FEBRERO 2015 – SEMANA 2 (MODELO A)

1. Dos socios que conforman una sociedad comercial deciden disolverla. Cada uno de los n activos que hay que repartir tiene un valor entero positivo. Los socios quieren repartir dichos activos a medias y, para ello, primero quieren comprobar si el conjunto de activos se puede dividir en dos subconjuntos disjuntos, de forma que cada uno de ellos tenga el mismo valor. ¿Cuál de los siguientes esquemas de los que puedan resolver el problema correctamente es más eficiente?

(a) Esquema voraz.

(b) Esquema de divide y vencerás.

(c) Esquema de vuelta atrás. → Su coste es $O(2^n)$

(d) Esquema de ramificación y poda.

Generalmente, todos los ejercicios relativos a de dividir empresas o sociedades se tratan de esquemas de Vuelta Atrás.

Respuesta C)

2. Dadas las matrices: A_1 (2x2), A_2 (2x5) y A_3 (5x3) y A_4 (3x1) siendo E(i,j) el número de operaciones mínimo para resolver la operación A_i x A_{i+1} x ... x Ai mediante programación dinámica, se pide indicar cuál de las siguientes opciones es cierta.

(a) E(2,4) = 25

(b) E(3,4) = 10

(c) E(1,2) = 40

(d) E(2,2) = 10

A (q x p); B (p x r) → A x B = q x p x r

a) E(2, 4)
   - $(A_2 \times A_3) \times A_4 = (2 \times 5 \times 3) + (2 \times 3 \times 1) = 30 + 6 = 36$
   - $A_2 \times (A_3 \times A_4) = (5 \times 3 \times 1) + (2 \times 5 \times 1) = 15 + 10 = 25$

b) E(3, 4) → $A_3 \times A_4 = 5 \times 3 \times 1 = 15$

c) E(1, 2) → $A_1 \times A_2 = 2 \times 2 \times 5 = 20$

d) E(2, 2) → No es posible calcularlo, ya que i=j (2 = 2), siendo E(i,j) = 0

Respuesta A)

3. Aplicando el algoritmo de Dijkstra al grafo con conjunto de vértices {1, 2, ... 8} y aristas con pesos (entre paréntesis) representadas por lista de adyacencia:

1 → 2(4), 3(10), 4(2), 6(8)	***2 → 4(4)***
3 → 6(1), 8(10)	***4 → 5(2), 6(1), 7(5)***
5 → 3(5), 6(1)	***6 → 3(9)***

7 → 1(7) ***8 → 3(2), 4(3), 7(2)***

Cuál de las siguientes NO es un valor correcto para el vector "Especial" D[1] generado por el algoritmo de Dijkstra a lo largo de las iteraciones, representando en cada iteración la distancia mínima hasta el momento entre los vértices 1 e i:

(a) D = [4, 10, 2, ∞, 8, ∞, ∞]
(b) D = [4, 10, 2, 4, 3, 7, ∞]
(c) D = [4, 9, 2, 4, 3, 7, ∞]
(d) D = [4, 10, 2, 4, 3, 7, 19]

- Caminos Mínimos
  En un grafo dirigido ponderador, se debe determinar la longitud del camino mínimo desde un nodo origen hasta cada uno de los demás nodos del grafo.
- Algoritmo Dijkstra
  - Coste: n^2 (puede llegar a ser n^3)
  - Candidatos: Conjunto de nodos del grafo
  - Funciones:
    - Solución: Se llega cuando se han visitado todos los nodos
    - Factible: Siempre es cierta
    - Selección: seleccionar el nodo con distancia mínima a C

En base a los datos del enunciado, si dibujamos la matriz de adyacencia de los nodos del grafo sería la siguiente:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	-	**4**	**10**	**2**	-	**8**	-	-
2	-	-	-	**4**	-	-	-	-
3	-	-	-	-	-	**1**	-	**10**
4	-	-	-	-	**2**	**1**	**5**	-
5	-	-	**5**	-	-	**1**	-	-
6	-	-	**9**	-	-	-	-	-
7	**7**	-	-	-	-	-	-	-
8	-	-	**2**	**3**	-	-	**2**	-

Nodos que han salido	**Nodos que no han salido**	**Vector Distancia especial [ ]**							**Predecesores**						
		2	**3**	**4**	**5**	**6**	**7**	**8**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	2, 3, 4, 5, 6, 7, 8	4	10	**2**	∞	8	∞	∞	1	1	1	1	1	1	1
1, 4	2, 3, 5, 6, 7, 8	4	10	2	4	**3**	7	∞	1	1	1	4	4	4	1
1, 4, 6	2, 3, 5, 7, 8	**4**	10	2	4	3	7	∞	1	1	1	4	4	4	1
1, 4, 6, 2	3, 5, 7, 8	4	10	2	**4**	3	7	∞	1	1	1	4	4	4	1
1, 4, 6, 2, 5	3, 7, 8	4	9	2	4	3	**7**	∞	1	5	1	4	4	4	1
1, 4, 6, 2, 5, 7	3, 8	4	**9**	2	4	3	7	9	1	5	1	4	4	4	7
1, 4, 6, 2, 5, 7, 3	8	4	9	2	4	3	7	9	1	5	1	4	4	4	7

Respuesta D)

4. Dado el vector M= [5, 6, 12, 1, 7, 9, 3, 7, 1, 11, 3, 2, 8, 6, 5, 9] al que aplicamos el algoritmo de ordenación Quicksort tomando como pivotes los valores M[0] y para una ordenación de menor a mayor. Se pide contestar la opción verdadera:

(a) La primera recursión del algoritmo genera dos llamadas a quicksort con argumentos [3 5 2 1 3 1] y [7 9 11 7 12 8 6 6 9] con pivote M[0] = 5.

(b) La primera recursión del algoritmo genera dos llamadas a quicksort con argumentos [6 12 1 7 9 3 7] y [1 11 3 2 8 6 5 9] con pivote M[0] = 5.

(c) La primera recursión del algoritmo genera dos llamadas a quicksort con argumentos [1 1 2 3 3 5] y [6 6 7 7 8 9 9 11 12] con pivote M[0] = 5.

(d) La primera recursión del algoritmo genera dos llamadas a quicksort con argumentos [1 3 6 7 7 9 12] y [1 2 3 5 6 8 9 11] con pivote M[0] = 5.

M = [5, 6, 12, 1, 7, 9, 3, 7, 1, 11, 3, 2, 8, 6, 5, 9] → Se toma al primer elemento, 5, como pivote.

El método de ordenación por partición (Quicksort) se define como un procedimiento recursivo.

Trabaja mejor con elementos desordenados en la entrada, que con elementos semiordenados.

Quicksort se basa en el algoritmo Divide y Vencerás, siendo más rápido y fácil de ordenar dos vectores o listas de datos pequeños, que un vector o una lista grande.

Al inicio de la ordenación se toma un elemento aproximadamente en la mitad del vector, por lo que, cuando se empieza a ordenar, hay que llegar hasta dicho vector ordenado respecto al punto de división o a la mitad del vector.

Se garantiza que los elementos a la izquierda de la mitad son los menores, y los situados a la derecha son los mayores.

Se avanza el recorrido hasta que se cumpla que i > p y j <= p, y se detiene cuando i >= j.

<u>Paso 1</u>

P														
5	6	12	1	7	9	3	7	1	11	3	2	8	6	5	9
	i														j

Como i > p (6 > 5), pero j no es menor o igual que p (9 > 5), el índice “i” se queda en el elemento con índice 2 en el vector, y j se mueve una posición hacia la izquierda.

<u>Paso 2</u>

P														
5	6	12	1	7	9	3	7	1	11	3	2	8	6	5	9
	i													j	

Ahora sí, se cumple tanto que i > p, como que j <= p, por lo tanto, se intercambian los elementos en los índices “i” y “j” entre sí.

El índice i se desplaza una posición a la derecha, y el índice j se desplaza una posición a la izquierda.

Paso 3

	P														
5	5	12	1	7	9	3	7	1	11	3	2	8	6	6	9
		i											j		

Se cumple que i > p (12 > 5), pero no que j <= p (6 > 5). Se desplaza una posición a la izquierda el índice “j”, hasta que se cumpla que j sea <= p.

	P														
5	5	12	1	7	9	3	7	1	11	3	2	8	6	6	9
		i										j			

Paso 4

	P														
5	5	12	1	7	9	3	7	1	11	3	2	8	6	6	9
		i									j				

Ahora sí, i > p, así como j <= p (2 < 5), por tanto, se intercambian los elementos entre sí en los índices “i” y “j”.

Paso 5

	P														
5	5	2	1	7	9	3	7	1	11	3	12	8	6	6	9
			i							j					

En esta ocasión, no se cumple que i > p (1 < 5), pero sí que j <= p (3 < 5), por tanto, se desplaza tan solo el índice “i” una posición a la derecha hasta que se cumpla que i > p

Paso 6

	P														
5	5	2	1	7	9	3	7	1	11	3	12	8	6	6	9
				i						j					

Como ahora si se cumplen ambas condiciones (i > p, j <= p), se intercambian los elementos entre sí en los índices “i” y “j”.

Paso 7

	P														
5	5	2	1	3	9	3	7	1	11	7	12	8	6	6	9
					i				j						

Aquí se cumple la condición i > p (9 > 5), pero no j <= p (11 > 5), por tanto, se desplaza en una posición a la izquierda el índice “j”.

Paso 8

	P														
5	5	2	1	3	9	3	7	1	11	7	12	8	6	6	9
					i			j							

Ahora si se cumplen ambas condiciones (i > p, j <= p), por lo que se intercambian los elementos entre sí en los índices “i” y “j”, y se desplazan a la derecha e izquierda, respectivamente.

Paso 9

P															
5	5	2	1	3	1	3	7	9	11	7	12	8	6	6	9
						i	j								

No se cumple ninguna de las dos condiciones (i > p → 3 < 5; j <= p → 7 > 5), por lo que se desplazan los índices "i" y "j" a la derecha e izquierda respectivamente.

Paso 10

P															
5	5	2	1	3	1	3	7	9	11	7	12	8	6	6	9
						j	i								

Llegados a este punto, i > j, no porque 7 sea mayor que 3 (que, en este caso, también ocurre), sino porque "i" ha sobrepasado el umbral de j, por tanto, el recorrido se detiene.

Se intercambia por tanto el índice "j" por el pivote P.

						P									
3	5	2	1	3	1	5	7	9	11	7	12	8	6	6	9

El vector resultante queda como: {3, 5, 2, 1, 3, 1} 5 {7, 9, 11, 7, 12, 8, 6, 6, 9}

Respuesta A)

5. El problema de la liga de equipos consiste en que dados n equipos con $n = 2^k$, se desea realizar una liga de forma que cada equipo puede jugar un partido al día y la liga debe celebrarse en n-1 días, suponiendo que existen suficientes campos de juego. El objetivo sería realizar un calendario que indique el día que deben jugar cada par de equipos. Si este problema se resuelve con un esquema divide y vencerás, considerando que el caso trivial se da cuando la liga consta de 2 equipos, la descomposición se realiza dividiendo el problema en dos partes similares, y la combinación se produce dando los subproblemas por resueltos y aplicando el principio de inducción.

Indica de entre los siguientes, cuál sería el coste mínimo de dicho algoritmo.

(a) $\Theta(n)$.

(b) $\Theta(n^2)$.

(c) $\Theta(n \log n)$.

(d) $\Theta(n^2 \log n)$.

Respuesta B)

6. Sea un grafo denso no dirigido representado con la siguiente matriz de adyacencia, en la que puede haber pesos de valor 0:

	1	2	3	4	5	6	7	8
1	-	2	3	4	5	0	1	2
2		-	0	2	4	0	2	4
3			-	0	3	0	3	0
4				-	2	0	4	2
5					-	0	5	4
6						-	0	0
7							-	2
8								-

Si se utiliza el algoritmo de Prim para calcular un árbol de recubrimiento mínimo tomando como raíz del árbol el nodo 1, indica cuál de las siguientes secuencias de aristas representa el orden en el que las selecciona el algoritmo de Prim como integrantes del árbol de recubrimiento mínimo:

(a) {1,6}, {1,7}, {6,2}, {6,3}, {4,6}, {6,5}, {6,8}.

(b) {1,6}, {6,2}, {2,3}, {3,8}, {4,6}, {6,5}, {6,7}.

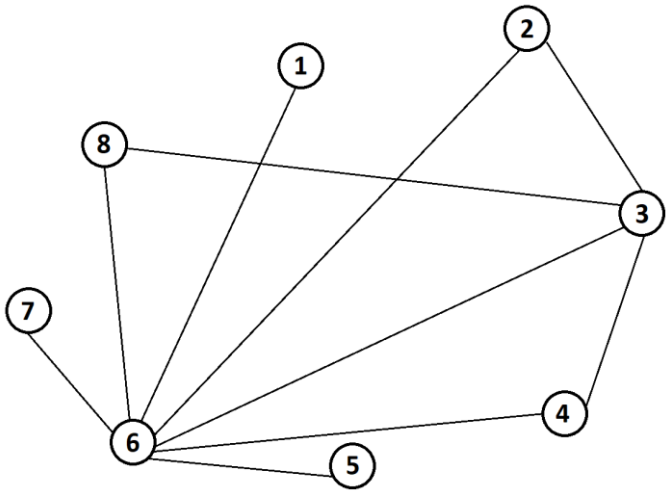
(c) {1,7}, {1,6}, {6,3}, {6,2}, {4,6}, {6,5}, {6,8}.

(d) {1,6}, {1,2}, {2,3}, {4,6}, {6,5}, {6,8}, {6,7}.

El algoritmo de PRIM (Página 66 del Libro de Texto Base de la asignatura) es utilizado siempre, únicamente, con grafos no dirigidos. Con él se pretende encontrar un árbol de recubrimiento mínimo (ARM) en un grafo conexo.

Un grafo denso es aquel en el que el número de aristas está cercano al número máximo de aristas. En contraparte, un grafo disperso es aquel que tiene solo algunas aristas.

Si nos fijamos en la matriz, y tomando en cuenta que el enunciado asume que puede haber pesos 0, dibujando el grafo, solamente para los pesos 0, este queda como:



Como los pesos 0 “no cuestan”, o “son gratuitos”, es mejor tomar estos (siempre y cuando el enunciado lo indica), en vez de querer ir determinando cuales serán las mejores aristas candidatas.

Aristas Elegidas	Nodos	Aristas (S)
1, 6	1	{1, 6}
6, 2	1, 6	{1, 6}, {6, 2}
2, 3	1, 6, 2	{1, 6}, {6, 2}, {2, 3}
3, 8	1, 6, 2, 3	{1, 6}, {6, 2}, {2, 3}, {3, 8}
4, 6	1, 6, 2, 3, 4	{1, 6}, {6, 2}, {2, 3}, {3, 8}, {4, 6}
6, 5	1, 6, 2, 3, 4, 5	{1, 6}, {6, 2}, {2, 3}, {3, 8}, {4, 6}, {6, 5}
6, 7	1, 6, 2, 3, 4, 5, 7	{1, 6}, {6, 2}, {2, 3}, {3, 8}, {4, 6}, {6, 5}, {6, 7}

Respuesta B

PROBLEMA (4 Puntos)

Se tiene una colección de n objetos "moldeables", cada uno con un volumen V_i, para i entre 1 y n, que hay que empaquetar utilizando envases de capacidad E. Se busca un algoritmo que calcule el empaquetamiento óptimo, es decir, que minimice la cantidad de envases utilizados, teniendo en cuenta que los objetos no se pueden fraccionar. Se pide:

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

El esquema más apropiado es el de Ramificación y Poda, cuyo esquema general es:

```
fun RamificacionYPoda (nodoRaiz, mejorSolución: TNodo, cota: real)
    monticulo = CrearMonticuloVacio()
    cota = EstimacionPes(nodoRaiz)
    Insertar(nodoRaiz, monticulo)
    mientras ¬ MonticuloVacio?(monticulo) ∧
              EstimacionOpt(Primero(monticulo)) ≤ cota hacer
        nodo ← ObtenerCima(monticulo)
        para cada hijo extensión válida de nodo hacer
            si solución(hijo) entonces
                si coste(hijo) ≤ cota entonces
                    cota ← coste(hijo)
                    mejorSolucion ← hijo
                fsi
            sino
                si EstimacionOpt(hijo) ≤ cota entonces
                    Insertar(hijo, monticulo)
                    si EstimacionPes(hijo) < cota entonces
                        cota ← EstimacionPes (hijo)
                    fsi
                fsi
            fsi
        fpara
    fmientras
ffun
```

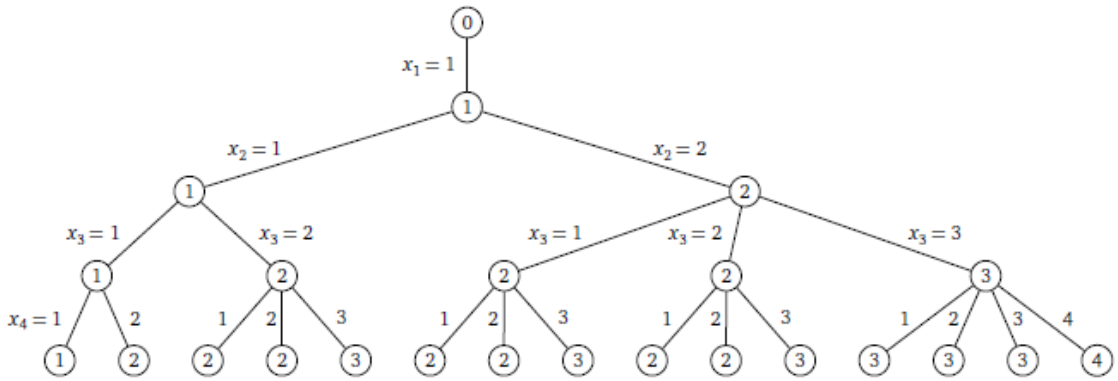
2. Descripción de las estructuras de datos necesarias (0,5 puntos solo si el punto 1 es correcto).

Suponemos que cada objeto cabe en un envase vacío.

Por tanto, como mucho, necesitamos n envases, que numeramos de 1 a n. Podemos representar las soluciones por tuplas de la forma (x_1, ..., x_n), siendo xi el envase donde hemos colocado el objeto i.

Como todos los envases vacíos son iguales, para cada objeto se puede usar uno de los envases ya ocupados, si cabe en alguno, o coger uno cualquiera de los que aún están vacíos.

Usamos un vector *capacidad[1..n]* para registrar la capacidad libre que queda en cada envase.



Árbol de Exploración

3. Algoritmo completo a partir del refinamiento del esquema general (2,5 puntos solo si el punto 1 es correcto).

Utilizaremos *envases* como estimación y utilizaremos el valor *óptimo* para detener la búsqueda cuando encontremos una solución con dicho valor. La estimación para todos los hijos excepto el último (utilizar un envase nuevo) coincide con la del padre, por lo que solo comprobaremos si este último hijo es prometedor.

El algoritmo queda como sigue:

```
{ (∀i : 1 ≤ i ≤ n : V[i] ≤ E) ∧ óptimo = ⌈(∑_{i=1}^{n} v_i)/E⌉ }
proc empaquetar-va(e E : real+, e V[1..n] de real+, sol[1..n] de 1..n, e k : 1..n, envases : 1..n,
                   capacidad[1..n] de real, e óptimo : 1..n,
                   sol-mejor[1..n] de 1..n, envases-mejor : 1..n + 1, encontrada : bool)
    { probamos con cada envase ya utilizado }
    i := 1
    mientras i ≤ envases ∧ ¬encontrada hacer
        si capacidad[i] ≥ V[k] entonces
            sol[k] := i
            capacidad[i] := capacidad[i] − V[k]     { marcar }
            si k = n entonces
                sol-mejor := sol ; envases-mejor := envases
                encontrada := (envases-mejor = óptimo)     { terminar }
            si no
                empaquetar-va(E, V, sol, k + 1, envases, capacidad, óptimo,
                              sol-mejor, envases-mejor, encontrada)
            fsi
            capacidad[i] := capacidad[i] + V[k]     { desmarcar }
        fsi
        i := i + 1
    fmientras
    si ¬encontrada entonces
        { probamos con un envase nuevo }
        sol[k] := envases + 1
        envases := envases + 1 ; capacidad[envases] := E − V[k]     { marcar }
        si envases < envases-mejor entonces
            si k = n entonces
                sol-mejor := sol ; envases-mejor := envases
                encontrada := (envases-mejor = óptimo)     { terminar }

            si no
                empaquetar-va(E, V, sol, k + 1, envases, capacidad, óptimo,
                              sol-mejor, envases-mejor, encontrada)
            fsi
        fsi
        capacidad[envases] := E ; envases := envases − 1     { desmarcar }
    fsi
fproc
```

La función principal que realiza la inicialización adecuada es:

```
{ ∀i : 1 ≤ i ≤ n : V[i] ≤ E }
fun empaquetar(E : real+, V[1..n] de real+) dev ⟨sol-mejor[1..n] de 1..n, envases-mejor : 1..n + 1⟩
var sol[1..n] de 1..n, capacidad[1..n] de real
    sol[1] := 1
    envases := 1
    capacidad[1] := E − V[1] ; capacidad[2..n] := [E]
    total := 0
    para i = 1 hasta n hacer total := total + V[i] fpara
    óptimo := ⌈total/E⌉
    encontrada := falso
    envases-mejor := n + 1     { peor que cualquier solución }
    empaquetar-va(E, V, sol, 2, envases, capacidad, óptimo, sol-mejor, envases-mejor, encontrada)
ffun
```

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

- Cota Optimista: Una cota inferior es el número de envases ya utilizados en la solución parcial.

- Cota Pesimista: Una cota superior consiste en sumar a los envases ya utilizados, los envases que se necesitan para almacenar los objetos pendientes que no quepan en ninguno de los ya utilizados, probando directamente en el orden en que se tienen los datos de entrada de los objetos.

- Coste: El número de recipientes está limitado a n, es decir al número de objetos. Una estimación del coste es el tamaño del árbol, que en el peor caso crece como O(n!), ya que cada nodo del nivel k puede expandirse con los n-k objetos que quedan por asignar a recipientes.