SEPTIEMBRE 2012 – ORIGINAL (MODELO A)

1. Considérese el vector [10,7,7,4,3,1] que representa un montículo. ¿Cuál sería la representación resultante de insertar en este montículo el valor 11 usando la función flotar?

a. [11,7,7,4,3,1,10].

b. [10,7, 11,4,3, 1,7].

c. [10,7, 11,4,3, 1,7].

d. Ninguna de las opciones anteriores.

La representación resultante sería: {11,7,10,4,3,1,7}

Los residentes de una ciudad no quieren pavimentar todas sus calles (que son de doble sentido), sino sólo aquellas que les permitan ir de una intersección a otra cualquiera de la ciudad con comodidad. Quieren gastarse lo menos posible en la pavimentación, teniendo en cuenta que el coste es directamente proporcional a la longitud de las calles que hay que pavimentar. El alcalde querría saber qué calles tiene que pavimentar para gastarse lo menos posible. ¿Cuál de los siguientes esquemas es más eficiente de los que puedan resolver el problema correctamente?

a. Esquema voraz.

b. Esquema de programación dinámica.

c. Esquema de vuelta atrás.

d. Esquema de ramificación y poda.

3. Suponga un algoritmo que dadas dos listas de tamaños n y m: $A=a_1,...a_n$ y $B=b_1,...,b_m$ dé como resultado la lista de aquellos elementos que pertenecen a A pero no a B. Se asume que A y B no contienen elementos duplicados, pero no que éstos estén acotados. Indique cuál de los siguientes costes se ajusta más al del algoritmo que puede resolver el problema con menor coste:

a. O(n log m + m log m).

b. O(n log m + m^2),

c. Máx (O (n log n), O(m log m)).

d. O(nm).

4. Dado un grafo con n vértices y a aristas sobre el que se pueden realizar las siguientes operaciones:

- ***esAdyacente: comprueba si dos vértices son adyacentes.***
- ***borrarVértice: Borra el vértice especificado y todas sus aristas.***
- ***añadirVértice: Añade un nuevo vértice al grafo.***

¿Cuál es la complejidad de cada una de las operaciones anteriores según se utilice una matriz de adyacencia (MA) o una lista de adyacencia (LA)?

a)

	MA	LA
esAdyacente	O(1)	O(1)
borrarVértice	O(n)	O(n)
añadirVértice	O(n)	O(1)

b)

	MA	LA
esAdyacente	O(1)	O(n)
borrarVértice	O(1)	O(n+a)
añadirVértice	O(1)	O(1)

c)

	MA	LA
esAdyacente	O(1)	O(n)
borrarVértice	O(n)	O(n+a)
añadirVértice	O(n)	O(1)

d)

	MA	LA
esAdyacente	O(1)	O(1)
borrarVértice	O(n)	O(n)
añadirVértice	O(n)	O(n)

Respuesta C

Explicación: Tabla de Costes de las Funciones de Manipulación de Grafos de la página 17 del libro de texto base.

	Matriz de adyacencia	Lista de adyacencia
CrearGrafo	$O(1)$	$O(1)$
AñadirArista	$O(1)$	$O(1)$
AñadirVertice	$O(n)$	$O(1)$
BorrarArista	$O(1)$	$O(n)$
BorrarVertice	$O(n)$	$O(n+a)$
Adyacente?	$O(1)$	$O(n)$
Adyacentes	$O(n)$	$O(1)$
Etiqueta	$O(1)$	$O(n)$

5. Indique cuál de las siguientes afirmaciones es cierta:

a. La programación dinámica se puede emplear en muchos de los problemas que se resuelven utilizando el esquema divide y vencerás.

b. La programación dinámica tiene como principal ventaja que supone un decremento tanto en coste computacional como en espacio de almacenamiento.

c. El esquema de programación voraz se aplica a problemas de optimización en los que la solución se puede construir paso a paso comprobando, en cada paso, la secuencia de decisiones tomadas anteriormente.

d. El esquema de ramificación y poda es el más indicado a utilizar en el caso que se deseen encontrar todas las soluciones posibles a un problema dado.

6. En el recorrido mediante doble hashing, la función h'(x) debe cumplir:

a. Necesariamente debe ser h'(x) f:. O.

b. h'(x) debe tener la forma h'(x) = 2h(x) + e, con e constante.

c. Se debe cumplir que h'(x) < h(x) + e, con e constante.

d. Ninguna de las anteriores es cierta.

Explicación: Es obvia la respuesta A), ya que, si no, se trataría de una colisión constante.

PROBLEMA (4 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: O 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).

La resolución de este problema debe incluir, por este orden:

1. Elección del esquema más apropiado, el esquema general y explicación de su aplicación al problema (0,5 puntos).

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 pérdida 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 debido a que no se pide el camino más corto sino uno cualquiera.

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 uno cualquiera, una búsqueda en profundidad resulta más adecuada que una búsqueda en anchura ya que buscamos un nodo profundo, y se puede esperar que en media una búsqueda en profundidad requiera explorar menos nodos que una búsqueda en anchura.

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.

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
```

2. Descripción de las estructuras de datos necesarias (0.5 puntos solo si el punto 1 es correcto).

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
- vacía
- añadir
- primero

3. Algoritmo completo a partir del refinamiento del esquema general (2,5 puntos solo si el punto 1 es correcto). Si se trata del esquema voraz debe hacerse la demostración de optimalidad.

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, ensayo);
                fsi
                devolver (es_solución, solución);
        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
```

4. Estudio del coste del algoritmo desarrollado (0.5 puntos solo si el punto 1 es correcto).

El espacio de búsqueda del problema es un árbol en el que cada nodo da lugar como máximo a tres ramas correspondientes a las cuatro direcciones de búsqueda, menos la casilla de la que procede el recorrido. El número de niveles del árbol es el número de casillas del tablero, ANCHOxLARGO. Luego una cota superior es $3^{ANCHOxLARGO}$.