UNIVERSIDAD AUTONOMA GABRIEL RENE MORENO

FACULTAD DE INGENIERIA Y CIENCIAS DE LA COMPUTACION Y TELECOMUNICACIONES

INTELIGENCIA ARTIFICIAL

CONTENIDO:

TAREA #4. EL PROBLEMA DE LA TORRE, ALFIL, DAMA..

PORCENTAJE TERMINADO: 100%

INTEGRANTES	DT	HG	HI	EVAL
Garcia Taboada Brayan Albaro	1	0	1	100

Fecha de presentación : Lunes, 20 de Mayo de 2024

Fecha Presentada : : Lunes, 20 de Mayo de 2024

Días de Atraso : 0

Caminos de Rey, Caballo, Torre, Alfil y la Dama.

TRABAJO INDIVIDUAL.

Dado una matriz de n x m, inicialmente con valores de ceros. Implementar Algoritmos con llamadas recursivas desde un ciclo, para cada uno de los problemas de los movimientos de la Torre, Alfil y la Dama. Implementar y ejecutar para diferentes valores de n y m.

a) Algoritmo para mostrar todos los caminos posibles desde una posición inicial a una posición final. Además, mostrar la cantidad de soluciones posibles.

```
public static LinkedList<Regla> reglasAplicablesTorre(int m[][],
                                                       int i, int j) {
    LinkedList<Regla> L1 = new LinkedList();
    int j1=j-1;
    while(posValida(m,i,j:j1)){
        L1.add(new Regla(fil:i,col:j1));
        j1--;
    }
    int i1=i-1;
    while(posValida(m,i:i1,j)){
        L1.add(new Regla(fil:i1,col:j));
        i1--;
    }
    j1=j+1;
    while(posValida(m,i,j:j1)){
        L1.add(new Regla(fil:i,col:j1));
        j1++;
    }
    i1=i+1;
    while(posValida(m,i:i1,j)){
        L1.add(new Regla(fil:i1,col:j));
        i1++;
    }
    return L1;
}
```

```java
public static LinkedList<Regla> reglasAplicablesAlfil(int m[][], int i, int j) {
    LinkedList<Regla> L1 = new LinkedList();
    int i1 = i - 1;
    int j1 = j - 1;
    while (posValida(m, i: i1, j: j1)) {
        L1.add(new Regla(fil:i1, col:j1));
        i1--;
        j1--;
    }
    i1 = i - 1;
    j1 = j + 1;
    while (posValida(m, i: i1, j: j1)) {
        L1.add(new Regla(fil:i1, col:j1));
        i1--;
        j1++;
    }
    i1 = i + 1;
    j1 = j + 1;
    while (posValida(m, i: i1, j: j1)) {
        L1.add(new Regla(fil:i1, col:j1));
        i1++;
        j1++;
    }
    i1 = i + 1;
    j1 = j - 1;
    while (posValida(m, i: i1, j: j1)) {
        L1.add(new Regla(fil:i1, col:j1));
        i1++;
        j1--;
    }
    return L1;
}
```

```
public static LinkedList<Regla> reglasAplicablesDama(int m[][], int i, int j) {
    LinkedList<Regla> L1 = new LinkedList();
    int j1=j-1;
    while(posValida(m,i,j: j1)){
        L1.add(new Regla(fil:i,col:j1));
        j1--;
    }
    int i1 = i - 1; j1 = j - 1;
    while (posValida(m, i: i1, j: j1)) {
        L1.add(new Regla(fil:i1, col:j1));
        i1--;
        j1--;
    }
    i1=i-1;
    while(posValida(m,i: i1,j)){
        L1.add(new Regla(fil:i1,col:j));
        i1--;
    }
    i1 = i - 1;j1 = j + 1;
    while (posValida(m, i: i1, j: j1)) {
        L1.add(new Regla(fil:i1, col:j1));
        i1--;
        j1++;
    }
    j1=j+1;
    while(posValida(m,i,j: j1)){
        L1.add(new Regla(fil:i,col:j1));
        j1++;
    }
    i1 = i + 1; j1 = j + 1;
    while (posValida(m, i: i1, j: j1)) {
        L1.add(new Regla(fil:i1, col:j1));
        i1++;
        j1++;
    }
    i1=i+1;
    while(posValida(m,i: i1,j)){
        L1.add(new Regla(fil:i1,col:j));
        i1++;
    }
    i1 = i + 1; j1 = j - 1;
    while (posValida(m, i: i1, j: j1)) {
        L1.add(new Regla(fil:i1, col:j1));
        i1++;
        j1--;
    }
    return L1;
}
```

b) Algoritmo para mostrar todos los caminos posibles desde una posición inicial a una posición final tal que se visiten todas las casillas de la matriz. Además, mostrar la cantidad de soluciones posibles.

```
public static void laberinto(int m[][], int i, int j, int iF, int jF, int paso) {
    m[i][j] = paso;
    if (i == iF && j == jF && paso==(m.length*m[i].length)) {
        mostrar(m);
        cant++;
    }
    LinkedList<Regla> L1 = reglasAplicables(m, i, j);
    while (!L1.isEmpty()) {
        Regla R = elegirRegla(L1);
        laberinto(m, i: R.fil, j: R.col, iF, jF, paso + 1);
        m[R.fil][R.col] = 0;
    }
}
```

c) Algoritmo para mostrar todos los caminos posibles desde una posición inicial a una posición final tal que NO se visiten todas las casillas de la matriz. Además, mostrar la cantidad de soluciones posibles.

```
public static void laberinto(int m[][], int i, int j, int iF, int jF, int paso) {
    m[i][j] = paso;
    if (i == iF && j == jF && paso!=(m.length*m[i].length)) {
        mostrar(m);
        cant++;
    }
    LinkedList<Regla> L1 = reglasAplicables(m, i, j);
    while (!L1.isEmpty()) {
        Regla R = elegirRegla(L1);
        laberinto(m, i: R.fil, j: R.col, iF, jF, paso + 1);
        m[R.fil][R.col] = 0;
    }
}
```

d) Algoritmo para mostrar todos los caminos posibles de máxima longitud desde una posición inicial a una posición final. Además, mostrar la cantidad de soluciones posibles.

```
public static int maxPasos = 0;
public static LinkedList<int[][]> soluciones = new LinkedList<>();

public static void laberinto(int m[][], int i, int j, int iF, int jF, int paso) {
    m[i][j] = paso;
    if (i == iF && j == jF) {
        if (paso > maxPasos) {
            maxPasos = paso;
            soluciones.clear();
            soluciones.add(e: m);
            cant = 1;
        } else if (paso == maxPasos) {
            soluciones.add(e: m);
            cant++;
        }
    }
    LinkedList<Regla> L1 = reglasAplicables(m, i, j);
    while (!L1.isEmpty()) {
        Regla R = elegirRegla(L1);
        laberinto(m, i: R.fil, j: R.col, iF, jF, paso + 1);
        m[R.fil][R.col] = 0;
    }
}
```

e) Algoritmo para mostrar todos los caminos posibles de mínima longitud desde una posición inicial a una posición final. Además, mostrar la cantidad de soluciones posibles.

```
public static int minPasos = Integer.MAX_VALUE;
public static LinkedList<int[][]> soluciones = new LinkedList<>();

public static void laberinto(int m[][], int i, int j, int iF, int jF, int paso) {
    m[i][j] = paso;
    if (i == iF && j == jF) {
        if (paso < minPasos) {
            minPasos = paso;
            soluciones.clear();
            soluciones.add(e: m);
            cant = 1;
        } else if (paso == minPasos) {
            soluciones.add(e: m);
            cant++;
        }
    }
    LinkedList<Regla> L1 = reglasAplicables(m, i, j);
    while (!L1.isEmpty()) {
        Regla R = elegirRegla(L1);
        laberinto(m, i: R.fil, j: R.col, iF, jF, paso + 1);
        m[R.fil][R.col] = 0;
    }
}
```

Ejecutar también los algoritmos para todos los incisos, incluyendo Atajos en la matriz. Implementar para moverse en sentido HORARIO y en Sentido ANTI HORARIO. Analizar los resultados, cantidad de soluciones y emitir un criterio analítico para cada una de ellas.

Para este algoritmo en cuestión el cambiar de sentido horario y antihorario se hace de una manera mucho mas sencilla y eficiente, en vez de cambiar todas las reglas dadas o en todo caso invertirlas, únicamente se cambia en el algoritmo principal la manera de acceder a ellas, de esta manera optimizando mejor nuestro tiempo

```
public static Regla elegirRegla(LinkedList<Regla> L1) {
    return L1.removeFirst();
}
public static Regla elegirReglaAntiHorario(LinkedList<Regla> L1) {
    return L1.removeLast();
}
```