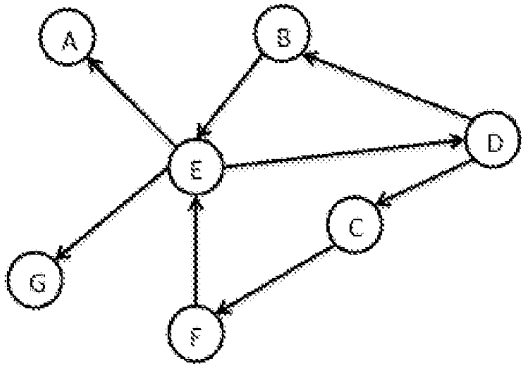
FEBRERO 2016 – SEMANA 1 (MODELO A)

1. Sea el grafo dirigido de la siguiente figura:



¿Cuál de las siguientes afirmaciones es <u>cierta</u>?

(a) Un posible recorrido en anchura visitaría los nodos en orden {D, B, C, E, F, A, G}, si se toma el nodo D como nodo de partida.

(b) Un posible recorrido en profundidad visitaría los nodos en orden {D, B, E, A, G, C, F}, si se toma el nodo D como nodo de partida.

(c) El nodo E es un punto de articulación.

(d) Todas las afirmaciones anteriores son ciertas.

- <u>Punto de Articulación (PDA)</u>

  Es un nodo que, si se elimina, el grafo deja de ser conexo.

- <u>Búsqueda de Puntos de Articulación sobre un Grafo</u>
  1. Recorrido en profundidad del grafo (Árbol de Recubrimiento). A cada nodo se le asigna un orden.
  2. Recorrido del Árbol de Recubrimiento en Postorden. A cada nodo se le asigna un valor bajo[v]. Mínimo entre:
     - orden[v]
     - orden[w] (w: nodo con arista de retroceso)
     - bajo[x] (x: nodo hijo de v)
  3. Puntos de Articulación:
     - La raíz será PDA si tiene más de un hijo
     - v es PDA si tiene un hijo w/bajo[w] >= orden[v]
- <u>Conectividad k</u>

  Un grafo tiene conectividad k si al eliminar k-1 nodos cualesquiera, el grafo sigue siendo conexo.

- <u>Recorrido Postorden de Árboles</u>

  El recorrido de los nodos de izquierda a derecha, y, por último, la raíz.

<u>Respuesta D)</u>

2. Indica cuál de las siguientes afirmaciones es <u>cierta</u>:

(a) El coste del algoritmo para fusionar dos subvectores ordenados que se utiliza en la ordenación por fusión es O(n), siendo n la suma de los tamaños de los dos vectores.

Cierto. La ordenación por fusión es la llamada "Mergesort".

(b) El coste del algoritmo que soluciona el problema del coloreado de grafos mediante el esquema de vuelta atrás es $O(n^m)$, siendo m el número de colores y n el número de nodos.

Falso. El coste es $O(n \cdot m^2)$

(c) El coste del algoritmo de Kruskal es $O(n^2)$, siendo n el número de nodos del grafo.

Falso. El coste de los distintos algoritmos es:

- Prim: $O(n^2)$
- Kruskal: O(a log n)

(d) El coste del algoritmo de ordenación rápida en el caso mejor es $O(n^2)$, siendo n el número de elementos del array.

Falso. El coste de Quicksort es:

- Normal: $O(n^2)$
- Mejorado: O(n log n)

<u>Respuesta A)</u>

3. Considérese el vector v[1..n] = [2,3,5,3,4,5,7,5,6]. Indica cuál de las siguientes opciones es cierta:

(a) El vector v es un montículo de mínimos.

(b) El vector v no es un montículo de mínimos porque el elemento v[5] = 4 debe ser flotado.

(c) El vector v no es un montículo de máximos porque el elemento v[5] = 4 deb ser hundido.

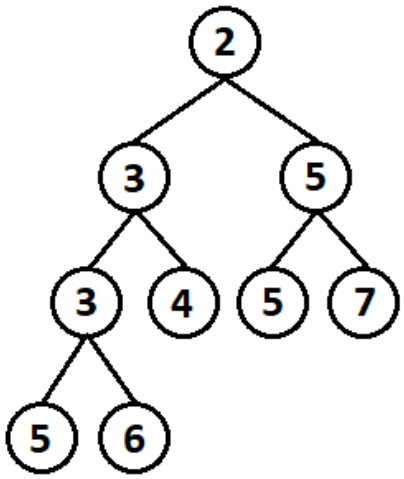
(d) Ninguna de las anteriores.

- Flotar: Intercambiar por el nodo inmediatamente superior.

- Hundir: Intercambiar por el nodo hijo de menor valor.

La respuesta B) es falso, ya que si v[5] se flotase no podría ser un montículo de mínimos.

La respuesta C) es falso, ya que v[5] = 4 no tiene nodos hijos.



Respuesta A)

4. Sea el problema de la mochila, en el que tenemos una mochila de capacidad M, n objetos con beneficios b_1, b_2, b_3, ... b_n y pesos p_1, p_2, p_3, ... p_n. El objetivo es maximizar el valor de los objetos transportados, respetando la limitación de la capacidad impuesta M. Indica, de los esquemas siguientes, cuál es el más adecuado y eficiente en el caso de que cada objeto puede meterse en la mochila entero o fraccionado.

(a) El esquema voraz utilizando como criterio de selección escoger el objeto de más valor de los que quedan.

(b) El esquema voraz utilizando como criterio de selección escoger el objeto cuyo valor por unidad de peso sea el mayor de los que quedan.

(c) El esquema de vuelta atrás calculando todas las soluciones posibles y escogiendo la mejor.

(d) El esquema de ramificación y poda.

Respuesta B)

5. Para resolver determinado problema hemos diseñado un algoritmo de tipo divide y vencerás. ¿Cuál de las siguientes afirmaciones es falsa?:

(a) La resolución debe alcanzar un caso trivial que se pueda resolver sin realizar nuevas descomposiciones.

(b) El problema se resuelve por divisiones sucesivas en subproblemas, que pueden ser de mayor 0 de menor tamaño que el de partida.

(c) Los algoritmos basados en este esquema pueden requerir un paso de combinación de las soluciones parciales.

(d) El algoritmo de la búsqueda binaria aplica la estrategia divide y vencerás.

Respuesta B)

6. Sea un procesador que ha de atender n procesos. Se conoce de antemano el tiempo que necesita cada proceso. Se desea determinar en qué orden se han de atender los procesos para minimizar la suma del tiempo que los procesos permanecen en el sistema. En relación a este problema, ¿cuál de las siguientes afirmaciones es <u>cierta</u>?

(a) El coste del algoritmo voraz que resuelve el problema es, en el mejor de los casos, $O(n^2)$.

Falso. Sería un coste de O(n log n)

(b) Suponiendo tres clientes con tiempos de servicio $t_1 = 5$, $t_2 = 10$ y $t_3 = 3$, el tiempo mínimo de estancia posible es de 26 segundos.

Falso. $t_3 < t_1 < t_2 \rightarrow 3 + (3 + 5) + (3 + 5 + 10) = 3 + 8 + 18 = 29$ segundos $\neq$ 26 seg.

(c) Suponiendo tres clientes con tiempos de servicio $t_1 = 5$, $t_2 = 10$ y $t_3 = 3$, el tiempo mínimo de estancia posible es de 31 segundos.

Falso. 29 seg. $\neq$ 26 seg.

(d) Todas las afirmaciones anteriores son falsas.

Cierto.

<u>Respuesta D)</u>

<u>PROBLEMA (4 Puntos)</u>

Dado el conjunto de caracteres alfabéticos, se quieren generar todas las palabras de cuatro
letras que cumplan las siguientes condiciones:

- ***La primera letra debe ser vocal.***
- ***Sólo pueden aparecer dos vocales seguidas si son diferentes.***
- ***No puede haber ni tres vocales ni tres consonantes seguidas.***
- ***Existe un conjunto C de parejas de consonantes que no pueden aparecer seguidas.***

Desarrolla un algoritmo que permita solucionar este problema con el menor coste.
La resolución del problema debe incluir, por este orden:

1. Elección del esquema <u>más apropiado</u>, el esquema general y explicación de Su aplicación al problema (0,5 puntos).

El esquema más apropiado es Vuelta Atrás. El árbol de búsqueda tiene como nodo raíz la palabra vacía.

En el primer nivel, cada nodo hijo de la raíz contendrá una de las posibles vocales. Cada nodo tiene como hijos el resultado de añadir una letra, para cada una de las letras disponibles.

En Completable, habrá que comprobar que la palabra generada cumple con las condiciones del enunciado.

```
fun VueltaAtras (v: Secuencia, k: entero)
    {v es una secuencia k-prometedora}
    IniciarExploraciónNivel(k)
    mientras OpcionesPendientes(k) hacer
        extender v con siguiente opción
        si SoluciónCompleta(v) entonces
            ProcesarSolución(v)
        sino
            si Completable (v) entonces
                VueltaAtras(v, k+1)
            fsi
        fsi
    fmientras
ffun
```

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

Para representar una palabra se utilizará un vector de 4 caracteres. Nos interesa añadir, además, un campo que indique el número de letras incorporadas, para no tener que hacer una comprobación lineal cada vez que se quiera incorporar una letra.

Por tanto, utilizaremos un registro con dos campos, *palabra* e *índice*. La tupla se puede definir así:

```
ensayo = tupla
  palabra: arreglo[1..4] de a..z
  indice: entero
  ftupla
```

3. Algoritmo <u>completo</u> 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.

Una vez establecida la forma de representar las palabras, pasamos a detallar el refinamiento del algoritmo dado anteriormente.

Consideramos que la función *compleciones* devuelve una lista de palabras resultado de añadir una letra nueva a la palabra que se le pasa como argumento. Si la palabra está completa se considera válida y se devuelve como resultado.

La función compleciones solo genera palabras posibles, es decir, que es necesario que posteriormente cumplan las restricciones impuestas en el enunciado. En el caso que nos ocupa tenemos coma condiciones de poda las siguientes:

- Primera letra vocal
- Dos vocales seguidas sólo si son distintas
- No tres vocales ni consonantes seguidas

- No pertenencia al conjunto de pares C

La función *compleciones* puede escribirse así:

```
fun compleciones (e : ensayo) dev lista de palabra
   lista-compleciones ← lista-vacía
   para cada letra en {a,b,c, .., z} hacer
   hijo ← añadir-letra (e, letra};
   lista-compleciones ← añadir(lista-compleciones,hijo)
   fpara
   dev lista-compleciones
ffun
```

Donde la función añadir-letra es la única función de modificación de la estructura de datos *ensayo* que necesitamos, y puede definirse así:

```
fun añadir-letra (e:ensayo, l:letra) dev ensayo
   nuevo-ensayo ← e;
   nuevo-ensayo.indice ← nuevo-ensayo.indice + 1;
   nuevo-ensayo.palabra[nuevo-ensayo.indice] ← e;
      dev nuevo-ensayo
ffun
```

Con respecto a la función condiciones de poda, esta queda como sigue:

```
fun condiciones de poda (e:ensayo} dev bool
   dev condicion1(e) ^ condicion2(e) ^ condicion3(e) ^ condicion4(e)
ffun
```

La primera condición dice que la primera letra ha de ser una vocal:

```
fun condicionl ( e:ensayo) dev bool
   dev vocal (e.palabra[1])
ffun
```

donde hemos utilizado una función auxiliar vocal que nos será útil para implementar las siguientes condiciones:

```
fun vocal (l:letra) dev bool
   dev pertenece(l,{a,e,i,o,u})
ffun

fun consonante (l:letra) dev bool
   dev ¬ vocal (l)
ffun
```

La segunda condición dice que solo pueden aparecer dos vocales seguidas si son diferentes. Podemos implementarla reescribiéndola así: la segunda condición se cumple si la palabra tiene menos de dos letras, o si la última no es vocal, o si la última no es igual a la penúltima.

Sólo comprobamos sobre las dos últimas porque el resto de la palabra ha debido pasar ya las condiciones de poda en el momento de ser generado.

```
fun condicion2 (e:ensayo) dev bool
   dev e.indice < 2 V
   consonante(ensayo.palabra[e.indice]) V
   e.palabra[e.indice] ≠ e.palabra[e. indice-1]
ffun
```

La tercera condición consiste en que no haya tres letras seguidas del mismo tipo (vocales o consonantes). De nuevo, teniendo en cuenta que solo debe comprobarse que la última letra introducida no hace que se viole ninguna condición, puede rescribirse así: la condición se cumple si la palabra tiene menos de tres letras, o bien la última y la penúltima son de distinto tipo, o bien la última y la antepenúltima son de distinto tipo:

```
fun condicion3 (e:ensayo) dev bool
 i ← e.indice;
    dev i < 3 V
      vocal(t.palabra[i-1]) ≠ vocal(e.palabra[i]) V vocal(e.palabra[i-2]) ≠ vocal(e.palabra[i])
ffun
```

Por último, la última condición nos dice que no debe aparecer ninguna pareja de entre las de una lista negra C. De nuevo hay que comprobarlo sólo para las dos últimas letras:

```
fun condicional4 (e:ensayo) dev bool
i ← e.indice;
dev ¬pertenece((e.palabra[i-1], e.palabra[i]), C)
ffun
```

Solo nos resta escribir la función principal que debe llamar a vuelta-atrás tomando como argumento ensayo-vacío:

```
fun caracteres dev lista de ensayo
   dev vuelta-atrás (ensayo-vacío)
ffun
```

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

El problema tiene un tamaño acotado y es, por tanto, de coste constante. Si lo generalizamos al problema de generar palabras de n letras con m restricciones, el coste estará en relación directa con el tamaño del árbol de búsqueda, que está acotado por 28^n, siempre que la verificación de las condiciones de poda se pueda realizar en tiempo constante, como en las cuatro condiciones que teníamos.

Este factor exponencial quiere decir que el problema es irresoluble, en la práctica, para valores grandes de n.