Ejercicio 6

Se han de procesar *n* tareas con instantes de terminación *ti* y tiempo de proceso *pi* . Se dispone de un procesador y las tareas no son interrumpibles. Una solución factible planifica todas las tareas de modo que terminan en, o antes de, su instante asignado. Diseñar un algoritmo voraz para encontrar la solución. Demostrar que, si existe solución, el algoritmo voraz la encuentra.

```
Función Procesar(T[1..n],P[1..n])
Inicio
        Sol[1..n]
        Mientras i<n
                Si P[i] <= T[i]
                        Sol[i]=i
                Fin si
        Fin mientras
Fin
```

Ejercicio 7

Dado un conjunto de *n* cintas cuyo contenido está ordenado y con *ni* registros cada una, se han de mezclar por pares hasta lograr una única cinta ordenada. La secuencia en que se haga la mezcla determina la eficiencia del proceso. Diseñar un algoritmo voraz que minimice el número de movimientos.

Ejemplo : se han de mezclar 3 cintas: *A* con 30 registros, *B* con 20 y *C* con 10.

Opción 1 : - Mezclar *A* con *B* requiere 50 movimientos (se obtiene *A'*)

- Mezclar *A'* con *C* requiere 60 movimientos

TOTAL = 110 movimientos

Opción 2 : - Mezclar *C* con *B* requiere 30 movimientos (se obtiene *A'*)

- Mezclar *A'* con *A* requiere 60 movimientos

TOTAL = 90 movimientos.

```
Función juntarcintas(R[1..n])
Inicio
        OrdenarPorMontículo(R)
        Sol[i]=R[i]
        Desde i←1 hasta n
                Sol=Mezcla(Sol[i],R[i+1])
        Fin desde
Fin
```

Ejercicio 8

Un automovilista ha de ir con su vehículo desde la población *A* hasta la población *B* siguiendo una ruta prefijada (por ejemplo la carretera N-340). Con el depósito completamente lleno puede hacer *X* km. El conductor conoce de antemano en qué lugares de la carretera existen gasolineras (la distancia entre dos gasolineras consecutivas es inferior a *X* km.). Diseñar un algoritmo que permita que el automovilista vaya de *A* a *B* repostando el *mínimo* número de veces posible (se supone que parte de *A* con el depósito lleno y que el coche no se averiará, ni tendrá un accidente, ni será abducido, etc. durante el trayecto). Indicar el coste del algoritmo y demostrar la optimalidad del criterio usado.

```
//precondición ptos km de las gasolineras ordenados
Función MínimoRepostaje(G[1..N] array entero, X, B entero): Sol[1..N]
sol[1..N]: int
VAR
      I[1..N] : array de enteros
INICIO
        I← OrdenadorPorMontículo(G): //de manera decreciente
        i←1, j←1
        Mientras X<B
                Si X < I[i]
                        Sol[j]←I[i]
                        i++, j ++
                Si no
                        i++
                fin si
        fin mientras
fin
```

Ejercicio 9

El informático megalómano comienza a estar harto de tener que esperar tanto rato para poder escuchar sus canciones favoritas en su extensa colección de casettes. Como no tiene suficiente dinero para comprarse un grabador de CD's ni tampoco quiere desperdiciar cinta ha decidido diseñar un algoritmo que dadas *n* canciones, sus respectivas duraciones en segundos (dur[*i*] es la duración de la canción *i*-ésima, 1 ≤ *i* ≤ *n*), y unas "frecuencias" de acceso (f[*i*] es la frecuencia de acceso a la canción *i*-ésima), encuentre la disposición de las *n* canciones que minimiza el tiempo medio de acceso. Es decir, el resultado del algoritmo es un vector pos que representa a una permutación de las *n* canciones, de tal modo que si *pos*[*j*] es la canción que ocupa la posición *j*-ésima en la cinta entonces

$$\sum_{j=1}^{n} \left(f[pos[j]] \cdot \sum_{k=1}^{j-1} dur[pos[k]] \right)$$

es mínima.

Nuestro informático megalómano ya empieza a desesperar. Hay que echarle una mano!

```
Función CancionesFavoritas(C[1..N] array canciones): Sol[1..N]
sol[1..N]: int
VAR
      I[1..N] : array de enteros
INICIO                                          //por duración entre frecuencia
        I← OrdenadorPorMontículo(G):    //por mayor  frecuencia
fin                                             //por mayor duración
```

Ejercicio 10

La jefatura de estudios de la BIF encara el siguiente problema cada cuatrimestre: una vez fijados los horarios de las *n* clases a impartir (una clase es una tripleta <*día_semana*,*hora_inicio*,*hora_fin*>) debe asignarse un aula a cada clase. El único requisito para que la asignación sea válida es que no puede haber dos clases el mismo día y a la misma hora que compartan aula. Hallar un algoritmo eficiente (polinómico en *n*) que encuentre una asignación válida de aulas a las clases de modo que se emplee el mínimo número de aulas posible.

Suponemos que la *hora_inicio* y la *hora_fin* de una clase son horas en punto o horas y media (ej: 10:30-12:30) y que una clase puede durar como mínimo media hora y no más de 4 horas. Ninguna clase acaba en un día diferente al día en que se inicia.

```
Función Horarios (ClaseDia[1..N], ClaseC[1..N], ClaseF[1..N], AulaDia[1..N], AulaC[1..N],
AulaF[1..N] array) : solAula[1..N]
        Desde i← 1 hasta n hacer
                Sol[i]= vacia
        I←1, j←1        //Contador de Clases y Aulas
        Mientras i  < N  hacer
                Si (aula == vacia)
                        Solución[j]←insetarClase[i]
                        I++;
                Si no
                        Si ClaseDia[i] <> Auladia[j]
                                Solución[j]←insetarClase[i]
                                        i++
                        Si no
                                Si ClaseC [i] > solAulaFin[j]
                                        Solución[j]←insetarClase[i]
                                        i++
```