# Auxiliar 3 - Diagramas de Estado, Backtracking y Recursión

Profesores: Nelson Baloian
Jeremy Barbay
Patricio Poblete
Auxiliares: Gabriel Flores, Sven Reisenegger
Juglar Díaz, Gabriel Norambuena
Cristóbal Muñoz

**P1. Problema**:

**P2. Chequear posición de las reinas**:

Implemente un método que chequee que una ubicación de las 8 reinas es correcta para el problema visto en clases. Recuerde que no pueden haber dos reinas en la misma fila o en la misma columna o en la misma diagonal.

La función *Valido* es la que comprueba las restricciones implícitas, realizando la prueba de fracaso:

```
PROCEDURE Valido(k:CARDINAL):BOOLEAN;
(* comprueba si el vector solucion X construido hasta el paso k
   es k-prometedor, es decir, si la reina puede situarse en la
   columna k *)
   VAR i: CARDINAL;
BEGIN
   FOR i:=1 TO k-1 DO
      IF (X[i]=X[k]) OR (ValAbs(X[i],X[k])=ValAbs(i,k)) THEN
         RETURN FALSE
      END
   END;
   RETURN TRUE
END Valido;
```

Utilizamos la funcion *ValAbs*(*x*,*y*), que es la que devuelve $|x - y|$:

```
PROCEDURE ValAbs(x,y:CARDINAL):CARDINAL;
BEGIN
   IF x>y THEN RETURN x-y ELSE RETURN y-x END;
END ValAbs;
```

```java
/* Java program to solve N Queen Problem using
   backtracking */
public class NQueenProblem
{
    final int N = 4;

    /* A utility function to print solution */
    void printSolution(int board[][])
    {
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
                System.out.print(" " + board[i][j]
                                 + " ");
            System.out.println();
        }
    }

    /* A utility function to check if a queen can
       be placed on board[row][col]. Note that this
       function is called when "col" queens are already
       placeed in columns from 0 to col -1. So we need
       to check only left side for attacking queens */
    boolean isSafe(int board[][], int row, int col)
    {
        int i, j;

        /* Check this row on left side */
        for (i = 0; i < col; i++)
            if (board[row][i] == 1)
                return false;

        /* Check upper diagonal on left side */
        for (i=row, j=col; i>=0 && j>=0; i--, j--)
            if (board[i][j] == 1)
                return false;

        /* Check lower diagonal on left side */
        for (i=row, j=col; j>=0 && i<N; i++, j--)
            if (board[i][j] == 1)
                return false;

        return true;
    }
```

```
/* A recursive utility function to solve N
   Queen problem */
boolean solveNQUtil(int board[][], int col)
{
    /* base case: If all queens are placed
       then return true */
    if (col >= N)
        return true;

    /* Consider this column and try placing
       this queen in all rows one by one */
    for (int i = 0; i < N; i++)
    {
        /* Check if queen can be placed on
           board[i][col] */
        if (isSafe(board, i, col))
        {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;

            /* recur to place rest of the queens */
            if (solveNQUtil(board, col + 1) == true)
                return true;

            /* If placing queen in board[i][col]
               doesn't lead to a solution then
               remove queen from board[i][col] */
            board[i][col] = 0; // BACKTRACK
        }
    }

    /* If queen can not be place in any row in
       this colum col, then return false */
    return false;
}
```

```
boolean solveNQ()
{
    int board[][] = {{0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0}
    };

    if (solveNQUtil(board, 0) == false)
    {
        System.out.print("Solution does not exist");
        return false;
    }

    printSolution(board);
    return true;
}
    // driver program to test above function
    public static void main(String args[])
    {
        NQueenProblem Queen = new NQueenProblem();
        Queen.solveNQ();
    }
}
// This code is contributed by Abhishek Shankhadhar
```

**P3. Recorrido del caballo de ajedrez**:
El problema del caballo es un antiguo problema matemático en el que se pide que, teniendo una cuadrícula de n x n casillas y un caballo de ajedrez colocado en una posición cualquiera (x, y), el caballo pase por todas las casillas una sola vez. Lo que resulta en $n^2 - 1$ movimientos.

- Implemente una solución al problema del recorrido del caballo de ajedrez en un tablero de 8x8 como en el ajedrez comenzando en la posición 0x0 en una clase KnightTour con un método: `static boolean solveKT() además de métodos auxiliares.`

```java
// Java program for Knight Tour problem
class KnightTour {
    static int N = 8;

    /* A utility function to check if i,j are
       valid indexes for N*N chessboard */
    static boolean isSafe(int x, int y, int sol[][]) {
        return (x >= 0 && x < N && y >= 0 &&
                y < N && sol[x][y] == -1);
    }

    /* A utility function to print solution
       matrix sol[N][N] */
    static void printSolution(int sol[][]) {
        for (int x = 0; x < N; x++) {
            for (int y = 0; y < N; y++)
                System.out.print(sol[x][y] + " ");
            System.out.println();
        }
    }
```

```
static boolean solveKT() {
    int sol[][] = new int[8][8];

    /* Initialization of solution matrix */
    for (int x = 0; x < N; x++)
        for (int y = 0; y < N; y++)
            sol[x][y] = -1;

   /* xMove[] and yMove[] define next move of Knight.
      xMove[] is for next value of x coordinate
      yMove[] is for next value of y coordinate */
    int xMove[] = {2, 1, -1, -2, -2, -1, 1, 2};
    int yMove[] = {1, 2, 2, 1, -1, -2, -2, -1};

    // Since the Knight is initially at the first block
    sol[0][0] = 0;

    /* Start from 0,0 and explore all tours using
       solveKTUtil() */
    if (!solveKTUtil(0, 0, 1, sol, xMove, yMove)) {
        System.out.println("Solution does not exist");
        return false;
    } else
        printSolution(sol);

    return true;
}

/* A recursive utility function to solve Knight
   Tour problem */
static boolean solveKTUtil(int x, int y, int movei,
                           int sol[][], int xMove[],
                           int yMove[]) {
    int k, next_x, next_y;
    if (movei == N * N)
        return true;

    /* Try all next moves from the current coordinate
        x, y */
    for (k = 0; k < 8; k++) {
        next_x = x + xMove[k];
        next_y = y + yMove[k];
        if (isSafe(next_x, next_y, sol)) {
            sol[next_x][next_y] = movei;
            if (solveKTUtil(next_x, next_y, movei + 1,
                            sol, xMove, yMove))
                return true;
            else
                sol[next_x][next_y] = -1;// backtracking
        }
    }

    return false;
}
```

```
    /* Driver program to test above functions */
    public static void main(String args[]) {
        solveKT();
    }
}
```