

Convocatoria ordinaria – 18 de mayo de 2023 (Parte 2)

Apellidos, nombre _____ NIF: _____

Pregunta 1 (2,5 p.)

Estamos jugando un juego contra una computadora. La idea es distribuir nuestras N bolas rojas en 8 grupos de diferentes tamaños. La computadora hará lo mismo con sus M bolas azules. Es decir, creará otros 8 grupos de diferentes tamaños. Debemos asignar a cada uno de nuestros grupos de color rojo un grupo de color azul. Ganaremos un punto cada vez que el número de bolas rojas sea mayor que el número de bolas azules en una tarea. Diseñe lo siguiente:

- a) (1 p.) Un algoritmo codicioso para calcular la solución óptima (maximizar el número total de puntos obtenidos). La complejidad temporal del algoritmo debe ser lo más baja posible.
 1. Ordenamos los 8 grupos rojos en orden ascendente según el número de bolas en cada grupo.
 2. Ordenamos los 8 grupos azules en orden ascendente según el número de bolas en cada grupo.
 3. Para todos los grupos rojos (indexados por i):
 - a. Si el grupo rojo \geq grupo azul \rightarrow asigna el grupo rojo al siguiente grupo azul en orden ascendente (el más pequeño de los que aún no han sido procesados, indexados por j).
 - b. Si el grupo rojo $<$ grupo azul \rightarrow asignar el grupo rojo al siguiente grupo azul en orden descendente (el mayor de los aún no procesados, señalado por k).

En Java, obviamos la parte 3b por simplicidad, pero para incluirla sólo habría que añadir otro índice desde el último elemento y un else que lo decrementara:

```
public class JuegoGrpBolas {
    private static int victorias= 0;
    private static int[] grpsRojo= {9, 48, 45, 10, 42, 43, 19, 28};
    private static int[] grpsAzul= {33,25, 69, 29, 76, 42, 23, 4};

    // Con escribir este método era suficiente para el examen
    public static void asignarGrps(int[] rojos, int[] azules) {
        Arrays.sort(rojos);
        Arrays.sort(azules);

        int iAzul= 0;
        for (int iRojo= 0; iRojo<rojos.length; iRojo++)
            if (rojos[iRojo]>azules[iAzul]) {
                victorias++;
                iAzul++;
            }

    }

    public static void main(String[] args) {
        asignarGrps(grpsRojo,grpsAzul);
        System.out.println("Victorias: "+victorias);
    }
}
```

}

b) (0,5 p.) ¿Cuál es la complejidad del algoritmo diseñado?

Dado que ordenamos ambos grupos, tenemos $O(n \log n)$ usando cualquier algoritmo de clasificación rápido. Además, iteramos secuencialmente a través de los diferentes grupos ($O(n)$), por lo que la complejidad final es $O(n \log n)$.

c) (1 p.) Haz una traza del algoritmo para el siguiente caso ($n=8$) para permitirnos maximizar el número de puntos. Después de eso, ¿cuántos puntos conseguimos contra el ordenador?

- Tenemos los siguientes tamaños para los grupos rojos = 9, 48, 45, 10, 42, 43, 19, 28
- Tenemos los siguientes tamaños para los grupos azules = 33, 25, 69, 29, 76, 42, 23, 4

Solución:

9	i	4	9
10	23	28	+
19	25	42	+
28	29	43	+
42	33	45	+
43	42	48	+
45	69	19	-
48	76	10	-
	k		

Con esta estrategia obtenemos **6 puntos** con los grupos rojos

Pregunta 2 (2,5 p.)

Dado un conjunto de números enteros positivos, se pide encontrar si es posible seleccionar un subconjunto de elementos cuya suma sea igual a un valor objetivo dado.

Formalmente, se da un conjunto de números enteros positivos $\{a_1, a_2, \dots, a_n\}$ y un valor objetivo S . Se debe determinar si existe un subconjunto de estos números cuya suma sea igual a S .

Por ejemplo, considere el siguiente conjunto de números: $\{3, 7, 2, 9\}$ y el valor objetivo $S = 13$. En este caso, es posible seleccionar los números 3, 7 y 2 para obtener una suma de 12, que es menor que S . Sin embargo, no hay forma de obtener una suma exacta de 13, por lo que la respuesta sería False.

a) (1,5 p.) Implementar una solución eficiente en Java utilizando programación dinámica. La signatura del método será la siguiente:

```
public static boolean subsetSum(int[] numbers, int target)
```

Solución:

```
public static boolean subsetSum(int[] numbers, int target) {
    int n = numbers.length;
    boolean[][] table = new boolean[n + 1][target + 1];

    for (int i = 0; i <= n; i++) {
        table[i][0] = true;
    }
}
```

```

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= target; j++) {
        if (numbers[i - 1] > j) {
            table[i][j] = table[i - 1][j];
        } else {
            table[i][j] = table[i - 1][j] || table[i - 1][j - numbers[i - 1]];
        }
    }
}

return table[n][target];
}

```

- b) (1 p.) Representa los valores intermedios que genera el algoritmo para el caso descrito anteriormente números= {3, 7, 2, 9} para obtener un objetivo de S= 13

Solución:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F
1 (3)	T	F	F	T	F	F	F	F	F	F	F	F	F	F
2 (7)	T	F	F	T	F	F	F	T	F	F	T	F	F	F
3 (2)	T	F	T	T	F	T	F	T	F	T	T	F	T	F
4 (9)	T	F	T	T	F	T	F	T	F	T	T	T	T	F

Pregunta 3 (2,5 p.)

El Sudoku es un rompecabezas de colocación de números basado en la lógica. En el Sudoku clásico, el objetivo es llenar una cuadrícula de 9×9 con dígitos para que cada columna, cada fila y cada una de las nueve subcuadrículas de 3×3 que componen la cuadrícula (también llamadas "cajas", "bloques" o "regiones") contienen todos los dígitos del 1 al 9. El colocador de rompecabezas proporciona una cuadrícula parcialmente completa, que para un rompecabezas bien planteado tiene una única solución.

Por ejemplo, el siguiente podría ser el estado inicial de un tablero, con estos números que permanecerán fijos y a la derecha una posible solución que cumple con las restricciones (cada columna, cada fila y cada una de las nueve subcuadrículas de 3×3 que componen la cuadrícula contienen todos los dígitos del 1 al 9):

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Completa el código a continuación para resolver este problema utilizando un algoritmo de backtracking.

Solución:

```
public class Sudoku {
    public final int N = 9; //size of the Sudoku
    public int[][] board; //board for the Sudoku puzzle
    public boolean wasFound; //whether a solution was found (true or false)

    public static void main(String arg[]) {
        Sudoku sudoku = new Sudoku();
        sudoku.backtracking();
        if (!sudoku.wasFound) System.out.println("THERE IS NO SOLUTION");
    }

    public Sudoku() {
        String name = "FILE.txt"; //name of the file
        board = new int[N][N];
        readFile(name);
        System.out.println("THE SUDOKU TO BE SOLVED IS:");
        writeBoard();
    }

    //method to load the initial state of the game in "board". It is already
    implemented.
    private void readFile(String name) { }

    //method to print the "board". It is already implemented.
    private void writeBoard() { }

    public void backtracking() {
        wasFound = false;
        int[] zero = new int[2]; // almacena fila y columna del siguiente hueco
        zero = findZero(0, -1); // busca el siguiente hueco comenzando en fila 0,
        columna -1
        backtracking(zero[0], zero[1]);
    }

    private void backtracking(int i, int j) {
        if (i == N) { //we found a solution (AND FINISH)
            wasFound = true;
            System.out.println("SOLUTION FOUND:");
            writeBoard();
        }
        else
            for (int k=1; k<=N; k++) {
                if (!wasFound && row(i,k) && column(j,k) && region(i,j,k)) {
                    board[i][j] = k; //mark

                    int[] zero = new int[2];
                    zero = findZero(i,j);
                    backtracking(zero[0], zero[1]);

                    board[i][j] = 0; //unmark
                }
            }
    }

    //look for the next position with 0 (i.e., that it is empty)
    private int[] findZero(int i, int j) { } // devuelve [n, n] si no quedan huecos

    //to check if the row i is valid for the number k
    private boolean row(int i, int k) { }
```

```
//to check if the column j is valid for the number k
private boolean column(int j, int k) {    }

//to check if the region is valid for the number k
private boolean region(int i, int j, int k) {
    boolean b = true;
    int startingI = i-i%3; //if for example i=7 => startingI = 6
    int startingJ = j-j%3; //if for example j=1 => startingJ = 0
    for (int posI=startingI; posI<startingI+3; posI++)
        for (int posJ=startingJ; posJ<startingJ+3; posJ++)
            if (board[posI][posJ] == k) b=false;
    return b;
}
```

Pregunta 4 (2,5 p.)

Contesta de forma breve (tres líneas como máximo por respuesta). Sea el problema de la asignación de tareas a agentes visto en clase.

- Si tenemos 6 tareas y 6 agentes. Al resolver el problema con backtracking ¿Cuántos estados hijos tendrá el estado inicial? ¿Qué altura tendrá el árbol de estados?
- Para este mismo caso ¿cuál sería el número de total de estados generado?
- ¿Qué complejidad tendría este algoritmo?
- ¿Qué diferencia tendrá el árbol desarrollado con backtracking y con ramifica y poda?
- ¿Qué estados hijos tiene el estado (A-3, B-5, C-1)
- ¿Qué mecanismo utiliza Ramifica y Poda para seleccionar el siguiente nodo a desarrollar?
- ¿Qué significado tiene el valor del heurístico de ramificación en un nodo hoja?
- ¿Qué ocurre en el desarrollo del árbol si el heurístico de ramificación devuelve el mismo valor para todos los estados?
- ¿Si el heurístico de ramificación tiene poca calidad (como por ejemplo, únicamente contar el número de tareas que faltan por asignar), llegaríamos a encontrar una solución óptima a un problema?
- ¿Qué misión tiene el heurístico de poda en un problema en el que buscamos la primera solución válida?

Soluciones:

a) El estado inicial tendrá **6 estados hijos**, uno para cada tarea asignada a un agente. El árbol de estados tendrá **una altura de 6**, correspondiente al número de tareas.

b) El número total de estados generados será el número de ramas del árbol completo. En el **último nivel habrá 6! (factorial de 6) = 720**. Pero hay que sumar los nodos de niveles anteriores $6+6*5+6*5*4+6*5*4*3+6*5*4*3*2+6!= 6+30+120+360+720+720=$ **1956**

c) La complejidad de este algoritmo de backtracking sería **$O(n!)$** , donde n es el número de tareas o agentes en el problema.

d) El árbol desarrollado con **backtracking** **explorará todos los caminos posibles sin realizar podas** y en **profundidad**. Mientras que el árbol desarrollado con **ramifica y poda** utiliza un **desarrollo**

en anchura para cada nodo y realizará podas para evitar explorar subárboles que no mejoran la solución actual.

e) Los estados hijos de (A-3, B-5, C-1) son: (A-3, B-5, C-1, D-2), (A-3, B-5, C-1, D-4), (A-3, B-5, C-1, D-6)

f) Ramifica y Poda utiliza una función de evaluación (heurística) para seleccionar el siguiente nodo a desarrollar. Elige aquel nodo que tenga la menor puntuación de acuerdo con la función de evaluación. Para hacer esta elección se basa en una cola de prioridad de la que extrae el siguiente estado.

g) El valor de una solución del problema, que será la nueva cota de poda para seguir explorando.

h) Si el heurístico de ramificación devuelve el mismo valor para todos los estados, el algoritmo funcionaría como un recorrido en anchura puro, porque la cola de prioridad funciona como una cola FIFO. Además, una vez encontrada una solución, el resto de nodos se podará porque tienen el mismo valor heurístico.

i) Sí, siempre encuentra una solución óptima, es posible que el algoritmo tarde mucho tiempo en encontrar una solución óptima. La calidad del heurístico influye en la eficiencia. No obstante, no impide encontrar una solución óptima.

j) El heurístico de poda no tendría cabida en ese escenario, ya que la poda se realiza siempre tras haber encontrado una solución válida, sin embargo aquí no hay que buscar más soluciones.