| UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA | **Programación III**<br>Solución<br>**Prueba Presencial** | **Prueba Presencial**<br>**Original**<br>Septiembre de 2005<br>Duración: **2 horas**<br>Material permitido: **NINGUNO** |
|---|---|---|

**Cuestión 1 (1 puntos).** En la práctica obligatoria del presente curso 2005/2006 se ha tenido que diseñar y desarrollar un algoritmo para resolver el juego del Su-doku. Codifique en java o en modula-2 un algoritmo iterativo que recorra el cuadrado de 3x3 casillas que corresponda a una casilla que ocupa las posiciones *i,j* del tablero.

Solución:

Se trataba de encontrar la relación entre las posiciones *i,j* del tablero y las posiciones de comienzo del cuadrado de 3x3 que les corresponde en el tablero del Su-doku 9x9. Supongamos que el tablero tiene como índices 0..8, 0..8 y que i y j son de tipo entero, la relación se puede establecer de la siguiente manera:

```
int coordFilaInicioCuadrado =  (i / 3) * 3; // división entera
int coordColumnaInicioCuadrado =  (j / 3) * 3; // división entera
```

ahora sólo queda hacer el recorrido:

```
for (int k= coordFilaInicioCuadrado; k < coordFilaInicioCuadrado+3; k++){
      for (int l= coordColumnaInicioCuadrado; l < coordColumnaInicioCuadrado+3; l++){
                  procesar(tab[k][l]);
      }
}
```

**Cuestión 2 (2 puntos).** Sea el famoso problema de la mochila. Se dispone de *n* objetos y una mochila. Para $i = 1,2,\ldots,n$, el objeto *i* tiene un peso positivo $w_i$ y un valor positivo $v_i$. La mochila puede llevar un peso que no sobrepase *W*. El objetivo es llenar la mochila de tal manera que se maximice el valor de los objetos almacenados, respetando la limitación de peso impuesta. Indique qué esquema o esquemas considera más adecuados para resolver este problema en los siguientes casos:

a) Los objetos se pueden fraccionar, luego se puede decidir llevar una fracción $x_i$ del objeto *i*, tal que $0 \leq x_i \leq 1$ para $1 \leq i \leq n$.
b) Los objetos no se pueden fraccionar, por lo que un objeto puede o no ser añadido, pero en éste último caso, sólo se añade 1.

Además de nombrar el esquema o esquemas, explica el porqué de su elección, los aspectos destacados de cómo resolverías el problema y el coste asociado. No se piden los algoritmos.

Solución:

En el caso a) se puede utilizar el esquema voraz ya que existe una función de selección que garantiza obtener una solución óptima. La función de selección consiste en considerar los

objetos en orden decreciente de $v_i/w_i$. El coste está en O($n$ log $n$), incluyendo la ordenación de los objetos. Ver página 227 del libro base de la asignatura.

En el caso b) no se puede utilizar el esquema voraz ya que no existe una función de selección que garantice obtener una solución óptima. Al ser un problema de optimización se puede utilizar el esquema de ramificación y poda. Se podrían seleccionar los elementos en orden decreciente de $v_i/w_i$. Así, dado un determinado nodo, una cota superior del valor que se puede alcanzar siguiendo por esa rama se puede calcular suponiendo que la mochila la rellenamos con el siguiente elemento siguiendo el orden decreciente de $v_i/w_i$. El coste en el caso peor sería de orden exponencial, ya que en el árbol asociado al espacio de búsqueda, cada nodo tendrá dos sucesores que representarán si el objeto se añade o no a la mochila, es decir, $O(2^n)$. Sin embargo, sería de esperar que, en la práctica, el uso de la cota para podar reduzca el número de nodos que se exploran.

**Cuestión 3 (3 puntos).** Un dentista pretende dar servicio a $n$ pacientes y conoce el tiempo requerido por cada uno de ellos, siendo $ti$ , $i= 1,2,\ldots,n$ el tiempo requerido por el paciente $i$. El objetivo es minimizar el tiempo total que todos los clientes están en el sistema, y como el nº de pacientes es fijo, minimizar la espera total equivale a minimizar la espera media. Se pide:

1. Identificar una función de selección que garantice que un algoritmo voraz puede construir una planificación óptima. (0.5 puntos)
2. Hacer una demostración de la optimalidad de dicha función de selección. (2.5 puntos)

Solución: los dos apartados están solucionados en el libro base de la asignatura, apartado 6.6.1, página 231.

**Problema (4 puntos).** 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. La resolución de este problema debe incluir, por este orden:

1. Elección del esquema más apropiado y explicación de su aplicación al problema (1 puntos).
2. Descripción de las estructuras de datos necesarias (0.5 puntos).
3. Algoritmo completo a partir del refinamiento del esquema general (2 puntos).
4. Estudio del coste del algoritmo desarrollado (0.5 puntos).

Solución

1. No se puede encontrar una función de selección que garantice, sin tener que reconsiderar decisiones, una elección de los activos que cumpla con la restricción del enunciado, por ello no se puede aplicar el esquema voraz. Tampoco se puede dividir el problema en subproblemas que al combinarlos nos lleven a una solución. Al no ser un problema de optimización, el esquema de exploración de grafos más adecuado es el esquema vuelta-atrás. Vuelta atrás es un recorrido en profundidad de un grafo dirigido implícito. En él una solución puede expresarse como una n-tupla [$x_1$, $x_2$, $x_3$,… $x_n$], donde cada $x_i$, representa una decisión tomada en la etapa i-ésima, de entre un conjunto finito de alternativas.

Descripción algorítmica del esquema:

```
función vuelta-atrás(e: ensayo)
        si valido(e) entonces
                dev e
        sino
                listaEnsayos ← complecciones(e)
                mientras ¬ vacia(listaEnsayos) ∧ ¬resultado hacer
                        hijo ← primero(listaEnsayos)
                        listaEnsayos ← resto(listaEnsayos)
                        si condicionesDePoda(hijo) entonces
                                resultado ← vuelta-atrás(hijo)
                        fsi
                fmientras
                dev resultado
        fsi
ffunción
```

Otra posible descripción:

```
función vuelta-atrás(v[1..k]: nTupla)
        si solucion(v) entonces
                dev v
        sino
                Para cada vector w k+1-prometedor hacer
                        si condicionesDePoda(w) entonces
                                vuelta-atrás(w[1..k+1])
                        fsi
                fpara
        fsi
ffunción
```

En este caso, el espacio de búsqueda es un árbol de grado 2 y altura n+1. Cada nodo del i-ésimo nivel tiene dos hijos correspondientes a si el i-ésimo activo va a un socio o al otro.

Para poder dividir el conjunto de activos en dos subconjuntos disjuntos, de forma que cada uno de ellos tenga el mismo valor, su valor debe ser par. Así, si el conjunto inicial de activos no tiene un valor par el problema no tiene solución.

2. Descripción de las estructuras de datos necesarias.

   - El conjunto de activos y sus valores se representa en un array de enteros v= [$v_1$, $v_2$, $v_3$,… $v_n$], donde cada $v_i$ representa el valor del activo i-ésimo.
   - La solución se representa mediante una array de valores x= [$x_1$, $x_2$, $x_3$,… $x_n$], donde cada $x_i$, podrá tomar el valor 1 o 2 en función de que el activo i-ésimo se asigne al subconjunto del un socio o del otro.
   - Un array de dos elementos, suma, que acumule la suma de los activos de cada subconjunto.

3. Algoritmo completo a partir del refinamiento del esquema general.

   En esta solución se sigue el segundo de los esquemas planteados en el apartado 2.
   Una posible condición de poda consistirá en que se dejarán de explorar aquellos nodos que verifiquen que alguno de los dos subconjuntos que se van construyendo tiene un valor mayor que la mitad del valor total.

**función** Solucion (k: entero; suma: array[1..2]) **dev** booleano

```
        si (k = n) AND (suma[1] = suma [2]) entonces
            dev Verdadero
        sino
            dev Falso
        fsi
    ffuncion
```

```
función SeparacionSociosMitad (v: array[1..n], k: entero, suma: array[1..2], sumaTotal: entero)
                                                    dev x: array[1..n]; separacion: boolean

    si Solucion (k, suma) entonces
            dev x, Verdadero
    sino
        para i desde 1 hasta 2 hacer
            x[k] ← i
            suma[i] ← suma[i] + v[k]
            si suma[i] <= (sumaTotal DIV 2) entonces
                si k < n entonces
                    x ← SeparacionSociosMitad(v, k+1, suma, sumaTotal)
                fsi
            sino
                suma[i] ← suma[i] - v[k]
            fsi
        fpara
    fsi
ffunción
```

```
    función ProblemaSeparacionSociosMitad (v: n-tupla) dev x: n-tupla; separacion: boolean
        para i desde 1 hasta n hacer
            sumaTotal ← sumaTotal + v[i]
        fpara
        si par(sumTotal) entonces
            dev SeparacionSociosMitad (v, 1, suma[0,0], sumaTotal)
        sino
            dev 0
        fsi
    ffuncion
```

Llamada inicial ProblemaSeparacionSociosMitad (v);

4. Estudio del coste del algoritmo desarrollado.

   En este caso, el coste viene dado por el número máximo de nodos del espacio de búsqueda, esto es: $T(n) \in O(2^n)$