Control 2 – 24 de abril de 2017

Apellidos, nombre _____ NIF: _____

Pregunta 1 (4 p.)

Tenemos información privilegiada sobre diferentes inversiones que pueden ofrecernos muy buenos beneficios. Sabiendo que el coste de las inversiones es alto (no nos dejan invertir sólo una parte de su coste), queremos crear un algoritmo que nos permita decidir qué inversión realizar, teniendo en cuenta que sólo tenemos 10000€ para invertir y hay varias opciones que pueden resultarnos interesantes. A continuación, se muestra una tabla con los costes en euros de las inversiones y los posibles beneficios que obtendremos.

	Inversión A	Inversión B	Inversión C	Inversión D
Coste	3000	4000	6000	2000
Posible beneficio	1000	2000	3000	1000

a) (2 puntos) Diseña un algoritmo mediante programación dinámica (sin utilizar ningún lenguaje de programación ni pseudocódigo) e indica cuál sería el beneficio máximo que podríamos obtener con nuestros 10000€ ¿Y con 5000€? ¿Y con 8000€?

Solución:

Esto es análogo al problema de la mochila (sin fragmentación). Nos indican que empleemos la técnica de **programación dinámica**. Esta técnica se basa en una función recursiva que en realidad se realiza de forma iterativa rellenando las celdas de la tabla correspondiente. Para calcularlo desarrollamos la siguiente tabla (filas, posibles inversiones, columnas, gasto en la inversión):

Inversión/Coste	0	1	2	3	4	5	6	7	8	9	10
A	0	0	0	1	1	1	1	1	1	1	1
B	0	0	0	1	2	2	2	3	3	3	3
C	0	0	0	1	2	2	3	3	3	4	5
D	0	0	1	1	2	**2**	3	3	**4**	4	**5**

El máximo beneficio sería 5000€.
Podríamos obtener:

- Con 10000€ → 5000€ de beneficio
- Con 5000€ → 2000€ de beneficio
- Con 8000€ → 4000€ de beneficio

b) (0,5 puntos) ¿Cuál sería la complejidad del algoritmo suponiendo que queremos implementarlo de forma genérica?

O(número_inversiones * beneficio_máximo)

c) (0,5 puntos) ¿Será la solución a este problema (para otros casos) siempre óptima utilizando programación dinámica?

Sí, el algoritmo por programación dinámica garantiza una solución óptima en cualquier caso.

d) (0,5 puntos) Si, en lugar de utilizar programación dinámica, utilizásemos un algoritmo voraz, ¿tendríamos siempre una solución óptima?

No, el algoritmo voraz no garantiza una solución óptima si no se pueden fragmentar las inversiones ya que no existe heurístico óptimo con estos requisitos.

e) (0,5 puntos) Si, en lugar de utilizar programación dinámica, utilizásemos Backtracking, ¿tendríamos siempre una solución óptima?

Si, backtracking buscará todas las opciones posibles hasta encontrar una solución óptima.

Pregunta 2 (3 p.)

Las nuevas instalaciones olímpicas de Tokio 2020 necesitan un nuevo tendido eléctrico que les proporcione la energía suficiente para su funcionamiento. Para optimizar la longitud de éste tendido los ingenieros informáticos japoneses han desarrollado una aplicación con el algoritmo de Kruskal. El cual, dado un grafo conexo no dirigido, donde cada arista posee una longitud no negativa, permite calcular el árbol de recubrimiento mínimo.

En el algoritmo de Kruskal se elige una arista del grafo de entre las de menor peso, se añaden paulatinamente las aristas con menor peso siempre que estas no formen un ciclo con las otras ya incorporadas, el proceso termina cuando se han seleccionado n-1 aristas.

a) (1 punto) Explicar, en función de sus características, qué tipo de algoritmo es este.

Se basa en una función heurística que guía al algoritmo en la selección del nodo adecuado. Esto evita probar un gran número de posibilidades para crear el árbol de recubrimiento que harían el proceso mucho más lento.

Se elige el siguiente estado con información local: la arista más corta que conecta el grafo. Esta decisión no se revoca nunca, no hay vuelta atrás.

b) (0,5 puntos) Escribir pseudocódigo para la función heurística.

En el heurístico definimos de qué forma seleccionamos la próxima arista del conjunto total de aristas, la repetición de este heurístico nos debería llevar al árbol de recubrimiento con el coste mínimo.

1. Seleccionar la arista de menor peso entre las que quedan.
2. Comprobar que no forme ciclos con las ya seleccionadas.

c) (1,5 puntos) Completar el código Java del método (sobre esta hoja) que permita obtener la solución a este problema mediante el algoritmo descrito. Para simplificar se

proporcionan varios métodos auxiliares en clase Grafo para trabajar con los nodos del grafo. La puntación máxima se obtendrá si se consigue una complejidad de O(nlogn), suponiendo que las operaciones auxiliares tienen como complejidad máxima O(log n).

```
public class Grafo {
	class Arista implements Comparable<Arista> {
		int origen, destino;		// nodos de origen y destino
		int peso;			// peso de la arista
		public Arista(int origen, int destino, int peso) {…}
		@Override
		public int compareTo(Arista a2) {…}
	}

	private int numNodos;			// nodos 0 .. numNodos -1
	private int [][] pesos;			// matriz pesos simétrica

	public Grafo() {…	}	// constructor
	// devuelve el número de nodos del grafo
	public int getNumNodos() {…}
	// devuelve todas las aristas del grafo
	public ArrayList<Arista> getTodasAristas() {… }
	// devuelve true si añadiendo Arista a al árobl forma algún ciclo
	public boolean formaCiclo(Arista a, ArrayList<Arista> arbol) {…	}


	public ArrayList<Arista> kruskal(Grafo grafo)
	{
		int n= getNumNodos();
		ArrayList<Arista> arbolRecubrimiento;

		// recupera todas las aristas del grafo
		ArrayList<Arista> aristas= getTodasAristas();
		int numAristas= aristas.size();

		// ordena para no tener que recorrer el array buscando cada una
		Collections.sort(aristas);

		int i= 0;
		int iArista= 0;
		// mientras no hayamos seleccionado n-1 aristas
		while (i<n && iArista<numAristas) // recorre aristas ordenadas
		{
			// Heurístico:
			// Buscar la arista de menor peso
			Arista arista= aristas.get(iArista);
			// Comprueba que no forma ciclo con la que tenemos
			if (!formaCiclo(arista,arbolRecubrimiento))
			{
				arbolRecubrimiento.add(arista);  // Añade al árbol
				i++;
			}
			iArista++;
		}

		return arbolRecubrimiento;

	}
}
```

Pregunta 3 (3 p.)

Tenemos un laberinto representado por una matriz cuadrada (*n x n*), de enteros donde 0 (camino) significa que se puede pasar y 1 (muro) que no se puede pasar.

El punto de *inicio* en el laberinto estará situado siempre en una de las filas de la primera columna y estará representado por una coordenada (Ini_x, Ini_y); y el punto de destino estará situado en la última columna en una de sus filas y estará representado por (des_x, des_y).

Los movimientos posibles son: arriba, abajo, izquierda y derecha. Debemos marcar el camino que se sigue desde el inicio al destino en las casillas. Al final, debe aparecer si se encontró solución o no, y si se encuentra mostrar el número de pasos realizados desde origen a destino.

Escribir el programa en Java, que implemente mediante backtracking la solución al problema. Rellenando los huecos en el siguiente código dado:

```
public class LaberintoUna {
    static int n;                //tamaño del laberinto (n*n)
    static int[][] lab;          //representación de caminos y muros
    static boolean haySolucion;      //se encontró una solución
      // arriba, abajo, izquierda, derecha
    static int[] movx= {0,  0,  -1, 1};      // desplazamientos x
    static int[] movy= {-1, 1,  0,  0};      // desplazamientos y

    static int inix;             //coordenada x de la posición inicial
    static int iniy;             //coordenada y de la posición inicial

    static int desx;             //coordenada x de la posición destino
    static int desy;             //coordenada y de la posición destino

    static void backtracking(int x, int y, int pasos) {
        if ((x == desx) && (y == desy) && (!haySolucion)) {
             //encontramos una solución y terminamos
            System.out.println("SOLUCIÓN ENCONTRADA CON " + pasos + " PASOS");
            haySolucion = true; //finalizamos el proceso
        }
        else
        {
            for (int k= 0; k<4; k++)
            {
                int u= x+movx[k];
                int v= y+movy[k];

                if (!haySolucion && u>=0 && u<=n-1 && v>=0 && v<=n-1 &&
                        lab[u][v]==0)
                {
                    lab[u][v] = 2; //marcar la nueva posición
                    backtracking(u, v, pasos+1);
                    lab[u][v] = 0; //desmarcar
                }
            }
        }
    }

    public static void main(String arg[]) {
       …
        inix= 0; iniy= 0; desx= n-1; desy= n-1;//damos valores a inicio y destino
        haySolucion = false; //iniciamos la variable solución encontrada
        lab[inix][iniy] = 2; //Marcamos inicio del camino: 2 es camino
```

```
        backtracking(inix, iniy, 0);
        if (!haySolucion) System.out.println("NO HAY SOLUCIм");
    }
```

Realmente la variable pasos no sería imprescindible, salvo por el requisito de indicar el número de pasos, ya que en este problema marcamos el camino con un único número.

b) (1 punto) Qué cambios habría que realizar para devolver el camino más corto para alcanzar la salida. Indícalo modificando el código Java del apartado anterior.

```
public class LaberintoOptimo {
        static int n;                           //tamaño del laberinto (n*n)
        static int[][] lab;             //representación de caminos y muros
        static boolean haySolucion;     //se encontró una solución
        // arriba, abajo, izquierda, derecha
        static int[] movx= {0,0,        -1,     1};             // desplazamientos x
        static int[] movy= {-1,         1,      0,      0};             // desplazamientos y

        static int posInicial;                  //número de casilla inicial (origen)
        static int inix;                        //coordenada x de la posición inicial
        static int iniy;                        //coordenada y de la posición inicial

        static int posFinal;            //número de casilla final (objetivo)
        static int desx;                        //coordenada x de la posición destino
        static int desy;                        //coordenada y de la posición destino

        static int[][] mejorSol;        // Guarda la mejor solución hasta el momento
        static int mejorPasos;// número de pasos de la mejor solución hasta el momento

        /**
         * Método Backtracking
         * @param x Coordenada actual x
         * @param y Coordenada actual y
         * @param pasos Número de pasos (pasos realizados por el animal a través del
camino)
         */
        static void backtracking(int x, int y, int pasos) {
                if ((x == desx) && (y == desy) && pasos<mejorPasos) { //encontramos una
solución mejor
                        System.out.println("Solución encontrada");
                        System.out.println(" con " + pasos + " pasos");
                        escribirLab(lab);
                        //haySolucion = true; //finalizamos el proceso
                        mejorPasos= pasos;
                        mejorSol= lab.clone();
                }
                else
                {
                        for (int k= 0; k<4; k++)
                        {
                                int u= x+movx[k];
                                int v= y+movy[k];

                                if (/*!haySolucion &&*/ u>=0 && u<=n-1 && v>=0 && v<=n-1 &&
                                                lab[u][v]==0 && pasos<mejorPasos)
                                {
                                        lab[u][v] = 2; //marcar la nueva posición
                                        backtracking(u, v, pasos+1);
                                        lab[u][v] = 0; //desmarcar
                                }
                        }
                }
        }

        public static void main(String arg[]) {

                …
                inix= 0; iniy= 0; desx= n-1; desy= n-1;//damos valores a inicio y destino
                haySolucion = false; //iniciamos la variable solución encontrada

                System.out.println("El laberinto inicial es el siguiente:");
```

```
                escribirLab(lab);
                System.out.println("El objetivo es ir desde la posición " + posInicial +
"("+inix+","+iniy+") a la posición " + posFinal + "("+desx+","+desy+")\n");

                haySolucion = false; //no tenemos una solución en este punto
                lab[inix][iniy] = 2; //posición inicial del animal. El número 2
representa el camino seguido por el animal
                mejorPasos= n*n;

                backtracking(inix, iniy, 0);
//              if (!haySolucion) System.out.println("** No hay solución **");
                if (mejorPasos==n*n) System.out.println("** No hay solución **");
        }
```