UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA	**Programación III** Códigos de asignatura: Sistemas: **402048**; Gestión: **41204-** **Prueba Presencial**	**Primera Semana** Enero de 2003 Duración: **2 horas** Material permitido: **NINGUNO**

Apellidos: _____ **DNI:** _____

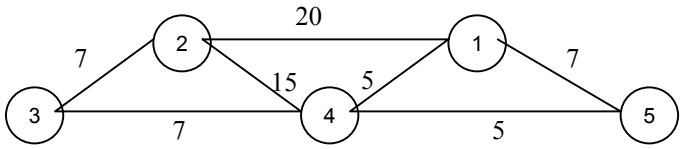
Nombre: _____ **Centro Asociado en el que entregó la práctica:** _____

Cuestión 1 (1 puntos). ¿Para qué se pueden utilizar montículos en el algoritmo de Kruskal? ¿Qué mejoras introduce en términos de complejidad?

Solución:

Brassard, página 220.

Cuestión 2 (2 punto). Dado el grafo de la figura, aplicar el algoritmo de Dijkstra para hallar los caminos más cortos desde el nodo 1 hasta cada uno de los otros nodos, indicando en cada paso: nodos seleccionados, nodos no seleccionados, vector de distancias y vector de nodos precedentes.



Solución:

			Distancias desde 1				Nodo precedente			
Paso	Nodos seleccionados	Nodos no seleccionados	2	3	4	5	2	3	4	5
Inicialización	1	2, 3, 4, 5	20	∞	5	7	1	1	1	1
1	1, 4	2, 3, 5	20	12	5	7	1	4	1	1
2	1, 4, 5	2, 3	20	12	5	7	1	4	1	1
3	1, 4, 5, 3	2	19	12	5	7	3	4	1	1

Cuestión 3 (2 puntos). Se desea implementar una función para *desencriptar* un mensaje numérico. La función *desencriptar* recibe tres enteros: el mensaje cifrado *c*, la clave privada *s* y la clave pública *z*; y devuelve el mensaje original *a*. El mensaje original se recompone con la fórmula:

$$a := c^s \bmod z$$

Sabiendo que no se dispone del operador de potencia, implementar la función utilizando el esquema divide y vencerás.

La resolución del problema debe incluir, por este orden:

1. Elección razonada del esquema algorítmico
2. Descripción del esquema usado e identificación con el problema
3. Estructuras de datos
4. Algoritmo completo a partir del refinamiento del esquema general
5. Estudio del coste

Según el esquema elegido hay que especificar, además:

- **Voraz:** demostración de optimalidad
- **Divide y Vencerás:** preorden bien fundado
- **Exploración en grafos:** descripción del árbol de búsqueda asociado

---

Solución:

a)
desencriptar (c, s, z: entero)
{
	devolver expoDV(c,s) mod z
}

expoDV en Brassard, página 276.

b)
implementarlo como *expomod,* Brassard, página 279.

Problema (5 puntos). Teseo se adentra en el laberinto en busca de un minotauro que no sabe dónde está. Se trata de implementar una función *ariadna* que le ayude a encontrar el minotauro y a salir después del laberinto. El laberinto debe representarse como una matriz de entrada a la función cuyas casillas contienen uno de los siguientes tres valores: 0 para "camino libre", 1 para "pared" (no se puede ocupar) y 2 para "minotauro". Teseo sale de la casilla (1,1) y debe encontrar la casilla ocupada por el minotauro. En cada punto, Teseo puede tomar la dirección Norte, Sur, Este u Oeste siempre que no haya una pared. La función *ariadna* debe devolver la secuencia de casillas que componen el camino de regreso desde la casilla ocupada por el minotauro hasta la casilla (1,1).

Solución:

5.1 Elección razonada del esquema algorítmico

Como no se indica nada al respecto de la distancia entre casillas adyacentes, y ya que se sugiere utilizar únicamente una matriz, es lícito suponer que la distancia entre casillas adyacentes es siempre la misma (1, sin pérdidad de generalidad). Por otra parte, no se exige hallar el camino más corto entre la entrada y el minotauro, sino que el enunciado sugiere, en todo caso, que el algoritmo tarde lo menos posible en dar una de las posibles soluciones (y ayudar a salir a Teseo cuanto antes).

Tras estas consideraciones previas ya es posible elegir el esquema algorítmico más adecuado. El tablero puede verse como un grafo en el que los nodos son las casillas y en el que como máximo surgen cuatro aristas (N, S, E, O). Todas las aristas tienen el mismo valor asociado (por ejemplo, 1).

En primer lugar, el algoritmo de Dijkstra queda descartado. No se pide el camino más corto y si se hiciera, las particularidades del problema hacen que el camino más corto coincida con el camino de menos nodos y, por tanto, una exploración en anchura tendrá un coste menor: siempre que no se visiten nodos ya explorados, como mucho se recorrerá todo el tablero una vez (coste lineal con respecto al número de nodos versus coste cuadrático para Dijkstra).

En segundo lugar, es previsible esperar que el minotauro no esté cerca de la entrada (estará en un nivel profundo del árbol de búsqueda) por lo que los posibles caminos solución serán largos. Como no es necesario encontrar el camino más corto, sino encontrar un camino lo antes posible,

una búsqueda en profundidad resulta más adecuada que una búsqueda en anchura. En el peor de los casos en ambas habrá que recorrer todo el tablero una vez, pero ya que buscamos un nodo profundo, se puede esperar que en media una búsqueda en profundidad requiera explorar menos nodos que una búsqueda en anchura.

Si se supone que el laberinto es infinito entonces una búsqueda en profundidad no sería adecuada porque no garantiza que se pueda encontrar una solución. En este enunciado se puede presuponer que el laberinto es finito.

En tercer lugar, es posible que una casilla no tenga salida por lo que es necesario habilitar un mecanismo de retroceso.

Por último, es necesario que no se exploren por segunda vez casillas ya exploradas anteriormente.

Por estos motivos, se ha elegido el esquema de vuelta atrás.

5.2. Descripción del esquema usado

Vamos a utilizar el esquema de vuelta atrás modificado para que la búsqueda se detenga en la primera solución y para que devuelva la secuencia de ensayos que han llevado a la solución en orden inverso (es decir, la secuencia de casillas desde el minotauro hasta la salida).

```
fun vuelta-atrás (ensayo) dev (es_solución, solución)

    si válido (ensayo) entonces
        solución ← crear_lista();
        solución ← añadir (solución, ensayo);
        devolver (verdadero, solución);
    si no
        hijos ← crear_lista();
        hijos ← compleciones (ensayo)
        es_solucion ← falso;
        mientras ¬ es_solución ∧ ¬ vacia (hijos)
            hijo ← primero (hijos)
            si cumple_poda (hijo) entonces
                (es_solución, solución) ← vuelta-atrás(hijo)
            fsi
        fmientras
        si es_solución entonces
            solución ← añadir (solución, ensayo);
        fsi
        devolver (es_solución, solución);
    fsi
ffun
```

5.3 Estructuras de datos

Para almacenar las casillas bastará un registro de dos enteros x e y. Vamos a utilizar una lista de casillas para almacenar la solución y otra para las compleciones. Para llevar control de los nodos visitados bastará una matriz de igual tamaño que el laberinto pero de valores booleanos.

Será necesario implementar las funciones de lista:

- crear_lista
- vacia
- añadir
- primero

5.4 Algoritmo completo

Suponemos "*laberinto"* inicializado con la configuración del laberinto y "*visitados"* inicializado con todas las posiciones a falso.

```
tipoCasilla = registro
                    x,y: entero;
              fregistro

tipoLista

fun vuelta-atrás (   laberinto: vector [1..LARGO, 1..ANCHO] de entero;
                     casilla: tipoCasilla
                     visitados: vector [1..LARGO, 1..ANCHO] de booleano;
                  ) dev (es_solución: booleano; solución: tipoLista)
      visitados [casilla.x, casilla.y] ← verdadero;
      si laberinto[casilla.x, casilla.y] == 2 entonces
            solución ← crear_lista();
            solución ← añadir (solución, casilla);
            devolver (verdadero, solución);
      si no
            hijos ← crear_lista();
            hijos ← compleciones (laberinto, casilla)
            es_solución ← falso;
            mientras ¬ es_solución ∧ ¬ vacia (hijos)
                  hijo ← primero (hijos)
                  si ¬ visitados [hijo.x, hijo.y] entonces
                        (es_solución, solución) ← vuelta-atrás (laberinto,hijo,visitados);
                  fsi
            fmientras
            si es_solución entonces
                  solución ← añadir (solución, casilla);
            fsi
            devolver (es_solución, casilla);
      fsi
ffun
```

En el caso de encontrar al minotauro se detiene la exploración en profundidad y al deshacer las llamadas recursivas se van añadiendo a la solución las casillas que se han recorrido. Como se añaden al final de la lista, la primera será la del minotauro y la última la casilla (1,1), tal como pedía el enunciado.

La función *compleciones* comprobará que la casilla no es una pared y que no está fuera del laberinto

```
fun compleciones ( laberinto: vector [1..LARGO, 1..ANCHO] de entero;
                   casilla: tipoCasilla) dev tipoLista
    hijos ← crear_lista();
    si casilla.x+1 <= LARGO entonces
        si laberinto[casilla.x+1,casilla.y] <> 1 entonces
            casilla_aux.x=casilla.x+1;
            casilla_aux.y=casilla.y;
            hijos ← añadir (solución, casilla_aux);
        fsi
    fsi
    si casilla.x-1 >= 1 entonces
        si laberinto[casilla.x-1,casilla.y] <> 1 entonces
            casilla_aux.x=casilla.x-1;
            casilla_aux.y=casilla.y;
            hijos ← añadir (solución, casilla_aux);
        fsi
    fsi
    si casilla.y+1 <= ANCHO entonces
        si laberinto[casilla.x,casilla.y+1] <> 1 entonces
            casilla_aux.x=casilla.x;
            casilla_aux.y=casilla.y+1;
            hijos ← añadir (solución, casilla_aux);
        fsi
    fsi
    si casilla.y-1 >= 1 entonces
        si laberinto[casilla.x,casilla.y-1] <> 1 entonces
            casilla_aux.x=casilla.x;
            casilla_aux.y=casilla.y-1;
            hijos ← añadir (solución, casilla_aux);
        fsi
    fsi
ffun
```

5.5 Estudio del coste

Todas las operaciones son constantes salvo la llamada recursiva a vuelta-atrás. En cada nivel, pueden realizarse hasta 4 llamadas. Sin embargo, las llamadas no se realizan si la casilla ya ha sido visitada. Esto quiere decir que, en el caso peor, sólo se visitará una vez cada casilla. Como las operaciones para una casilla son de complejidad constante, la complejidad será O(ANCHO*LARGO), lineal con respecto al número de casillas.