Examen de Teoría de la Programación

E. U. ING. TEC. EN INFORMÁTICA DE OVIEDO

Final Febrero – Curso 2004-2005 14 de febrero de 2005

DNI _____ Nombre _____ Apellidos _____

Titulación: ☐ Gestión ☐ Sistemas

Soluciones

3. (1,5 puntos) Especificar una función que devuelva un valor booleano que indique si el subvector *v[1..n]* está o no ordenado crecientemente, cuya cabecera es la siguiente:
Fun ordenado_crecientemente(a: vect; n: entero) dev (r: booleano)
Donde Tipo vector= vect [1..1000] de entero.
a) Realizar la especificación formal de la función con la técnica pre/post. Poner en la precondición los límites apropiados para *n*.
b) Supón que queremos implementar esta función en Java como método privado de una clase. Escribir en Java el *esqueleto* de esta función e implementar con asertos la especificación formal del apartado anterior (no hace falta escribir el código que realice la función, sólo la cabecera del método y el código de los asertos).

a)
{Q≡ 1≤ n ≤ 1000}
Fun ordenado_crecientemente(a: vect; n: entero) dev (r: booleano)
{R≡ ∀i ∈ {1..n-1}.a[i]<a[i+1]}

b)
```
private boolean estaOrdenado(int[] a, int n)
{
        boolean ordenado= true;
        assert (n>=1 && n<=1000);
        // comprueba si el vector está ordenado

        // no hace falta aserto para la postcondición, ya que la propia función es la que hace la comprobación
        return ordenado;
}
```

4. (2 puntos) Ante el enfrentamiento de dos equipos en un play-off definimos la función P(i,j) que se define como la probabilidad de que el equipo A gane la eliminatoria cuando le quedan por ganar i partidos frente a los j partidos que le quedan por ganar al equipo B. La definición matemática de la función es la siguiente:

$$P(i,j)=\begin{cases} 1 \text{ si } i=0 \text{ y } j>0 \\ 0 \text{ si } i>0 \text{ y } j=0 \\ p*P(i-1,j)+q*P(i,j-1) \text{ si } i>0 \text{ y } j>0 \end{cases}$$

Donde *p* es la probabilidad de que A gane un partido y *q* la probabilidad de que gane B.
a) Escribir el pseudocódigo del algoritmo divide y vencerás que implemente esta función.
b) Calcular la complejidad de esta función divide y vencerás.
c) Escribir el pseudocódigo para esta función utilizando la técnica de programación dinámica.
d) Calcular la complejidad de la función con programación dinámica.

Ver el capítulo de Programación Dinámica en el **Cuaderno Didáctico Nº 31. Técnicas de Diseño de Algoritmos.** Ver sección de bibliografía de la página web.

5. (1,5 puntos) Supongamos que tenemos n hombres y n mujeres y dos matrices M y H que contienen las preferencias de los unos por los otros. Más concretamente, la fila M[i,·] es una ordenación (de mayor a menor) de las mujeres según las preferencias del i-ésimo hombre y, análogamente, la fila H[i,·] es una ordenación (de mayor a menor) de los hombres según las preferencias de la i-ésima mujer.

Diseñar un algoritmo *backtracking* que encuentre, si es que existe, un emparejamiento de hombres y mujeres tal que todas las parejas formadas sean estables. Diremos que una pareja (h,m) es estable si no se da ninguna de estas dos circunstancias:
1) Existe una mujer m1 (que forma la pareja (h1,m1)) tal que el hombre h la prefiere sobre la mujer m y además la mujer m1 también prefiere a h sobre h1.
2) Existe un hombre h2 (que forma la pareja (h2,m2)) tal que la mujer m lo prefiere sobre el hombre h y además el hombre h2 también prefiere a m sobre la mujer m2.
Suponer que disponemos de una función: booleana *estable()* con los parámetros que sean necesarios que nos devuelve información sobre una pareja en concreto.
a) Declarar las estructuras de datos necesarias para almacenar el estado del problema.
b) Representar un esquema del árbol correspondiente al backtracking para este problema, en cada nodo representa el estado correspondiente.
c) Realizar el pseudocódigo del algoritmo.

Este problema se parece bastante al problema de las n reinas visto en clase.

a)
Para almacenar un estado nos sirve con un vector de enteros cuyo tamaño se corresponde con el número de parejas que queremos formar. Cada posición se corresponderá con un hombre y el valor de cada posición representará la mujer con la que lo hemos emparejado (o viceversa).
int[] x;
Podemos añadir un vector para marcar las mujeres o hombres que quedan libres y alguna estructura auxiliar más.

b)

c)

```
PROCEDURE Parejas(hombre:entero;VAR exito:BOOLEAN);
VAR mujer,prefiere,preferencias:CARDINAL;
BEGIN
        prefiere:=0; (* recorre las posibles elecciones del hombre *)
        REPEAT
                INC(prefiere);
                mujer:=M[hombre,prefiere];
                IF Estable(hombre,mujer) THEN
                BEGIN
                        X[hombre]:=mujer;
                        IF hombre<n THEN
                                Parejas(hombre+1,exito);
                                IF NOT exito THEN
                                        // no haría falta borrar nada
                                END
                        ELSE
                                exito:=TRUE;
                        END
                END
        UNTIL (prefiere=n) OR exito;
END;
```

6. (1 punto) El problema del cambio consiste en devolver una cantidad con el mínimo número de monedas posible. Disponemos de un heurístico que consiste en elegir la moneda con mayor valor que no supere la cantidad que queda por devolver. Este heurístico nos permite aplicar la técnica del devorador obteniendo la solución óptima para el sistema monetario Euro.
a) Supongamos que para el mismo problema tenemos un número limitado de monedas disponibles (aunque siempre suficientes para devolver el cambio), justifica si el heurístico anterior sigue siendo óptimo o plantea una demostración si no lo sigue siendo.

a)
Ver el capítulo de Devorador en el **Cuaderno Didáctico Nº 31. Técnicas de Diseño de Algoritmos.** Ver sección de bibliografía de la página web. Aparece esto mismo con el sistema monetario en pesatas.
La cuestión es encontrar un contraejemplo con el sistema mometario euro en el que el número de monedas devuelto usando el heurístico no sea el óptimo.
Devolver 0,60 €
Monedas disponibles: 0,50 €, 0,20 €, 0,02 €.

Utilizando el heurístico:
0,50 € + 5 * 0,02 € → 6 monedas
Pero existe una alternativa:
3 * 0,20 € → 3 monedas

Por tanto, queda demostrado que el heurístico no devuelve el óptimo cuando tenemos limitado el número de monedas.

7. (1 punto) Representa en un diagrama estático de clases las clases e interfaces básicas necesarias con sus métodos correspondientes para la implementación de un problema de ramificación y poda de forma que estas se puedan utilizar sin cambios para implementar cualquier problema que se quiera resolver con esta técnica. Si no recuerdas los signos UML que representan las relaciones entre clases, simplemente pon una flecha y el nombre de la relación: implementa, hereda, agrega, etc.

Ver sección de materiales de la página web. 9 Ramificación y Poda. Estructura para implementación del problema de asignación agentes-tareas. Diagrama de clases y plantilla para implementación en Java.