FEBRERO 2017 – SEMANA 2 (MODELO A)

1. Un peluquero pretende dar servicio a *n* clientes y conoce el tiempo requerido por cada uno de ellos, siendo t_i, i= 1,2,...,n el tiempo requerido por el cliente *i*. El objetivo es minimizar el tiempo total que todos los clientes están en el sistema, y como el número de clientes es fijo, minimizar la espera total equivale a minimizar la espera media. ¿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.

   El esquema más apropiado es el Esquema o Algoritmo Voraz, con el que se resuelve problemas de minimización de tiempo en el sistema.

Respuesta A)

2. Sea el problema de la mochila en su versión de objetos no fraccionables solucionado mediante programación dinámica. Suponga que se dispone de 3 objetos con pesos {9, 6, 5} y que aportan unos beneficios de {38,40,24}, respectivamente. Suponga también que dispone de una mochila con una capacidad máxima de 15. Considere que, al aplicar el algoritmo, los objetos se consideran en el orden dado en el enunciado ({9,6, 5}). Indique qué afirmación es **cierta**:

   (a) El beneficio máximo final obtenido es 80.
   (b) La tabla de resultados parciales en la fila correspondiente al objeto de peso 6 es: 0 0 0 0 0 0 24 24 24 40 40 40 64 64 64 78
   (c) La tabla de resultados parciales en la fila correspondiente al objeto de peso 5 es: 0 0 0 0 0 24 40 40 40 40 40 64 64 64 64 78
   (d) El problema propuesto se resuelve más eficientemente utilizando un enfoque voraz.

	CAPACIDADES																
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**BENEFICIOS**
PESOS	**9**	0	0	0	0	0	0	0	0	38	38	38	38	38	38	38	**38**
	6	0	0	0	0	0	0	40	40	40	40	40	40	40	40	78	**40**
	5	0	0	0	0	0	24	40	40	40	40	40	64	64	64	78	**24**

Para realizar el cálculo de cada celda, se realiza como:

Valor máximo entre

- Valor de la misma columna en la fila anterior
- Beneficio del Peso + X

X = Valor en la columna Y de la fila anterior
Y = Capacidad de la mochila (C) – Peso

Respuesta C)

3. Dadas las siguientes matrices (cuyas dimensiones se especifican entre paréntesis): A_1 (3x5), A_2 (5x2), A_3 (2x3) y A_4 (3x2) y siendo E(i,j) el número de operaciones mínimo para resolver la operación $A_i \times A_{i+1} \times \ldots \times A_j$ mediante programación dinámica, se pide indicar cuál de las siguientes opciones es **cierta**.

   (a) E(2,3) = 15
   (b) E(1,3) = 30
   (c) E(2,4) = 32
   (d) E(2,2) = 10

   El producto de un número "n" de matrices es optimizable en cuanto al número de multiplicaciones escalares requeridas. A la hora de multiplicar una serie de matrices se puede elegir en que orden queremos realizar las multiplicaciones entre estas. Se pueden realizar en un orden cualquiera, dada la propiedad asociativa de la multiplicación.

   En el caso que se nos plantea, las posibilidades de realizar el cálculo son las siguientes:

   1. ((A1 * A2) * A3) * A4
   2. (A1 * (A2 * A3)) * A4
   3. A1 * ((A2 * A3) * A4)
   4. A1 * (A2 * (A3 * A4))
   5. (A1 * A2) * (A3 * A4)

   Decidamos el orden que decidamos, el resultado siempre es el mismo. La diferencia está en el número de multiplicaciones que implica elegir un orden u otro. Al multiplicar dos matrices A_1 de tamaño pxq y A_2 de tamaño qxr, el número de multiplicaciones escalares es p * q * r.

   La cantidad total de multiplicaciones será, la suma de todas las multiplicaciones que hacen falta para multiplicar cada submatriz obtenida como resultado con la siguiente en el orden escogido.

   1. ((A1 * A2) * A3) * A4 → 3 * 5 * 2 + 3 * 2 * 3 + 3 * 3 * 2 = 30 + 18 + 18 = 66
   2. (A1 * (A2 * A3)) * A4 → 5 * 2 * 3 + 3 * 5 * 3 + 3 * 3 * 2 = 30 + 45 + 18 = 93
   3. A1 * ((A2 * A3) * A4) → 5 * 2 * 3 + 5 * 3 * 2 + 3 * 5 * 2 = 30 + 30 + 30 = 90
   4. A1 * (A2 * (A3 * A4)) → 2 * 3 * 2 + 5 * 2 * 2 + 3 * 5 * 2 = 12 + 20 + 30 = 62
   5. (A1 * A2) * (A3 * A4) → 3 * 5 * 2 + 2 * 3 * 2 + 3 * 2 * 2 = 30 + 12 + 12 = **54**

   Eligiendo un orden adecuado, como podemos ver, optimizamos el coste de realizar las multiplicaciones escalares necesarias para llegar al resultado. Para encontrar el modo de ordenar las multiplicaciones vamos a utilizar un algoritmo de Programación Dinámica.

   Respuesta A) E(2, 3) = 15 → Falsa
   A2 * A3 = 5 * 2 * 3 = **30**

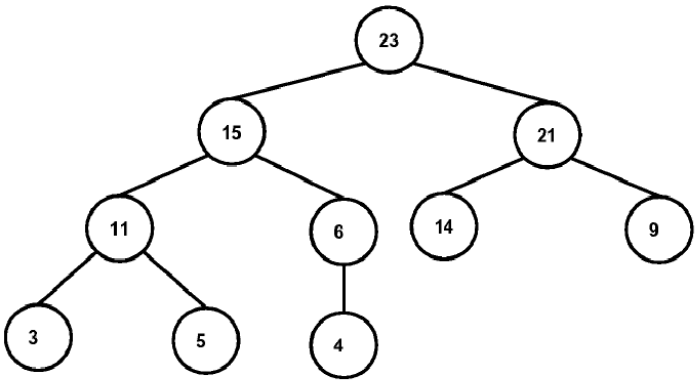
   Respuesta B) E(1, 3) = 30 → Falsa
   1. (A1 * A2) * A3 = 3 * 5 * 2 + 3 * 2 * 3 = 30 + 18 = **48**
   2. A1 * (A2 * A3) = 5 * 2 * 3 + 3 * 5 * 3 = 30 + 45 = 75

   Respuesta C) E(2, 4) = 32 → Cierta
   1. (A2 * A3) * A4 = 5 * 2 * 3 + 5 * 3 * 2 = 30 + 30 = 60
   2. A2 * (A3 * A4) = 2 * 3 * 2 + 5 * 2 * 2 = 12 + 20 = **32**

Respuesta D) E(2, 2) = 10 → Falsa
A2 * A2 → Si i = j → E(i, j) = 0

Respuesta C)

4. Dado el siguiente montículo de máximos, ¿Cuál de los siguientes vectores lo representa de forma **correcta**?



(a) 23, 15, 21, 11, 6, 14, 9, 3, 5, 4
(b) No es un montículo de máximos, sino de mínimos
(c) 23, 15, 11, 3, 5, 6, 4, 21, 14, 9
(d) 23, 15, 21, 11, 6, 3, 5, 4, 14, 9

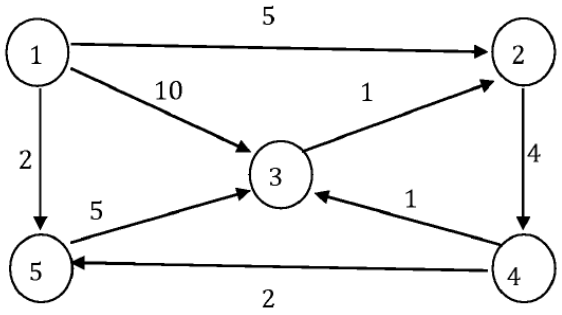
Para poder saber cuál es el vector que representa al grafo (montículo de máximos), se comienza desde la raíz, continuando por sus nodos hijos de izquierda a derecha, hasta llegar a la máxima profundidad del mismo.

Siendo así, el vector que representa a este grafo es:

[23, 15, 21, 11, 6, 14, 9, 3, 5, 4]

Respuesta A)

5. Aplicando el algoritmo de Dijkstra al grafo:



Indicar cuál de las siguientes **NO** es un valor correcto para el vector "Especial" D[i] 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 = [5,10,∞,2]
(b) D = [5,7,∞,2]
(c) D = [5,10,9,2]
(d) D = [5,7,9,2]

Nodos que han salido	Nodos que no han salido	Vector Distancia				Predecesores o Vector Especial			
		2	3	4	5	2	3	4	5
1	2, 3, 4, 5	5	10	∞	**2**	1	1	1	1
1, 5	2, 3, 4	**5**	10	∞	2	1	1	1	1
1, 5, 2	3, 4	5	**10**	9	2	1	1	2	1
1, 5, 2, 3	4	5	10	9	2	1	1	2	1

Respuesta C)

6. ¿Cuál de las siguientes afirmaciones es **falsa** con respecto al coste de las funciones de manipulación de grafos?

   (a) La función `AñadirVertice` que añade un vértice a un grafo y devuelve el grafo con dicho vértice incluido, tiene un coste constante cuando el grafo se implementa con una matriz de adyacencia.
   (b) La función `BorrarArista` es menos costosa cuando el grafo se implementa mediante una matriz de adyacencia.
   (c) La operación `Adyacente?`, que comprueba si dos nodos son adyacentes, es más costosa cuando el grafo se implementa con una lista de adyacencia.
   (d) La función `Adyacentes`, que devuelve una lista con los vértices adyacentes a uno dado, es más costosa cuando el grafo se implementa con una matriz de adyacencia.

Función Manipulación Grafo	Matriz Adyacencia	Lista 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)

Respuesta A)

Problema (4 puntos).

Un informático autónomo, que se ofrece a programar aplicaciones software para empresas y se compromete a entregarlas antes de una fecha límite fijada por la empresa, ha recibido *n* solicitudes de empresas. El informático conoce, para cada una de las aplicaciones, el beneficio b_i que obtendría por programarla. El tiempo que se tarda en programar cada aplicación es también variable y viene dato por t_i. También sabe los días d_i que quedan antes de la fecha tope fijada por la empresa que ha solicitado cada aplicación. Se pide un algoritmo que el informático pueda utilizar para decidir qué aplicaciones debe escoger para maximizar el beneficio total obtenido.

La resolución del 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)

   El esquema más apropiado puede ser tanto el de "Programación Dinámica" como el de "Ramificación y Poda". Se puede usar cualquier solución siempre que no nos den un conjunto de posibilidades en el enunciado.

Programación Dinámica

```
fun ObjetosMochila(vol: vector, M:Tabla, n:entero, V:entero, objetos: Vector)
    var
        i,W: entero
    fvar
    W ← V
    para i ← n hasta 1 incremento -1 hacer
        si M[i,W] = M[i-1,W] entonces
            objetos[i] ← 0
        sino
            objetos[i] ← 1
            W ← W - vol[i]
        fsi
    fpara
ffun
```

Ramificación y Poda

```
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
```

Se trata de un problema de optimización en el que hay dos variables, el beneficio de cada curso y el tiempo requerido para impartirlo.

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

   El problema se resuelve del mismo modo que el problema de los cursos de formación, que se encuentra en el Libro de Texto Base de la asignatura en la página 205.

   Podemos representar la solución mediante un vector cursos, donde si la posición i tiene el valor cierto es porque el curso correspondiente a la solicitud i se impartirá. También utilizamos un montículo en el que cada nodo además de almacenar la solución parcial, etapa y cota, almacena el tiempo y beneficio acumulado.

```
tipo TVectorB = matriz[0..n] de booleano
tipo TVectorR = matriz[0..n] de real
tipo TVector = matriz[0..n] de entero
tipo TNodo = registro
    cursos: TVectorB
    k: entero
    beneficioT: real
    tiempoT: real
    estOpt: real
fregistro
```

   Para que un conjunto de cursos se pueda impartir (sea admisible), la impartición de todos ellos tiene que poder organizarse de forma que cada curso se termine sin superar el correspondiente límite de días (d_i). Concretamente, si tomamos los cursos $i_1, \cdots, i_m$ en este orden, el tiempo que se tarda en terminar de impartir el curso i_k es $\Sigma^k_{j=1} t_j$, de modo que la secuencia es admisible si $\Sigma^k_{j=1} t_j <= d_k$, para todo k entre 1 y m.

   Para comprobar si un conjunto es admisible ordenamos los cursos por fecha tope creciente y hacemos la comprobación. Para diseñar el algoritmo de ramificación y poda necesitamos definir las estimaciones pesimista y optimista que utilizará el algoritmo para realizar la poda.

   Podemos obtener una estimación optimista sumando el beneficio de todos los cursos que se pueden impartir sin llegar a su fecha límite después de los ya elegidos, como si cada uno se fuera a empezar a impartir justo después de terminar el último impartido.

   La estimación pesimista o cota de poda la podemos calcular sumando el beneficio de los cursos ya seleccionados para ser impartidos, y considerando que el resto de los cursos se van impartiendo en orden (por fecha tope creciente) mientras no se supere su fecha límite de impartición.

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 realizarse la demostración de optimalidad. Si se trata del esquema de programación dinámica, deben proporcionarse las ecuaciones de recurrencia.

```
fun Cursos (beneficios: TVectorR, tiempos, limites: TVector,
            cursos: TVertorB, beneficioT: real)
  var
    monticulo: TMonticulo
    nodo,hijo: TNodo
    cota,estPes: real
  fvar
```

```
monticulo ← CrearMonticuloVacio()
valor ← 0
{Construimos el primer nodo}
nodo.cursos ← cursos
nodo.k ← 0
nodo.tiempoT ← 0
nodo.beneficioT ← 0
nodo.estOpt ← EstimacionOpt(beneficios, tiempos, limites,
                nodo.k,nodo.tiempoT,nodo.beneficioT)
Insertar(nodo, monticulo)
cota ← EstimacionPes(beneficios, tiempos, limites,
                nodo.k,nodo.tiempoT,nodo.beneficioT)
mientras ¬ MonticuloVacio?(monticulo) ∧
          EstimacionOpt(Primero(monticulo)) ≥ cota hacer
   nodo ← ObtenerCima(monticulo)
   {se generan las extensiones válidas de nodo}
   hijo.k ← nodo.k + 1
   hijo.cursos ← nodo.cursos
   {se imparte el curso}
   si nodo.tiempoT + tiempos[hijo.k] ≤ limites[hijo.k] entonces
      hijo.cursos[hijo.k] ← cierto
      hijo.beneficioT ← nodo.beneficioT + beneficios[hijo.k]
      hijo.tiempoT ← nodo.tiempoT + tiempos[hijo.k]
      hijo.estOpt ← nodo.estOpt
      si hijo.k = n entonces
         si beneficioT ≤ hijo.beneficioT entonces
            cursos ← hijo.cursos
            beneficio ← hijo.beneficioT
            cota ← beneficio
         fsi
      sino {la solución no está completa}
         Insertar(hijo, monticulo)
      fsi
   fsi
   {no se imparte el curso}
   hijo.estOpt ← EstimacionOpt(beneficios,tiempos,limites,
         hijo.k,nodo.tiempoT,nodo.beneficioT)
   si hijo.estOpt ≥ cota entonces
      hijo.cursos[hijo.k] ← falso
      hijo.beneficioT ← nodo.beneficioT
      hijo.tiempoT ← nodo.tiempoT
      si hijo.k = n entonces
         si beneficio ≤ hijo.beneficioT entonces
            cursos ← hijo.cursos
            beneficio ← hijo.beneficioT
            cota ← beneficio
         fsi
      sino {la solución no está completa}
         Insertar(hijo, monticulo)
         estPes ← EstimacionPes(beneficios, tiempos, limites,
                  hijo.k,hijo.tiempoT,hijo.beneficioT)
         si cota < estPes entonces
            cota ← estPes
         fsi
      fsi
   fsi
fmientras
ffun
```

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

   Coste Computacional = (Factor Ramificación)^{nn} = 2^n → $O(2^n)$

   n = Nº de niveles (Profundidad del Árbol)