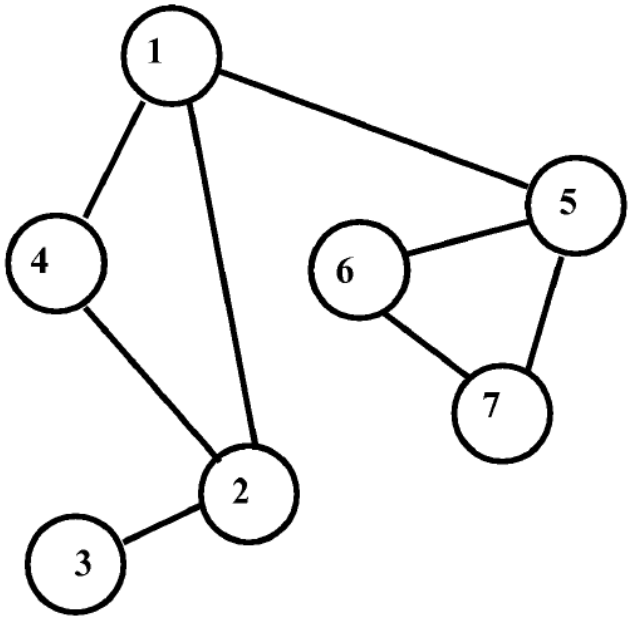
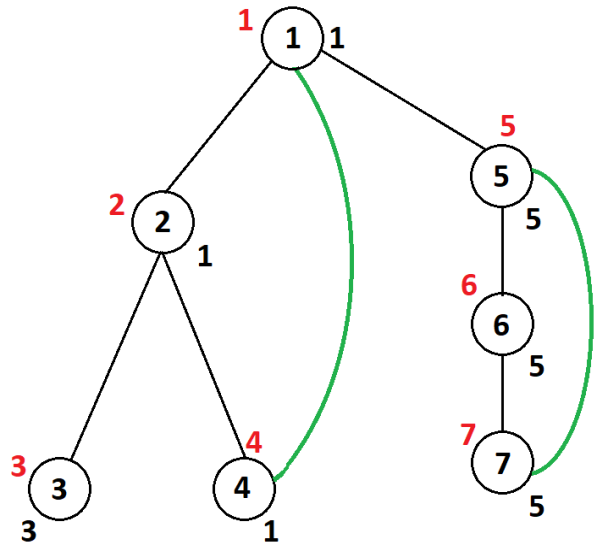
FEBRERO 2012 – SEMANA 1 (MODELO A)

1. Dado el grafo no dirigido de la figura, la aplicación del algoritmo de cálculo de los puntos de articulación (y de su árbol de recubrimiento asociado) da como resultado el vector *bajo[]* siguiente:



a.- [1,1,3,1,5,5,5]
b.- [1,2,3,4,1,6,7]
c.- [1,3,3,1,1,3,3]
d.- Ninguna de las anteriores

En primer lugar, se realiza un recorrido en profundidad, por orden de los índices de cada nodo, quedando el grafo como:



Valor de Orden
Aristas de Retroceso
Vector bajo[]

Se dibuja el valor de **orden**, que, en este caso, coincide con el número del nodo.

En siguiente lugar, se dibujan las **aristas de retroceso**, las cuales son aristas que conectan nodos descendientes con nodos antecesores en el árbol.

En nuestro caso existía una de estas aristas de retroceso, desde el nodo 1 al nodo 4, y otra de ellas desde el nodo 5 al nodo 7.

Realizamos entonces el recorrido del grafo empezando por el nodo más a la izquierda de todos, es decir, el nodo 3. Se toma en cuenta el número de orden de dicho nodo, el número de orden de cualquiera de sus aristas de retroceso, y el valor del vector **bajo[v]**.

El valor de bajo[] para el nodo 3 es el valor comprendido entre su número de orden (3), el número de orden de cualquier arista de retroceso que tenga (que en este caso no tiene), y el valor de bajo[] para alguno de sus hijos, por lo que el valor de bajo[] para el nodo 3 es: 3.

El siguiente es el nodo 4, cuyo valor de bajo[] es el valor entre su número de orden (4), el número de orden de cualquiera de sus aristas de retroceso (1), y el valor de bajo[] de sus hijos (que no tiene), por lo que el valor de bajo[] para el nodo 4 es: 1.

bajo[2]	Nº de orden: 2	El menor valor es: 1
	bajo[] aristas de retroceso: 0	
	bajo[] nodos hijos: 3 y 1	

bajo[5]	Nº de orden: 5	El menor valor es: 5
	bajo[] aristas de retroceso: 0	
	bajo[] nodos hijos: 5 y 5	

bajo[6]	Nº de orden: 6	El menor valor es: 5
	bajo[] aristas de retroceso: 0	
	bajo[] nodos hijos: 5	

bajo[7]	Nº de orden: 7	El menor valor es: 5
	bajo[] aristas de retroceso: 5	
	bajo[] nodos hijos: 0	

bajo[1]	Nº de orden: 1	El menor valor es: 1
	bajo[] aristas de retroceso: 0	
	bajo[] nodos hijos: 1, 3, 1	

Este grafo posee 3 Puntos de Articulación (PDA).

Los PDA o Vértices de Corte (VDC) son un vértice de un grafo, tal que, si se elimina del grafo, se produce un incremento en el número de componentes conexos.

Los PDA que hay en este grafo son: 1, 2, 5. Calculamos el valor del vector bajo, comenzando en el primer PDA, y por número de orden:

Nodo	bajo[v]
1	1
2	1
3	3
4	1
5	5
6	5
7	5

Respuesta A)

2.- Sea el problema de la mochila en su versión de objetos no fraccionables solucionado con programación dinámica. Supongamos que se dispone de 5 objetos con pesos: 1,3,4,5,7 y beneficios: 2,5,10,14,15 respectivamente, y un volumen máximo de 8.Identifica cuál de las siguientes respuestas correspondería al contenido de la tabla de resultados parciales en la fila correspondiente al objeto de peso 5, si dichos objetos se consideran en orden creciente de pesos.

a. 0 2 2 5 10 12 12 15 15
b. 0 2 2 5 10 14 16 16 19
c. 0 2 2 5 10 12 14 16 19
d. Ninguna de las anteriores.

Lo que busca o pretende el problema de la mochila es maximizar el beneficio. En este caso, como sabemos que los objetos no son fraccionables, podemos realizar el cálculo mediante dos algoritmos distintos, pero como además nos indican cual de ellos (con Programación Dinámica), hay que hacer uso de este. Los métodos de resolución para el problema de la Mochila son:

PROBLEMA DE LA MOCHILA

- Objetos Fraccionables → Algoritmos Voraces (Algoritmo de Optimización)
- Objetos no Fraccionables } - Programación Dinámica (Mejor Coste Espacial O(nC)) - Ramificación y Poda (Mayor Coste Exponencial)

Dibujamos una tabla que tendrá dos dimensiones. Una de ellas es el Volumen/Peso de os objetos, que irán en las filas de dicha tabla. La otra dimensión es las Capacidades/Volumen de la propia mochila, correspondientes a las columnas en dicha tabla, yendo dichas capacidades desde 0 hasta la Capacidad Máxima ($C_{MÁX}$) de la Mochila (8 en este caso).

La primera fila y columna corresponderán al caso base, por lo que el valor de cada celda en estos casos tendrá el valor 0. El valor en cada una de las celdas será el valor máximo entre:

Valor Máximo entre:

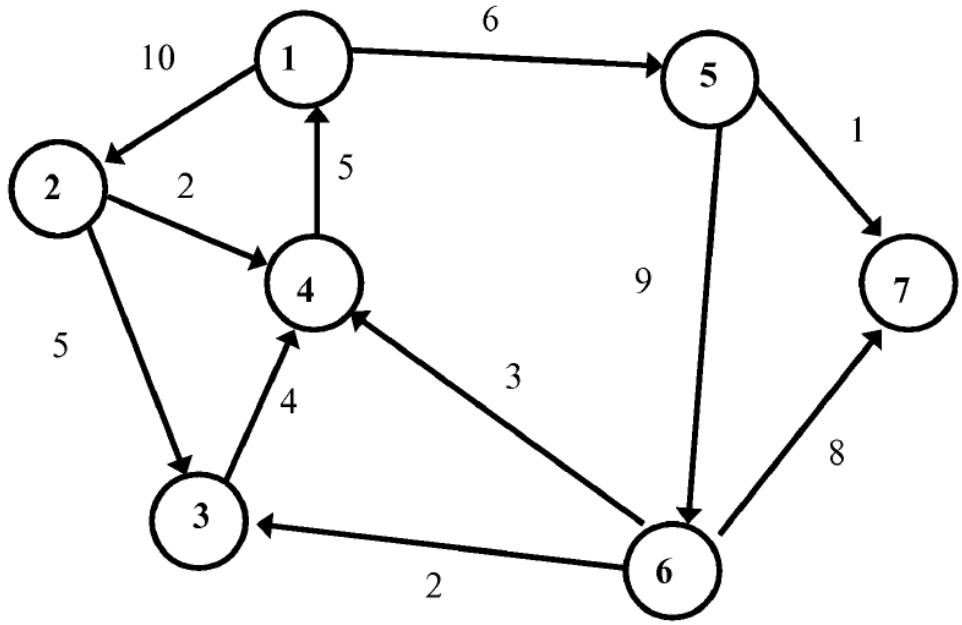
- Valor fila anterior en la misma columna
- Beneficio del Volumen + X

X = Fila anterior, Columna Y
Y = Capacidad - Volumen

		CAPACIDADES								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
	0	0	0	0	0	0	0	0	0	**BENEFICIOS**
PESOS	**1**	0	2	2	2	2	2	2	2	**2**
	3	0	2	2	5	7	7	7	7	**5**
	4	0	2	2	5	10	12	12	15	15	**10**
	5	0	2	2	5	10	14	16	16	19	**14**
	7	0	2	2	5	10	14	16	16	19	**15**

Respuesta B)

3.- Dado el siguiente grafo dirigido:



Se pide detallar el valor del vector de distancias *especial[]* en el paso del algoritmo de Dijkstra en el que se selecciona el nodo v=7, tomando como nodo origen el nodo 1.

a. [10,15,12,6,15,7]
b. [10, ∞, ∞,6,15,7]
c. [10,∞,∞,6,∞,∞]
d. Ninguna de las anteriores.

Nodos que han salido	Nodos que no han salido	Vector Distancia especial [ ]						Predecesores					
		2	3	4	5	6	7	2	3	4	5	6	7
1	2, 3, 4, 5, 6, 7	10	∞	∞	**6**	∞	∞	1	1	1	1	1	1
1, 5	2, 3, 4, 6, 7	10	∞	∞	6	15	**7**	1	1	1	1	5	5
1, 5, 7	2, 3, 4, 6	**10**	∞	∞	6	15	7	1	1	1	1	5	5
1, 5, 7, 2	3, 4, 6	10	15	**12**	6	15	7	1	2	2	1	5	5
1, 5, 7, 2, 4	3, 6	10	**15**	12	6	15	7	1	2	2	1	5	5
1, 5, 7, 2, 4, 3	6	10	15	12	6	15	7	1	2	2	1	5	5

Respuesta B)

4.- En un problema de mochila con objetos fraccionables tenemos n = 8 objetos disponibles. Los pesos de los objetos son w= (8, 7, 6, 5, 4, 3, 2, 1) y los beneficios son v= (10, 9, 8, 7, 6, 5, 4, 3). ¿Cuál es el beneficio óptimo para este ejemplo, suponiendo que la capacidad de la mochila es M = 9?

a. 12
b. 15
c. 16.5
d. Ninguna de las otras respuestas es correcta.

Como sabemos que los objetos son fraccionables, esto nos pone en la pista de que podemos resolver el cálculo con el Algoritmo Voraz. La fórmula a aplicar sería:

Beneficio / Peso → Vi / Wi

Peso (w)	Beneficio (v)	Vi / Wi
8	10	10/8 = 1'25
7	9	9/7 = 1'28
6	8	8/6 = 1'33
5	7	7/5 = 1'4
4	6	6/4 = 1'5
3	5	5/3 = 1'6
2	4	4/2 = 2
1	3	3/1 = 3

Se irán tomando los valores obtenidos de aplicar la fórmula v_i/w_i, de mayor a menor beneficio obtenido.

$\Sigma_v = 3 + 4 + 5 + (3/4 \cdot 6) = 12 + 18/4 = (48 + 18) / 4 = 66 / 4 = 16'5$

$\Sigma_w = 1 + 2 + 3 + (3/4 \cdot 4) = 1 + 2 + 3 + (12/4) = 1 + 2 + 3 + 3 = 9$ → M = 9

A partir de que la suma de los pesos es 6, como el siguiente peso es 4, 6+4 = 10, lo cual supera el valor e la capacidad máxima buscada, M = 9.

Siendo así, hay que introducir una fracción que, sumada al acumulado del sumatorio de los pesos, sea 9 o menor que 9.

Como estamos en el volumen 4, tomamos 3/4 partes del peso, por lo que 3/4 · 4 = 12 / 4 = **3**

Este mismo valor, 3/4, lo tomamos en el sumatorio del beneficio.

Respuesta C)

5.- Considérese el vector v[1..n] = [6,5,5,2,5,5,3,2,2] cuál de las siguientes opciones es cierta:

a. El vector v es un montículo de máximos.
b. El vector v no es un montículo de máximos porque el elemento v[4] = 2 debe ser flotado.
c. El vector v no es un montículo de máximos porque el elemento v[4] = 2 debe ser hundido.
d. Ninguna de las anteriores.

Los montículos son un tipo especial de árbol binario que se implementan sobre vectores con las siguientes propiedades:

- Es un árbol balanceado y completo (los nodos internos tienen siempre dos hijos) con la posible excepción de un único nodo cuando el número de elementos es par.
- Cada nodo contiene un valor mayor o igual que el de sus nodos hijos (montículo de máximos), o menor o igual (montículo de mínimos).

A la segunda de estas propiedades se la denomina “propiedad de montículo”, la cual permite tener en la cima (nodo raíz) del montículo el elemento mayor o menor, si se trata de un montículo de máximos o de mínimos, respectivamente, siendo esta la utilidad fundamental del montículo.

Los nodos de profundidad K están situados en las posiciones 2^k y siguientes del vector hasta la posición 2^{k+1}-1.

El nodo padre del elemento en la posición “i” estará en “i/2 (div., entera)”. Los nodos hijos al elemento en la posición “i” estarán en “2 * i, 2 * (i + 1)”.

La inserción y borrado de elementos es muy eficiente, teniendo un coste de O(log n). Las operaciones que podemos realizar con los montículos son las siguientes:

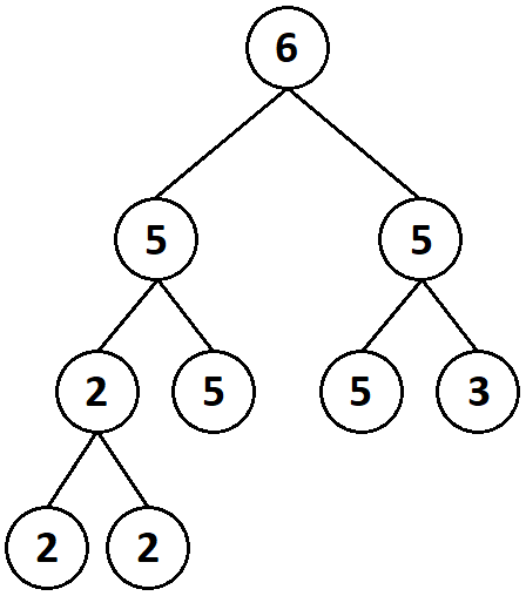
- Flotar: Intercambiar por el nodo inmediatamente a un nivel superior.
- Hundir: Intercambiar por el nodo hijo de menor valor (montículo de mínimos).
- Insertar: Añadir el nodo al final, y flotar a su sitio correspondiente.
- Primero: Lleva asociado un coste constante O(1).
- Obtener Cima y Borrarla: Situar el último nodo en la cima y hundir esta.

- *Montículo de Mínimos*
  Cada nodo contiene un valor menor o igual que el de sus nodos hijos, disponiendo con el de una estructura de datos en la que encontrar el valor mínimo es una operación de coste constante.

- *Montículo de Máximos*
  Cada nodo contiene un valor mayor o igual que el de sus nodos hijos.

Nota: El montículo sirve de apoyo a la creación de un algoritmo de ordenación eficiente, conocido este como ***Heapsort***.

Si representamos el vector dibujándolo como un árbol binario, se obtiene el siguiente:



La respuesta A) podría ser cierta. B) no es cierta, ya que si intercambiamos el elemento v[4] = 2 por v[2] = 5, esto hace que el mónticulo no sea de máximos.

La respuesta C) no es cierta, ya que al hundir v[4] = 2, no existe un nodo hijo de menor valor que 2. Con todo ello, la respuesta D) tampoco es cierta.

Respuesta A)

6.- Se dispone de un vector, V, que almacena números enteros en orden estrictamente creciente, y se desea averiguar si existe algún elemento que cumpla V[i]=i. ¿Cuál sería la estrategia más eficiente para resolver el problema?

a. Algoritmo voraz.
b. Divide y vencerás.
c. Programación dinámica.
d. Ninguna de las anteriores es aplicable al problema

a) Algoritmo Voraz

Se aplica a problemas de optimización en los que la solución se puede construir paso a paso sin necesidad de reconsiderar decisiones ya tomadas.

El problema que se puede resolver con este tipo de algoritmos, es el de encontrar un conjunto de candidatos que constituyan una solución y que optimice una función objetivo.

(Ejemplos: Planificación de tareas, problemas modelados con grafos, cálculos de recorrido, optimización de pesos, etc.).

b) Divide y Vencerás

Técnica que se basa en la descomposición de un problema en subproblemas de su mismo tipo, que permite disminuir la complejidad y, en algunos casos, paralelizar la resolución de los mismos.

La estrategia que persigue 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

Un ejemplo sencillo de aplicación de la técnica Divide y Vencerás es la búsqueda binaria en un vector ordenado, que conlleva un coste logarítmico.

c) Programación Dinámica

Registra resultados parciales que se producen durante la resolución de algunos problemas y que se utilizan repetidamente.

De esta forma, se consigue reducir el coste del cómputo, al evitar la repetición de ciertos cálculos.

(Ejemplos: Fibonacci, Coeficientes Binomiales, Devolución del Cambio, Viaje por el Río, la Mochila, Multiplicación asociativa de Matrices, Camino de Coste mínimo entre nodos de un grafo dirigido, Distancia de Edición, etc.).

Respuesta B)

Problema (4 puntos)

Una empresa de mensajería tiene n repartidores con distintas velocidades según el tipo de envío. Se trata de asignar los próximos n envíos, uno a cada repartidor, minimizando el tiempo total de todos los envíos. Para ello se conoce de antemano la tabla de tiempos T[1..n,1..n] en la que el valor t_{ij} corresponde al tiempo que emplea el repartidor i en realizar el envío j. Se pide:

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

- Elección del esquema más apropiado, el esquema general y explicación de su aplicación al problema (0,5 puntos).
- Descripción de las estructuras de datos necesarias (0.5 puntos).
- Algoritmo completo a partir del refinamiento del esquema general (2,5 puntos).
- Estudio del coste del algoritmo desarrollado (0.5 puntos).

Determinar qué esquema algorítmico es el más apropiado para resolver el problema

Se trata de un problema de optimización con restricciones. Por tanto, podría ser un esquema voraz o un esquema de ramificación y poda.

Sin embargo, descartamos el esquema voraz porque no es posible encontrar una función de selección y de factibilidad tales que una vez aceptado un candidato se garantice que se va alcanzar la solución óptima.

Se trata, por tanto, de un algoritmo de Ramificación y Poda.

Escribir el esquema general

```
funcion ramificacion_poda (ensayo) dev ensayo {ensayo es un nodo}
m ← monticulo_vacio();
cota_superior ← cota_superior_inicial;
solucion ← primera_solucion;
añadir_nodo(m, ensayo); {asignamos tareas al agente 1}
mientras ¬ vacio(m) hacer
    nodo ← extraer_raiz(m);
    si valido(nodo) entonces { están completas las asignaciones }
        si coste_asig(nodo) < cota_superior entonces
                solucion ← nodo;
                cota_superior ← coste_asig(nodo)
        fsi
    si no
        si cota_inferior(nodo) >= cota_superior entonces dev solucion
            si no
                para cada hijo en compleciones(nodo) hacer
                    si cota_inferior(hijo) < cota_superior
                        añadir_nodo(m, hijo)
                    fsi
                fpara
```

fsi
fsi
fmientras
ffuncion

Indicar que estructuras de datos son necesarias

nodo=tupla
asignaciones: vector[1..N];
último_asignado: cardinal;
filas_no_asignadas: lista de cardinal;
coste: cardinal;

Montículo de mínimos (cota mejor: la de menor coste).

Desarrollar el algoritmo completo

Las funciones generales del esquema general que hay que instanciar son:

0) solución(nodo): si se han realizado N asignaciones (último_asignado==N)

1) acotar(nodo,costes): nodo.coste + "mínimo coste de las columnas no asignadas"

2) compleciones(nodo,costes): posibilidades para la siguiente asignación (valores posibles para asignaciones[último_asignado+1])

Algoritmo Completo:

```
Función asignación(costes[1..N,1..N]) dev solución[1..N]
    Montículo:=montículoVacío();
    nodo.último_asignado=0;
    nodo.coste=0;
    cota:=acotar(nodo,costes);
    poner((cota,nodo),Montículo);
    mientras no vacío(Montículo) hacer
        (cota,nodo):=quitarPrimero(Montículo);
        si nodo.último_asignado==N entonces
            devolver nodo.asignaciones;
        si no para cada hijo en compleciones(nodo,costes) hacer
            cota:=acotar(hijo,costes);
            poner((cota,hijo),Montículo);
        fsi;
    fmientras
    devolver ∅;

Función acotar(nodo,costes[1..N,1..N]) dev cota
    cota:=nodo.coste;
    para columna desde nodo.último_asignado+1 hasta N hacer
        mínimo=∞;
        para cada fila en nodo.filas_no_asignadas hacer
            si costes[columna,fila]<mínimo entonces
                mínimo:=costes[columna,fila];
            fsi
        fpara
        cota:=cota+mínimo;
    fpara
    devolver cota;
```

```
Función compleciones(nodo,costes[1..N,1..N]) dev lista_nodos
        lista:=crearLista();
        para cada fila en nodo.filas_no_asignadas hacer
                hijo:=crearNodo();
                hijo.último_asignado:=nodo.último_asignado+1;
                hijo.asignaciones=nodo.asignaciones;
                hijo.asignaciones[hijo.último_asignado]:=fila;
                hijo.coste:=nodo.coste+costes[hijo.último_asignado,fila];
                hijo.filas_no_asignadas:=nodo.filas_no_asignadas;
                eliminar(fila,hijo.filas_no_asignadas);
                añadir(hijo,lista);
        fpara;
devolver lista;
```

Coste

En este caso únicamente podemos hallar una cota superior del coste del algoritmo por descripción del espacio de búsqueda.

En el caso peor se generan (k-1) hijos por cada nodo del nivel k, habiendo n. Por tanto, el espacio a recorrer siempre será menor que **n!**.