EJERCICIOS TEMA 5

Algoritmia y Complejidad

Realizado por:

Gabriel López Cuenca
Miguel Ángel Losada Fernández
Sergio Sanz Sacristán
Álvaro Zamorano Ortega

Ejercicio 4

Se dispone de un tablero M de tamaño FxC (F es la cantidad de filas y C la cantidad de columnas) y se pone en una casilla inicial (posx, posy) un caballo de ajedrez. El objetivo es encontrar, si es posible, la forma en la que el caballo debe moverse para recorrer todo el tablero de manera que cada casilla se utilice una única vez en el recorrido (el tablero 8x8 siempre tiene solución independientemente de dónde comience el caballo). El caballo puede terminar en cualquier posición del tablero.

Un caballo tiene ocho posibles movimientos (suponiendo, claro está, que no se sale del tablero). Un movimiento entre las casillas M_{ij} y M_{pq} es válido solamente si:

- **(|p–i|=1)&&(|q–j|=2), o bien si**
- **(|p–i|=2)&&(|q–j|=1),**

es decir, una coordenada cambia dos unidades y la otra una única unidad.

Primero debemos observar como es el movimiento de un caballo en el tablero. El movimiento de un caballo podríamos decir que es en forma de L, esto quiere decir, si el caballo se mueve solo una posición a la derecha o izquierda, también se mueve dos posiciones arriba o abajo, si se mueve dos posiciones a la derecha o a la izquierda, también se mueve una posición arriba o abajo.

También debemos comprobar que el movimiento aparte de realizarse correctamente, que esté dentro de los límites del tablero.

```
public class TEMA5_EJ4 {

    //Atributos de la clase
    private static final int columnas = 8;
    private static final int filas = 8;
    private static int[][] tablero = new int[filas][columnas];
    private static int[][] movimientos = new int[2][8];

    public static void main(String[] args) {
        Inicializar();

        //Posición de partida del caballo
        int x = 0, y = 0;

        boolean sol = movCab(x, y, 1);

        if (sol) {
            System.out.println("El caballo puede recorrer todas las posiciones"
                    + " del tablero desde la posición (" + (x + 1) + ","
                    + (y + 1) + ")");
        } else {
            System.out.println("El caballo no puede recorrer todas las "
                    + "posiciones del tablero desde la posición ("
                    + (x + 1) + "," + (y + 1) + ")");
        }
    }

    /**
     * Método para inicializar el tablero y los movimientos que puede realizar
     * el caballo
     *
     */
```

```java
/**
 * Método para generar las llamadas recursivas en las que el caballo se
 * mueve por el tablero
 *
 * @param x
 * @param y
 * @param n
 * @return
 */
private static boolean movCab(int x, int y, int n) {
    int ax, ay;

    //Indicar en que movimiento pasa el caballo por esa casilla, es decir,
    //se recorre
    tablero[x][y] = n;

    if (n == filas * columnas) {
        //El número de movimientos es igual al número de casillas del
        //tablero, habrá terminado
        imprimirMatriz(tablero);

        return true;
    }

    for (int i = 0; i < columnas; i++) {
        //Para cada uno de los posibles movimientos que puede hacer el
        //caballo
        ax = x + movimientos[0][i];
        ay = y + movimientos[1][i];
        if (ax >= 0 && ax < filas && ay >= 0 && ay < columnas) {
            //Si el movimiento se encuentra dentro del tablero
            if (tablero[ax][ay] == 0) {
                //En caso de que la casilla no haya sido recorrida,
                //se recorre y se intenta hacer un nuevo movimiento
                //(se vuelve a empezar)
                if (movCab(ax, ay, n + 1)) {
                    return true;
                }
            }
        }
    }

    //Desmarcar la casilla que no sirve en esta rama para que se pueda
    //pasar por esta casilla en llamadas posteriores
    tablero[x][y] = 0;

    //Una vez llegados aquí, no se podrá continuar en la recursividad
    //o el caballo no habrá recorrido todas las posiciones sin tener
    //más movimientos disponibles
    return false;
}

/**
 * Método para imprimir el tablero donde se muestra en que movimiento ha
 * pasado el caballo por cada casilla
 *
 * @param matriz
 */
private static void imprimirMatriz(int[][] matriz) {
    System.out.println("**** MATRIZ DE MOVIMIENTOS ****");
    for (int i = 0; i < filas; i++) {
        for (int j = 0; j < columnas; j++) {
            System.out.print(matriz[i][j]);
```

```
                if (j < columnas - 1) {
                    System.out.print(", ");
                }
            }
            System.out.println("\n");
        }
    }
}
```

Ejercicio 6

Se tiene la tabla de sustitución que aparece a continuación:

	a	b	c	d
a	b	b	a	d
b	c	a	d	a
c	b	a	c	c
d	d	c	d	b

que se usa de la manera siguiente: en una cadena cualquiera, dos caracteres consecutivos se pueden sustituir por el valor que aparece en la tabla, utilizando el primer carácter como fila y el segundo carácter como columna. Por ejemplo, se puede cambiar la secuencia ca por una b, ya que M[c,a]=b.

Implementar un algoritmo Backtracking que, a partir de una cadena no vacía texto y utilizando la información almacenada en una tabla de sustitución M, sea capaz de encontrar la forma de realizar las sustituciones que permite reducir la cadena texto a un carácter final, si es posible.

Ejemplo: Con la cadena texto=acabada y el carácter final=d, una posible forma de sustitución es la siguiente (las secuencias que se sustituyen se marcan para mayor claridad): acabada → acacda → abcda → abcd → bcd → bc → d.

Lo primero que debemos hacer es crear una matriz que contendrá los datos de la tabla de sustitución del enunciado. Vamos seleccionando dos letras de la palabra de entrada y realizamos la sustitución correspondiente, continuamos este proceso hasta que solo quede una letra y esta:

- Sea igual a la letra final, por lo tanto, hemos terminado con éxito.
- Sea diferente a la letra final y se vuelve atrás mediante el árbol generado de la recursividad. Al volver cogeríamos dos letras distintas y repetiríamos el proceso. Si después de recorrer todo el árbol en ningún momento se encuentra el resultado esperado entonces no será posible realizar la transformación.

```
public class TEMA5_EJ6 {

    //Atributo para almacenar la matriz de transformaciones
    private static final char[][] matriz = {{'b', 'b', 'a', 'd'},
    {'c', 'a', 'd', 'a'},
    {'b', 'a', 'c', 'c'},
    {'d', 'c', 'd', 'b'}};

    //Atributo para almacenar las posibles letras que hay en la matriz
    private static ArrayList<String> letras;

    public static void main(String[] args) {
        String entrada = "acabada";
        String fin = "d";

        letras = letrasMatriz();

        if (reemplazar(entrada, matriz, fin)) {
            System.out.println("La palabra " + entrada.toUpperCase()
                    + " se puede sustituir por una " + fin);
        } else {
            System.out.println("La palabra " + entrada.toUpperCase()
                    + " no se puede sustituir por una " + fin);
        }
    }
```

```java
/**
 * Método para hacer las posibles transformaciones mediante llamadas
 * recursivas
 *
 * @param entrada
 * @param m
 * @param f
 * @return
 */
private static boolean reemplazar(String entrada, char[][] m, String f) {
    String[] aux = new String[3];

    //Separar la cadena de entrada por letras
    String[] e = new String[entrada.length()];
    for (int i = 0; i < entrada.length(); i++) {
        e[i] = "" + entrada.charAt(i);
    }
    int len = e.length;

    if (len == 1) {
        //En caso de que la cadena sea de tamaño 1
        if (e[0].equals(f)) {
            //Si el único carácter que tiene es el final que queremos
            //obtener, se puede hacer la transformación
            return true;
        }
    } else {
        for (int i = 1; i < len; i++) {
            //En la primera posición del array se almacena el comienzo de
            //la palabra hasta i-1 (posición en la que se está haciendo
            //la transformación
            aux[0] = str(e, 0, i - 1);

            //En la segunda posición del array almacenamos la transformación
            if (i + 1 == len) {
                aux[1] = pos(e[i], e[len - 1]);
            } else {
                aux[1] = pos(e[i], e[i + 1]);
            }

            //En la tercera posición del array se almacena el resto de la
            //palabra aún sin procesar
            aux[2] = str(e, i + 2, len);

            if (reemplazar(aux[0] + aux[1] + aux[2], m, f)) {
                //Llamamos al método recursivamente con la nueva palabra
                //conseguida tras la transformación
                return true;
            }
        }
    }
    //Llegados a este punto, la transformación no se ha podido realizar
    return false;
}

/**
 * Método para obtener una subcadena de una cadena dada en un array que
 * contiene sus letras
 *
 * @param e
 * @param i
 * @param j
 * @return
 */
```

```
    private static String str(String[] e, int i, int j) {
        String dev = "";
        if (i < j) {
            for (int k = i; k < j; k++) {
                dev += e[k];
            }
        }
        return dev;
    }

    /**
     * Método para obtener la letra por la que se deben cambiar las 2 letras
     * correspondientes
     *
     * @param a
     * @param b
     * @return
     */
    private static String pos(String a, String b) {
        return ("" + matriz[letras.indexOf(a)][letras.indexOf(b)]);
    }

    /**
     * Método para buscar las diferentes letras que hay en la matriz de
     * transformaciones
     *
     * @return
     */
    private static ArrayList<String> letrasMatriz() {
        ArrayList<String> ls = new ArrayList<>();
        String l;

        for (int i = 0; i < matriz.length; i++) {
            for (int j = 0; j < matriz.length; j++) {
                l = "" + matriz[i][j];
                if (!ls.contains(l)) {
                    ls.add(l);
                }
            }
        }
        Collections.sort(ls);
        return ls;
    }
}
```