Laboratorio Nro. 2: Fuerza Bruta (*Brute force* o *Exhaustive search*)

Daniel Alejandro Mesa Arango
Universidad EAFIT
Medellín, Colombia
damesaa@eafit.edu.co

Kevin Arley Parra Henao
Universidad EAFIT
Medellín, Colombia
kaparrah@eafit.edu.co

3) Simulacro de preguntas de sustentación de Proyectos

1. **Teniendo en cuenta lo anterior, respondan: Para resolver el problema de las N Reinas, fuera de fuerza bruta, ¿qué otras técnicas computacionales existen?**

- Recorrido en profundidad.
  Las soluciones del problema de las n reinas se pueden obtener explorando un árbol. Sin embargo, no generamos explícitamente el árbol para explorarlo después. Los nodos se van generando y abandonando en el transcurso de la exploración mediante un recorrido en profundidad.
- Resolución en paralelo
- Algoritmos genéticos.

2. **Tomen los tiempos de ejecución del programa implementado en el numeral 1.1 y completen la siguiente tabla. Si se demora más de 50 minutos, coloque "*se demora más de 50 minutos*", no sigan esperando, podría tomar siglos en dar la respuesta, literalmente. Además, realicen una gráfica con los datos de la tabla.**

Valor de N	Tiempo de ejecución
4	1 milisegundos
5	1 milisegundos
6	5 milisegundos
7	14 milisegundos
8	69 milisegundos
9	242 milisegundos
10	3652 milisegundos
11	Mas de 20 minutos
12	Mas de 20 minutos
13	Mas de 20 minutos
14	Mas de 20 minutos
15	Mas de 20 minutos
16	Mas de 20 minutos
17	Mas de 20 minutos
18	Mas de 20 minutos
19	Mas de 20 minutos
20	Mas de 20 minutos
21	Mas de 20 minutos

22	Mas de 20 minutos

23	Mas de 20 minutos
24	Mas de 20 minutos
25	Mas de 20 minutos
26	Mas de 20 minutos
27	Mas de 20 minutos
28	Mas de 20 minutos
29	Mas de 20 minutos
30	Mas de 20 minutos
31	Mas de 20 minutos
32	Mas de 20 minutos
N	**O ( )**

3. **Escriban una explicación entre 3 y 6 líneas de texto del código del ejercicio en línea del numeral 2.1. Digan cómo funciona, cómo está implementado y destaquen las estructuras de datos y algoritmos usados**

En primer lugar utilizamos la implementación ya hecha en clases anteriores, solo modificamos el algoritmo para que en la lectura guardemos en un arreglo las posiciones en donde no se deben poner reinas es decir donde están asteriscos, luego pasamos el número de reinas y el arreglo y cada vez que generamos un tablero verificamos en primer lugar que no se ataquen y en segundo que ninguna reina se posicione en el lugar del cajón malo es decir dónde está el asterisco, de acuerdo a lo anterior cuenta o no el tablero como valido, las permutaciones se calculan recursivamente y se guarda cada una en una lista, los tableros se representan en arreglos ya que es más eficiente que utilizar una matriz.

4. **Expliquen con sus propias palabras la estructura de datos que utilizan para resolver el problema del numeral 2.1 y cómo funciona el algoritmo**.

En primer lugar, utilizamos arreglos para representar los tableros, una matriz sígnica mas espacio del necesario, además guardamos en un ArrayList todas las permutaciones las cuales convertimos a enteros cada carácter para luego verificar si la permutación cumple todas las condiciones, nos pareció mas eficiente gracias a que nos permite acceder a ella en un tiempo constante, aunque sacrificamos un poco el tiempo en el que se agrega una nueva permutación.

5. **Calculen la complejidad del ejercicio en línea del numeral 2.1 y agréguenla al informe PDF**

```
private static ArrayList<String> permutations(String s) {
      ArrayList<String> r = new ArrayList<String>();      //c
      permutations("", s, r);  // n!
      return r; //c
   }
   //T(n) = c+c+n!
   //O(n!)

   private static void permutations(String pre, String pos, ArrayList<String> list) {
      if (pos.length() == 0) {
         list.add(pre);
      } else {
```

```
        for (int i = 0; i < pos.length(); ++i) {                    //c*n
            permutations(pre + pos.charAt(i), pos.substring(0, i) + pos.substring(i + 1), list);  //n*T(n-1)
        }
    }
}
//T(n) = c*n+ n*T(n-1)
//O(n!)

private static int nReinas(int numero, int [] tableroMalos) {
    if (numero != 0) {
        int resultado = 0;
        String original = "";
        int[] tablero = new int[numero];
        for (int i = 0; i < numero; i++) {            //n
            original = original + i;                  //c*n
        }
        ArrayList<String> todos = new ArrayList<String>();
        todos = permutations(original);                    //n!

        for (int j = 0; j < todos.size(); j++) {           //c*m
            String prueba = todos.get(j);                  //c*m
            for (int i = 0; i < prueba.length(); i++) {    //c*n*m
                tablero[i] = Integer.parseInt(prueba.charAt(i) + "");  //c*n*m
            }
            if (valido(tablero, tableroMalos) == true) {  //c*n^2*m
                resultado = resultado + 1;                // c*m
            }
        }
        return resultado;                                  //c
    } else {
        return -1;
    }
}

//T(n) = n+c*n+n!+c*m+c*m+c*n*m+c*n*m+c*n^2*m+c*m+c
//O(n!)

public static boolean valido(int[] tablero, int [] tableroMalos) {
    for (int i = 0; i < tablero.length; i++) {     //c*n
        for (int j = i + 1; j < tablero.length; j++) {     //c*n*n
            if (tablero[i] == tablero[j] || Math.abs(tablero[i] - tablero[j]) == Math.abs(i - j) || tablero[i] ==
tableroMalos[i]) //c*n^2
                return false;                 //c
            }
    }

    /*for(int j = 0; j < tablero.length; j++)
        {
            if(tablero[j] == tableroMalos[j])
                return false;
```

```
            }*/
    return true; //c
}
//T(n) = c*n+c*n*n+c+c*n^2+c
//O(n^2)


public static void main(String[] args)
{
    Scanner in  =  new Scanner(System.in);
    int numero = in.nextInt();
    int contador = 0;
    while(numero != 0)
        {
            contador++;
            int [] malos = new int[numero];
            for(int r = 0; r < numero; r++)
                malos[r] = -1;

            char auxiliar = ' ';
            String linea = "";
            for(int j = 0; j < numero; j++)
                {
                    linea = in.next();
                    for(int i = 0; i < linea.length(); i++) //n
                        {

                            auxiliar = linea.charAt(i); //c*n
                            if(auxiliar == '*')          //c*n
                                {

                                    malos[j] = i;      //c*n
                                }
                        }

                }
            System.out.println("Case "+ contador+ ": " + nReinas(numero, malos)); //n!
            numero = in.nextInt();
        }
}
//T(n) = n+c*n+c*n+c*n+n!
//O(n!)
```

Nota: en el main no tomamos en cuenta el ciclo para pedir los datos, ya que consideramos que no hace parte del algoritmo como tal.

6. **Expliquen con sus palabras las variables (*qué es 'n', qué es 'm', etc.*) del cálculo de complejidad del numeral 3.5**
   En este caso "n" representa el tamaño del tablero que no es "n^2" porque utilizamos un arreglo para mejorar la eficiencia, en cambio "m" son todas las permutaciones que se derivaron del arreglo en este caso lo tratamos como cadena que luego convertimos a enteros para su manipulación.

4) Simulacro de Parcial

1.
   A. if(actual > maximo)
   B. n^2
2.
   A. ordenar(arr, k+1);
   B. O(n!)
3.
   1. return i-m;
   2. return n;
   3. O(n*m)

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co