FEBRERO 2019 – SEMANA 1 (MODELO A)

Grado en Ingeniería Informática y Grado en Ingeniería en Tecnologías de la Información

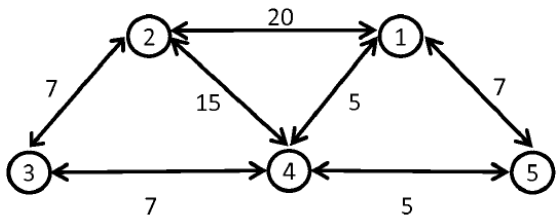
Normas de valoración del examen:

- La nota del examen representa el 80% de la valoración final de la asignatura (el 20% restante corresponde a las prácticas).
- Cada cuestión contestada correctamente vale 1 punto.
- Cada cuestión contestada incorrectamente baja la nota en 0.3 puntos.
- Debe obtenerse un mínimo de 3 puntos en las cuestiones para que el problema sea valorado (con 3 cuestiones correctas y alguna incorrecta el examen está suspenso).
- La nota total del examen debe ser al menos de 4.5 para aprobar.
- **Las cuestiones se responden en una hoja de lectura óptica**.

Examen tipo A:

Cuestiones:

1. Dado el siguiente grafo:



Indique el valor del vector de distancias *especial[]* en el paso del algoritmo de Dijkstra en el que se selecciona el nodo *v=3*, tomando como nodo origen el nodo 1:

(a) [19,12,5,7]
(b) [20,12,5,7]
(c) [20,19,5,7]
(d) Ninguna de las anteriores.

Nodos que han salido	Nodos que no han salido	Vector Distancia especial [ ]				Predecesores			
		2	3	4	5	2	3	4	5
1	2, 3, 4, 5	20	∞	**5**	7	1	1	1	1
1, 4	2, 3, 5	20	12	5	**7**	1	4	1	1
1, 4, 5	2, 3	20	**12**	5	7	1	4	1	1
1, 4, 5, 3	2	19	12	5	7	3	4	1	1

El vector de distancias especial cuando se selecciona v=3 es: [19, 12, 5, 7]

Respuesta A)

2. Un lejano país emite *n* sellos diferentes de valores naturales positivos $s_1, s_2, \ldots, s_n$. Se quiere enviar una carta y se sabe que la correspondiente tarifa postal es *T*. Se quiere saber de cuántas formas diferentes se puede franquear exactamente la carta, si el orden de los sellos no importa. ¿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.

Hay que calcular el número total de formas que hay de pagar la cantidad T con los sellos de n tipos (se consideran una cantidad ilimitada de sellos de cada valor). Se define la función:

formas (n, T) = número de formas de franquear T con n tipos de sellos

Consideramos los sellos de valor s_n. Podemos no poner ningún sello de este valor y calcular el número total de formas de franquear T con el resto de los sellos, o poner un sello de clase n y franquear el resto con cualquier tipo de sellos. Los sellos de tipo n solo los podemos considerar cuando $s_n \leq T$. El número total de formas será la suma de las formas de ambas posibilidades, puesto que estas son disjuntas y no hay más. Notemos que cada forma de franquear $T - s_n$ se extiende a una única forma de franquear T añadiendo el sello de valor s_n.

Planteamos la recurrencia en el caso general, cuando consideramos los sellos del 1 al i, y queremos franquear una cantidad j:

$$formas(i,j) = \begin{cases} formas(i-1,j) & \text{si } s_i > j \\ formas(i-1,j) + formas(i, j-s_i) & \text{si } s_i \leq j \end{cases}$$

donde $1 \leq i \leq n$ y $1 \leq j \leq T$.

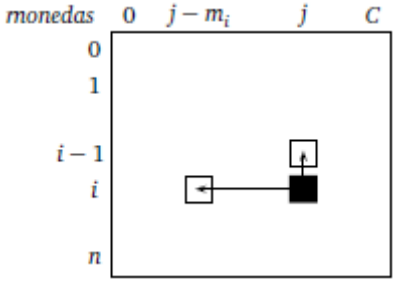
Los casos básicos se presentan cuando ya no nos quedan tipos de sellos que considerar (i = 0) y todavía nos queda alguna cantidad por franquear (j > 0), en cuyo caso no hay ninguna forma de franquear la cantidad:

formas(0, j) = 0 $1 \leq j \leq T$

O bien cuando la cantidad a franquear es 0, la única posibilidad es no poner ningún sello:

formas(i, 0) = 1 $0 \leq i \leq n$

Utilizamos una matriz formas[0..n, 0..T] para calcular los valores de la recurrencia. Para rellenar la posición (i, j) necesitamos la posición (i − 1, j) en la fila anterior y misma columna y alguna posición en la fila i y en una columna a la izquierda de j. Como en la figura siguiente:



Podríamos rellenar la matriz por filas de arriba abajo y de izquierda a derecha; pero ya que solo nos interesa el número de formas, podemos optimizar y utilizar solo un vector formas[0..T], que rellenamos de izquierda a derecha.

El algoritmo, de coste en Θ(nT) en tiempo y en Θ(T) en espacio adicional.

El mejor de los esquemas con los que se puede resolver este problema es el de programación dinámica.

Respuesta B)

3. Respecto a la estructura de datos grafo, ¿Cuál de las siguientes afirmaciones es **falsa**?

   (a) El máximo número de aristas en un grafo no dirigido de n vértices es n(n-1).
   (b) Los recorridos en profundidad o en anchura de un grafo conexo le asocian un árbol de recubrimiento.
   (c) La búsqueda en profundidad puede dar lugar a diferentes árboles o bosques de recubrimiento.
   (d) Dado un grafo conexo, un nodo *u* es un punto de articulación si al eliminar *u* y todas las aristas que inciden en él el grafo deja de ser conexo.

a) **Falso**. El máximo número de aristas en un grafo no dirigido de n vértices es *n * (n-1)/2*. En un grafo dirigido es de *n * (n-1)*.

b) **Verdadero**. Al asociarse un árbol de recubrimiento a los recorridos en profundidad o en anchura de un grafo conexo, las aristas del grafo se dividen en dos conjuntos: las que pertenecen al árbol, y las que no.

c) **Verdadero.** Si el árbol no es conexo, el recorrido en profundidad le asocia un bosque de árboles, uno por cada componente conexa del árbol, por lo que la búsqueda en profundidad puede dar lugar a diferentes árboles o bosques de recubrimiento, según el orden en el que se examinen los adyacentes. En un grafo dirigido las aristas del grafo que no están en el árbol o bosque asociado pueden ser de 3 tipos:

   - Las aristas que unen un nodo con uno de sus antecesores.
   - Las aristas que unen un nodo con uno de sus descendientes o sucesores.
   - Las aristas que unen un nodo con otro que no es antecesor ni descendiente suyo.

   En el caso de grafos no dirigidos, las aristas solo pueden unir un nodo con un antecesor o con un descendiente.

d) **Verdadero**. Dado un grafo conexo, un nodo *u* es un punto de articulación si al eliminar *u* y todas las aristas que inciden en el grafo, este deja de ser conexo. Un grafo conexo sin puntos de articulación se llama biconexo.

Respuesta A)

4. Con respecto al esquema divide y vencerás indique cuál de las siguientes afirmaciones es **verdadera**:

   (a) En algunos casos permite la paralelización de la resolución de los subproblemas del mismo tipo.
   (b) El coste de este tipo de algoritmos depende del número de subproblemas en los que se descompone el problema pero no de la reducción del tamaño en las sucesivas llamadas recursivas.
   (c) Siempre es necesario combinar las soluciones de los subproblemas.
   (d) Este esquema aplica el principio de deducción sobre los diversos ejemplares del problema.

   a) **Verdadero**. La estrategia de Divide y Vencerás es una técnica algorítmica que se basa en la descomposición de un problema en subproblemas de su mismo tipo, lo que permite disminuir la complejidad y en algunos casos, paralelizar la resolución de los mismos.

   b) **Falso**. El coste del algoritmo Divide y Vencerás depende de cuántas descomposiciones se realicen y del tipo de decrecimiento de las sucesivas llamadas recursivas a los subproblemas. En el caso de un decrecimiento del tamaño del problema en progresión geométrica, la ecuación de recurrencia se plantea como:

   $$T(n) = aT(n/b) + cn^k$$

   En el caso de decrecimientos en progresión aritmética:

   $$T(n) = aT(n-b) + cn^k$$

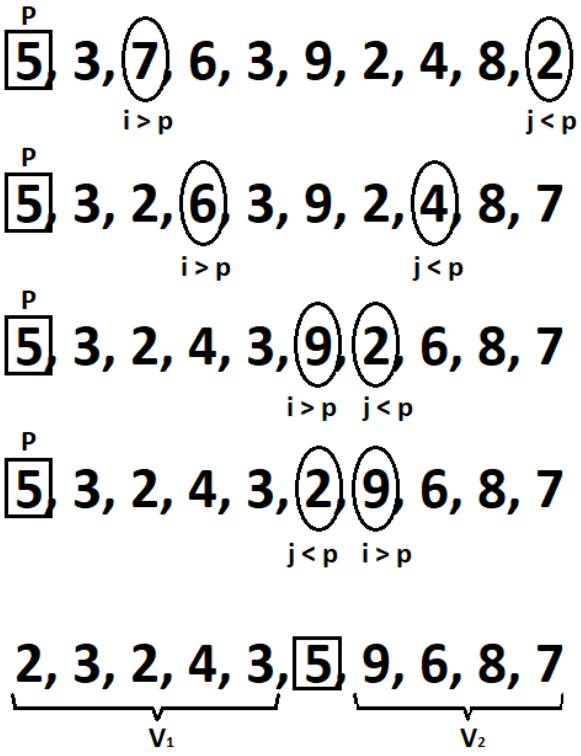
   c) **Falso**. La estrategia del algoritmo es la siguiente:
   - Descomposición del problema en subproblemas de su mismo tipo o naturaleza.
   - Resolución recursiva de los subproblemas.
   - Combinación, *si procede*, de las soluciones de los subproblemas.

   d) **Falso.** Este esquema aplica el principio de inducción sobre los diversos ejemplares del problema. De esta manera, supone solucionados los subproblemas, y utiliza las soluciones parciales de los mismos para componer la solución al problema. Para casos suficientemente pequeños, el esquema proporciona una solución trivial no recursiva, lo que equivale a la base de la inducción.

Respuesta A)

5. Considérese el vector [5,3,7,6,3,9,2,4,8,2]. Los vectores argumento de la primera invocación recursiva del algoritmo de quicksort, cuando se toma el elemento 5 de la primera posición como pivote, son:

   (a) [2,2,3,4,3] y [9,6,7,8]
   (b) [2,3,2,4,3] y [9,6,8,7]
   (c) [2,2,3,3,4] y [6,8,7,9]
   (d) Ninguna de las opciones anteriores.



Respuesta B)

6. Sea una tabla hash con m=5 y h(x) = x mod 5 usando para la resolución de colisiones un hashing abierto con 3 niveles combinado con encadenamiento directo. En ella se van insertando en orden los siguientes valores: 41,58,16,66,77,12,25,21,31,14. Indique cuál de las siguientes afirmaciones es **cierta**:

   (a) Tras insertar el 25 la tabla hash no ha necesitado realizar encadenamiento.
   (b) Tras insertar el 14 la posición 0 de la tabla carece de valores.
   (c) Tras insertar el valor 77 la tabla hash tiene la posición 1 incompleta.
   (d) Tras insertar el valor 21 la tabla tiene todos los niveles completos.

m = 5; h(x) = x mod 5; Valores = 41, 58, 16, 66, 77, 12, 25, 21, 31, 14

0	1	2	3	4
25	41	77	58	14
	16	12		
	66			
	21			
	31			

41 mod 5 = 1
58 mod 5 = 3
16 mod 5 = 1 Colisión
66 mod 5 = 1 Colisión
77 mod 5 = 2
12 mod 5 = 2 Colisión
25 mod 5 = 0
21 mod 5 = 1 Colisión
31 mod 5 = 1 Colisión
14 mod 5 = 4

a) Cierto.
b) Falso. En la posición 0 ya estaba el valor 25.
c) Falso. La posición 1 ya tenía el valor 41.
d) Falso. No se pueden llenar todos los niveles.

Respuesta A)

Problema (4 puntos). Sea V[1..N] un vector con la votación de unas elecciones. La componente V[i] contiene el nombre del candidato que ha elegido el votante i. Se pide escribir un algoritmo que decida si algún candidato aparece en más de la mitad de las componentes (tiene mayoría absoluta) y que devuelva su nombre. Sirva como ayuda que para que un candidato tenga mayoría absoluta considerando todo el vector (al menos N/2+1 de los N votos), es condición necesaria pero no suficiente que tenga mayoría absoluta en alguna de las mitades del vector.

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

1. Elección razonada del esquema más apropiado de entre los siguientes: Voraz, Divide y Vencerás, Vuelta Atrás o Ramificación y Poda. Escriba la estructura general de dicho esquema e indique cómo se aplica al problema (0,5 puntos).

   El esquema más apropiado es Divide y Vencerás, cuyo esquema general se encuentra en el libro de texto base en la página 109, siendo éste:

```
fun DyV(problema)
    si trivial(problema) entonces
        dev solución-trivial
    sino hacer
        {p1, p2, ...pk} ← descomponer(problema)
        para i ∈ (1..k) hacer
            si ← DyV(pi)
        fpara
    fsi
    dev combinar(s1, s2, ...sk)
ffun
```

2. Descripción de las estructuras de datos necesarias (0,5 puntos solo si el punto 1 es correcto).

   Este problema es análogo al problema del "Elemento mayoritario en un vector", cuya solución se encuentra en el libro del texto base en la página 117 en adelante.

   El algoritmo nos debe proporcionar bien el valor mayoritario o bien indicarnos que éste no existe. Como caso trivial tendremos los vectores de un único elemento, cuyo elemento mayoritario es él mismo. Partimos del esquema principal y particularizamos:

```
fun Mayoritario(i,j:natural;,v:vector [1..n] de natural): entero
    var
        m:natural
        s1, s2:entero
    fvar
    si i = j entonces
        dev v[i]
    sino
        m ← (i + j) ÷ 2
        s1 ← Mayoritario(i,m,v)
        s2 ← Mayoritario(m+1,j,v)
    dev Combinar(s1, s2)
ffun
```

La función de combinación debe recibir los valores s_1 y s_2 y decidir qué implicaciones tienen con respecto del vector inicial. Los casos que se nos pueden presentar son los siguientes:

- s_1 y s_2 valen ambos -1. En tal caso, incluso si $s_1 = s_2$, no hay elemento mayoritario ya que al menos una de las dos mitades debe tener uno.
- Uno de los dos valores es -1 y otro no. En este caso un subvector tiene un elemento mayoritario. Para comprobar si es mayoritario en el resto del vector lo que hacemos es comprobar si ocurre n/ 2 + 1 más veces.
- Ambos valores son iguales y no negativos, lo que indica que dicho valor es el elemento mayoritario.
- Ambos valores son no negativos, luego ambos son candidatos a ser mayoritario y habrá que realizar dos comprobaciones.

3. Algoritmo completo a partir del refinamiento del esquema general (2.5 puntos sólo si el punto 1 es correcto). Si se trata del esquema voraz debe hacerse la demostración de optimalidad.

La función Combinar queda como sigue:

```
fun Combinar (a,b:entero;v:vector [1..n] de natural):entero
    si a = -1 ∧ b = -1 entonces dev -1 fsi
    si a = -1 ∧ b ≠ -1 entonces dev ComprobarMayoritario(b,v) fsi
    si a ≠ -1 ∧ b = -1 entonces dev ComprobarMayoritario(a,v) fsi
    si a ≠ -1 ∧ b ≠ -1 entonces
        si ComprobarMayoritario(a,v) = a entonces dev a
        sino si ComprobarMayoritario(b,v) = b entonces dev b fsi
        fsi
    fsi
fun
```

Por último, la función ComprobarMayoritario recorre el vector y cuenta las apariciones del argumento para ver si son más de n/2; en tal caso devuelve el valor pasado como argumento para su comprobación, y en caso contrario devuelve -1.

```
fun ComprobarMayoritario (x:natural;v:vector [1..n] de natural):entero
    var
        c:natural
    fvar
    c ← 1
    para k ← 1 hasta n hacer
        si v[k]=x entonces c ← c + 1 fsi
    fpara
    si c > n/2 entonces dev c sino dev -1 fsi
ffun
```

4. Estudio del coste del algoritmo desarrollado (0,5 puntos solo si el punto 1 es correcto).

La función de recurrencia asociada es la siguiente:

$$T(n) = 2T(n/2) + cn$$

En el peor caso es necesario recorrer el vector tras las llamadas recursivas para contar los elementos mayoritarios. En este caso tenemos a = b = 2 y k = 1 y, por tanto:

$$t(n) \in \Theta(n \log n)$$