Robot - komentář a řešení

Protože úloha je poměrně jednoduchá, deklarujeme tentokrát veškeré důležité proměnné jako globální, není jich mnoho. Následující kód je všechny zachycuje i s relevantními komentáři.

```
static int M, N;                    // network dimensions
static int yStart, xStart;          // search start coordinates
static int yTarget, xTarget;        // search end coordinates
static int y, x;                    // current step coordinates
static int yNext, xNext;            // next step coordinates
static int minNeighMark;            // best mark in the neighbourhood

static boolean visited [][];        // visited square == !fresh square in DFS
static int [] stack;
static int SP;                      // stack pointer
static int totalSteps = 0;

static int a, b;                    // constants define square marks
```

Funkce main bude také poměrně jednoduchá.

```
public static void main(String[] args) throws IOException  {
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

  // suffer through the data input
  StringTokenizer st = new StringTokenizer(br.readLine());
  M = Integer.valueOf(st.nextToken());
  N = Integer.valueOf(st.nextToken());
  a = Integer.valueOf(st.nextToken());
  b = Integer.valueOf(st.nextToken());
  st = new StringTokenizer(br.readLine());
  yStart = Integer.valueOf(st.nextToken());
  xStart = Integer.valueOf(st.nextToken());
  yTarget = Integer.valueOf(st.nextToken());
  xTarget = Integer.valueOf(st.nextToken());

  // initialize
  stack = new int [2*M*N];
  visited = new boolean [M][N];

  // run the program
  DepthFirst_Search();
  System.out.printf("%d\n",totalSteps);
 }
```

Úloha vyžaduje provést jen základní průchod do hloubky, takže spoužijeme zcela jednoduchý zásobník, kam si budeme ukládat souřadnice navštívených políček, na něž se případně později budeme vracet. Obyčejné 1D pole celých čísel nám jako takový zásobník bohatě postačí. Protože ale každé políčko má dvě souřadnice, budeme tyto dvě souřadnice do pole-zásobníku ukládat vždy do dvou prvků pole jdoucích za sebou. Operace push a pop nad tímto zásobníkem pak vždy změní hodnotu ukazatele vrcholu zásobníku (stack pointer, SP) přesně o 2. Na této strategii není nic exotického, z pohledu blížšího stroji se tak chová prakticky každý zásobník, protože se na něj běžně ukládají data (čísla, reference, ukazatele), která většinou zabírají počet Bytů větší než 1 (typicky násobek 4 nebo 8), a tak interní hodnota ukazatele vrchu zásobníku se málokdy mění jen o 1.

Další naše činnost už jen bude kopírovat předepsaný postup provádění DFS s tím, že uzly našeho grafu jsou políčka ve 2D čtvercové síti. Výhodou je uniformní tvar okolí skoro každého políčka v síti (s výjimkou okrajů, které ošetříme snadno), a tak nemusíme psát speciální vnitřní cyklus, který by toto okolí prohledával, takže jen provedeme

čtyři vhodné testy **checkNeighbour** za sebou. Tyto testy musí nastat ve vhodném pořadí, protože robot při prohledávání a objevení stejných hodnot značek na sousedních políčkách má předepsáno specifické pořadí, v němž je prohledává. Drobnou nevýhodou je, že hodnoty **minNeighMark, yNext, xNext** nemohou být parametry funkce, protože se všechny mohou měnit a funkce v Javě může vracet jen jednu hodnotu. Přehlednost důležité části kódu je tím trochu snížena, (např . proměnné **yNext, xNext** nejsou v těle funkce **DepthFirst_Search** vůbec formálně inicializovány) ale v programu této velikosti si takový prohřešek možná můžeme dovolit.

```
static void DepthFirst_Search() {

  // initialize stack, meaning:  stack.push((startY,startX));
  stack[0] = yStart;
  stack[1] = xStart;
  SP = 0;                      // stack pointer points to the very stack bottom

  while (true) {
    // look at the stack top
    y = stack[SP];
    x = stack[SP+1];
    if ((y == yTarget) && (x == xTarget))
      break;                   // stop, we have arrived at the desired position

    visited[y][x] = true;      // in any case note that this square has been visited

    //scan the (maximum) four neighbouring positions and find where to move next
    minNeighMark = Integer.MAX_VALUE;
    if (x > 0)   checkNeighbour(y, x-1);  // check W direction
    if (x < N-1) checkNeighbour(y, x+1);  // check E direction
    if (y > 0)   checkNeighbour(y-1, x);  // check N direction
    if (y < M-1) checkNeighbour(y+1, x);  // check S direction

    // now it is known where to move:
    if (minNeighMark < Integer.MAX_VALUE) {
      SP += 2;
      stack[SP] = yNext;       // decrease in minMark value => go forward == push
      stack[SP+1] = xNext;
    }
    else                       // no change in minMark value  => go back == pop
      SP -= 2;

    totalSteps++;              // anyway a step is being performed
  }
} // DFS
```

Opět, funkce **checkNeighbour** ověřující, zda lze postoupit na sousední políčko, je pokud možno kompaktně provedena, využívá faktu, že proměnné jsou deklarovány globálně, aby mohla de facto vracet všechny tři podstatné hodnoty **minNeighMark, yNext, xNext**.

```
static void checkNeighbour(int yNeigh, int xNeigh) {
  int mark = (( ((r*N) %b ) + s ) * a) % b;  // two operations % to avoid int overflow
  if ((mark < minNeighMark) &&  !visited[yNeigh][xNeigh])  {
      minNeighMark = mark;
      yNext = yNeigh;
      xNext = xNeigh;
    }
}
```

Pro úplnost připojujeme také historicky bezprostředně předchozí variantu prohledávání do hloubky, kde je zkoumání čtyř sousedních políček dopodrobna rozepsáno, a pro čtení proto bude patrně snazší. Autorovi se ale zželelo kódu, kde se čtyřikrát opakuje téměř totéž, což za cenu vetší přehlednosti naopak zvětšuje prostor pro „oblíbené“ překlepy typu x/y, +/– 1 a vůbec nafukuje kód. Čtenář sám posoudí, co je za daných okolností vhodnější.

```
static int mark(int r, int s) {
  return ((  (r*N) % b + s) * a) % b;  //twice operation % to avoid overflow
}

static void DepthFirst_Search() {

  // initialize stack, meaning:  stack.push((startY,startX));
  stack[0] = startY;
  stack[1] = startX;
  SP = 0;                       // stack pointer points to the stack bottom

  while (true) {
    // look at the stack top
    Y = stack[SP];
    X = stack[SP+1];
    if ((Y == targetY) && (X == targetX))
      break;                   // stop, we have arrived at the desired position

    visited[Y][X] = true;     //even if it was visited before

    //scan the (maximum) four neighbouring positions and find where to move next
    minMark = Integer.MAX_VALUE;
    int nextMark;
    nextY = nextX = -1;
    // check W direction
    nextMark = mark(Y, X-1) ;
    if ((X > 0) && (!visited[Y][X-1]) && (nextMark < minMark) )  {
      minMark = nextMark;
      nextY = Y;
      nextX = X-1;
    }
    // check E direction
    nextMark = mark(Y, X+1);
    if ((X < N-1) && (!visited[Y][X+1]) && (nextMark < minMark) ){
      minMark = nextMark;
      nextY = Y;
      nextX = X+1;
    }
    // check S direction
    nextMark = mark(Y+1, X);
    if ((Y < M-1) && (!visited[Y+1][X]) && (nextMark < minMark) ){
      minMark = nextMark;
      nextY = Y+1;
      nextX = X;
    }
    // check N direction
    nextMark = mark(Y-1, X);
    if ((Y > 0) && (!visited[Y-1][X]) && (nextMark < minMark) ){
      minMark = nextMark;
      nextY = Y-1;
      nextX = X;
    }
```

```
    // now it is known where to move:
    if (minMark < Integer.MAX_VALUE) {
      SP += 2;
      stack[SP] = nextY;           // push the new position == go forward
      stack[SP+1] = nextX;
    }
    else
      SP -= 2; // if no change in minmark occured => go BACK == pop the stack

    // anyway a step is being performed
    totalSteps++;
  }
}
```