovnávat index maxima a minima v okně. Kromě toho při vkládání a odstraňování prvku z fronty je nutno aktualizovat hodnoty pozic ve frontě všech prvků, které při vložení nebo mazání změnily svou pozici ve frontě.

Poslední trik je nejjednodušší -- implementujeme pouze minimální priorotní frontu, místo maximální prioritní fronty použijeme minimální prioritní frontu, do které budeme vkládat hodnoty vynásobené (−1).

```
static int N;     // total length of the sequence
static int L;     // length of a segment
static int S;     // integer seed of the sequence
static int M;     // modulus of the sequence

// by using negative values the original min-heap becomes a max-heap.
static PriorityQ minQueue;
static PriorityQ maxQueue; //  priority = (-1)*priority

// each sequence element must know its position in the queue
// to allow its effective deletion in O(log(queue.size))time
static int [] posInMinQueue;
static int [] posInMaxQueue;


static int nextValue(int value) {
  return  (int) (( 754043 *(long) value + 500009) % M);
}

static void readAndInitAll() throws IOException {
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  StringTokenizer st = new StringTokenizer(br.readLine());
  N = Integer.valueOf(st.nextToken());
  L = Integer.valueOf(st.nextToken());
  S = Integer.valueOf(st.nextToken());
  M = Integer.valueOf(st.nextToken());

  posInMinQueue = new int [N];
  posInMaxQueue = new int [N];
  minQueue = new PriorityQ(L, posInMinQueue);
  maxQueue = new PriorityQ(L, posInMaxQueue);
}
```

```
static void processAll() {
  int iLeft = 0;      // start of current segment
  int iRight;         // end of current segment
  int total = 0;
  int value = S;      // seed value of the sequence

  // store the initial segment [0 .. L-1] in both queues
  for( iRight = 0; iRight < L; iRight++) {
    minQueue.insert(value, iRight);
    maxQueue.insert((-1)*value, iRight);
    value = nextValue(value);
  }

  iRight--;

  // process segments one by one
  while(true) {
    // check the current segment
    if (minQueue.topindex() < maxQueue.topindex())
      total++;
    // shift left segment  endpoint to the left
    minQueue.delete(posInMinQueue[iLeft]);
    maxQueue.delete(posInMaxQueue[iLeft]);
    iLeft++;

    // shift right segment endpoint to the left
    iRight++;
    if (iRight >= N) break;
    minQueue.insert(value, iRight);
    maxQueue.insert((-1)*value, iRight);
    value = nextValue(value);
  }

  System.out.printf("%d\n", total);
}

public static void main(String[] args) throws IOException {
  readAndInitAll();
  processAll();
}
}

// ======================== PRIORITY QUEUE ================================
class PriorityQ {
// standard binary heap 1-based min-priority queue,
// tail points to the last elem, not behind it

private int [] priority;
private int [] dataPtr;  // reference to the data
private int tail;        // points to last elem in the queue
private int [] posInQ;   // reference to an outer array holding the positions
                         // of elems in the queue

public PriorityQ(int size, int [] positionsArr){
  priority = new int[size+1];
  dataPtr = new int[size+1];
  posInQ = positionsArr;
  tail = 0;  // because unempty heap starts at index 1;
}
```

```
public long topValue() { return priority[1]; }
public long topindex() { return dataPtr[1]; }

public void delete(int delPos) {
  if ((delPos > 1) && (priority[delPos/2] > priority[tail]) ) { // go up
    int j = delPos;
    int i = j/2;
    do {
      priority[j] = priority[i];
      dataPtr[j] = dataPtr[i];
      posInQ[dataPtr[j]] = j;
      j = i;
      i /= 2;
    } while ((i > 0) &&  (priority[i] > priority[tail]) );
    priority[j] = priority[tail];
    dataPtr[j] = dataPtr[tail];
    posInQ[dataPtr[j]] = j;
    tail--;
    return;
  }
  // else go down
  int i = delPos;
  int j = delPos*2;
  while(true) {
    if (j > tail ) break;
    if ((j < tail) && (priority[j] > priority[j+1])) j++;
    if (priority[j] < priority[tail]) {
      priority[i] = priority[j];
      dataPtr[i] = dataPtr[j];
      posInQ[dataPtr[i]] = i;
      i = j;
      j *= 2;
    }
    else break;
  }
  // finally
  priority[i] = priority[tail];
  dataPtr[i] = dataPtr[tail];
  posInQ[dataPtr[i]] = i;
  tail--;
}

public void insert(int  val, int  ptr) {

  if (tail == 0) {
    posInQ[ndx] = 1;
    priority[1] = val;
    dataPtr[1] = ptr;
    tail = 1;
    return;
  }

  int j = ++tail;
  int i = j/2;
  while (true) {
    if(priority[i] >= val) {
      posInQ[dataPtr[i]] = j;
      priority[j] = priority[i];
```

```
      dataPtr[j] = dataPtr[i];
      j = i;
      i = i/2;
      if(i == 0) break;
    }
    else break;
  }
  posInQ[ndx] = j;
  priority[j] = val;
  dataPtr[j] = ptr;
}
```