Záblesky laseru - komentář a řešení

Obrys algoritmického řešení je jednoduchý. Koncové (levé a pravé) body všech intervalů vložíme do jediného pole a uspořádáme vzestupně. Při následném procházení uspořádaného pole pouze průběžně registrujeme rozdíl mezi dvěma veličinami: Kolik levých bodů a kolik pravých koncových bodů jsme již překročili. Rozdíl udává, uvnitř kolika intervalů (záblesků) se aktuálně nacházíme. Stačí registrovat pouze hodnotu rozdílu a při návštěvě každého koncového bodu rozdíl o 1 zvýšit nebo snížit, podle toho, zda se jedná o bod levý nebo pravý.

V implementaci se projeví několik aspektů. Zaprvé, levé a pravé koncové body je pohodlné mít seřazené v jediném poli, ale přitom je třeba rozeznávat levé od pravých. To lze zařídit přidáním jediného bitu tak, že místo souřadnice L levého koncového bodu do pole uložíme hodnotu 2L a místo souřadnice R pravého koncového bodu do pole uložíme hodnotu 2R+1. Nejméně signifikantní bit čísla pak udává, zda jde o levý koncový bod (0), nebo pravý koncový bod (1). Původní hodnotu souřadnice získáme kdykoli zpět pouhým celočíselným vydělením 2. Pořadí bodů při řazení se tím také nepoškodí, pokud má více bodů stejné souřadnice, budou v seřazeném poli napřed uvedeny levé koncové body (mají nulu na pozici 0 řádu) a pak pravé koncové body (mají jedničku na pozici 0 řádu).

Uvedeným obratem se počet datovývch struktur potřebných pro řešení úlohy redukuje na jediné celočíselné pole o délce rovné dvojnásobku počtu záblesků.

Generující posloupnost souřadnic záblesků vytváří čísla s až 11 platnými číslicemi. Vytvoření dalšího členu posloupnosti vyžaduje výpočet druhé mocniny předchozího členu, a tak je nutno pracovat až s 22 platnými číslicemi.

Informace obsažená v 22 desetinných číslicích vyžaduje ke svému uložení horní celou část $\log_2(10^{22})$ bitů, což se rovná $\text{ceil}(22 \cdot \log_2(10)) = 74$. Běžné číselné datové typy programovacího jazyka (long, double apod) obsazují maximálně 64 bity, takže pro spolehlivý výpočet je nelze použít.

Relativně nenáročným cvičením je proto vytvoření vlastní reprezentace velkých čísel a připsání několika funkcí, které s těmito velkými čísla provádějí nezbytné aritmetické operace. V předkládaném řešení je číslo uloženo do pole, přičemž každý prvek pole obsahuje jednu číslici čísla. Každá číslice je ovšem zároveň sama sebou také číslem, takže základ číselné soustavy, ve které se daná čísla reprezentují a ukládají do pole, nemusí být 10. Pro úsporu místa (a tím i rychlejší výpočet) je zvolen základ 10^9. Součin dvou číslic (budeme muset násobit) tak nepřevýší 10^{18}. Tím se paměť využije poměrně efektivně, protože největší zobrazitelná celočíselná hodnota v primitivním typu je $9.223372036854775807 \cdot 10^{18}$ v Javě a $18.446744073709551615 \cdot 10^{18}$ v jiných, kulturnějších jazycích. Jako základ soustavy lze ovšem použít i maximálně možný základ $2^{31} - 1$ místo 10^9, jenž se ale zdál na první pohled "lidsky srozumitelnější".

Funkce pro násobení a dělení jsou doslovnou kopií algoritmů dělení a násobení celých čísel známých ze základní školy, pro jednoduchost je implementováno pouze dělení jednociferným číslem. Úloha žádá pouze dělení číslem 10^P, což lze zařídit P-krát opakovaným dělením 10 a desítka je zapsána jedinou číslicí v soustavě s větším základem.

Uvedený přístup je prezentován jako cvičení, v praktické aplikaci bychom nejspíše použili v Javě Třídu BigInteger a v C/C++ některou z knihoven (http://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic#Libraries).

```
import java.io.*;
import java.util.StringTokenizer;
import java.util.Arrays;

public class Uloha07 {

static int P;                   // precision
static long ScreenWidth;
static int L;                   // no of flashes, length of the data set
static long D;                  // width of flash;
static int K;                   // after K flashes screen locally darkens
static long Z;                  // chaotic seed
```

```
static long [] flashBordersLR;
// Flash border is the coordinate of the boundary between
// illuminated and not illuminated part of the screen in a single laser flash.
// Each flash generates exactly two borders L and R.
// In this implementation the L and R borders are distinguished
// by the least significant bit: L := L*2+0, R := R*2+1.
// This allows to store L and R border coordinates in a single array,
// and sort them effectively. To obtain original border value just do L/2 or R/2.

static long shortestDarkZoneLen = ScreenWidth;
static long longestDarkZoneLen = 0;
static long TotalDarkZoneLen = 0;


static long trim(long value, long min, long max ) {
  // trim the flash area when it goes partially (or completely) off the screen
  if (value < 0) return 0;
  if (value > max) return max;
  return value;
}

static void fillFlashBorders() {

  long Yi = Z;                       // seed
  long flashBorderL, flashBorderR;  // for more readability only

  for (int i = 0; i < L; i++) {
    flashBorderL = trim(Yi-D, 0, ScreenWidth);
    flashBordersLR[2*i] = flashBorderL*2;
    flashBorderR = trim(Yi+D, 0, ScreenWidth);
    flashBordersLR[2*i+1] = flashBorderR*2+1;
    Yi = nextY(Yi);
  }
}

static void processSortedFlashBorders() {
  long currDarkStart = 0;
  long currDarkLen;
  long state = 0; // Indicates whether the current screen pos is out or in a dark zone
                  // Passing accros the L resp. R. border from left to right means:
                  // increase resp. decrease the state by one.
  long prevState = 0;
  int screenPos = 0;        // scans the whole sorted array of flash borders

  while(screenPos < flashBordersLR.length) {
    prevState = state;      // remember previous state

    // check if at the current pos there is L or R flash border
    if (flashBordersLR[screenPos] % 2 == 0) state++; else state--;
    // evaluate all L and R flash borders which share current screen position
    while ((screenPos < flashBordersLR.length-1)
           && (flashBordersLR[screenPos]/2 == flashBordersLR[screenPos+1]/2)) {
      screenPos++;
      if (flashBordersLR[screenPos] % 2 == 0) state++; else state--;
    }
    // maybe at current screen position there is a start of a new dark segment
    if ((prevState < K) && ( state >= K )) {
      currDarkStart = flashBordersLR[screenPos]/2;
    }
```

```
    // maybe at current screen position there is an end of a dark segment
    if ((prevState >=  K) && ( state < K )) {
      currDarkLen = flashBordersLR[screenPos]/2- currDarkStart;
      TotalDarkZoneLen += currDarkLen;
      if (currDarkLen < shortestDarkZoneLen)
        shortestDarkZoneLen = currDarkLen;
      if (currDarkLen > longestDarkZoneLen)
        longestDarkZoneLen = currDarkLen;
    }
    // move to next registered position of a flash border
    screenPos++;
  } // while

  if (TotalDarkZoneLen == 0) shortestDarkZoneLen = 0;
}

static void readAndInitAll () throws IOException {
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  StringTokenizer st = new StringTokenizer(br.readLine());
  P = Integer.valueOf(st.nextToken());
  D = Integer.valueOf(st.nextToken());
  L = Integer.valueOf(st.nextToken());
  K = Integer.valueOf(st.nextToken());
  Z = Integer.valueOf(st.nextToken());

  flashBordersLR = new long [L * 2];

  int auxP = P;
  ScreenWidth = 1;
  while(auxP > 0) { ScreenWidth *= 10; auxP--; }

  shortestDarkZoneLen = ScreenWidth;
  longestDarkZoneLen = 0;
  TotalDarkZoneLen = 0;
}

public static void main(String[] args) throws IOException {
  readAndInitAll();
  fillFlashBorders();
  Arrays.sort(flashBordersLR);
  processSortedFlashBorders();
  System.out.printf("%d %d %d\n",
                    shortestDarkZoneLen, longestDarkZoneLen, TotalDarkZoneLen);
}

// ------------------------------------------------------------------------
//                      GENERATING THE SEQUENCE
// ------------------------------------------------------------------------

// The sequence generating formula requires computing the square of a number
// with max 11 decimal digits. Thus the square may have up to 22 digits
// which does not fit into any of 64 bits data types and consequently
// if these were used a rounding error could destroy the correct sequence.

// Trivial arithmetic routines dealing with larger integers are implemented:
// Integer is represented as an array of digits from the set {0, .. base-1}
// in the number system with base anywhere between 2 and 2^32-1.
// Actually, the base 10^9 is used here.
```

```
// Apart from multiplication the formula in the problem needs to perform only
// division by a power of 10, thus only a simple integer division
// by one digit (10 is used ) is implemented here and applied more times when needed.

// Three arrays store the operands of the formula  4*Y[i]*(1-Y[i])/10^P )
static long [] factor1 = new long [3];
static long [] factor2 = new long [3];
static long [] product = new long [6];
static long base = 1000000000; //10^9,

static long nextY(long Yi ){
  // implementing  formula Y[i+1] = floor( 4*Y[i]*(ScreenWidth-Y[i])/ScreenWidth )
  if (Yi == 0) return 1L;

  // 1.  calculate 4*Y[i]*(M-Y[i])
  // 1.a  multiply by 4
  long Yi2 = ScreenWidth-Yi;
    if (Yi < Yi2) Yi *= 4; else Yi2 *= 4;

  // 1.b  multiply Yi * Yi2
  encode(Yi, factor1, base);
  encode(Yi2, factor2, base);
  multiply(factor1, factor2, product, base);

  // 2.  divide by screen width
  for(int i = 0; i < P; i++) {
    divByDigit(product, 10, base);  // divide by 10 P times == divide by 10^P
  }
  return decode(product, base);
 }

static void divByDigit(long [] a, long digit,  long base ) {
  // clear result
  long carry = 0;
  long twoDigitsVal;
  for (int i = a.length-1; i >= 0; i--)  {
    twoDigitsVal = carry*base+a[i];
    a[i] = twoDigitsVal / digit;
    carry = twoDigitsVal % digit;
  }
}

static void encode(long x, long [] a, long base) {
  int i = 0;
  while(x > 0) {
    a[i++] = x % base;
    x /= base;
  }
  while( i < a.length) a[i++] = 0;
}

static long decode(long [] a, long base) {
  long x = a[a.length-1];
  for(int i = a.length-2; i >= 0; i--)
    x = x * base + a[i];
  return x;
}
```

```
static void multiply(long [] a, long [] b, long [] result, long base) {
  // clear result
  for(int i = 0; i < result.length; i++)
    result[i] = 0;
  // multiply
  long carry = 0;
  long tmp;
  for(int j = 0; j < b.length; j++) {
    carry = 0;
    for(int i = 0; i < a.length; i++) {
      tmp = result[i+j] + a[i]*b[j] + carry;
      result[i+j] = tmp % base;
      carry = tmp / base;
    }
  }
}

}
```