Examen de Teoría de la Programación

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

Final Septiembre – Curso 2004-2005 10 de septiembre de 2005

DNI _____ Nombre _____ Apellidos _____

Titulación: ☐ Gestión ☐ Sistemas

3. (1 punto) Sea una clase *Capicua* que contiene como datos miembro privados un vector *a* y un entero *n*. Queremos establecer una invariante de clase que permita asegurar que el vector seguirá siendo capicúa después de cualquier operación pública de la clase.
Donde Tipo *a:* vector [1..1000] de entero y *n*: entero

a) **Suponer que queremos implementar la clase *Capicua* en Java. Escribir en Java el método que permita verificar la invariante**
b) **Implementar con asertos la verificación de la invariante para el método *public void insertar(int valor)* (no hace falta escribir el código de la clase, sólo la cabecera del método y el código de los asertos).**

```
public class Capicua
{
	[…]
	public void insertar(int valor)  (b)
	{
		[…]
		assert esCapicua();
	}
[…]
	private boolean esCapicua()	(a)
	{
		boolean seCumple= true;

		for (int i= 0; i<n/2; i++)
		{
			if (a[1+i]!=a[n-i])
				seCumple= false;
		}

		return seCumple;
	}
}
```

4. (1,5 puntos) El problema de la mochila 0/1 consiste en que disponemos de *n* objetos y una "mochila" para transportarlos. Cada objeto *i*= 1,2, ...*n* tiene un peso w_i y un valor v_i. La mochila puede llevar un peso que no sobrepase *W*. Los objetos no se pueden fragmentar: o tomamos un objeto completo o lo dejamos. El objetivo del problema es maximizar valor de los objetos respetando la limitación de peso.
Para resolver este problema aplicaremos la técnica de vuelta atrás (Backtracking).

a) **Completar el código Java para dar solución a este problema con la técnica de Backtracking (escribirlo en los recuadros preparados a tal efecto).**

```
	// Datos del problema
	Objeto[] objetos;		// objetos que podemos meter en la mochila
	int w;				// peso máximo que puede llevar la mochila
	int[] x;			// que objetos tenemos en la mochila
	int pesoMochila, valorMochila;
	// Guarda la mejor solución hasta el momento
	int valorMejor;
	int[] sol;
[…]
	public void rellenar(
```

int i

```
)
	{
		for (
```

int k= 0; k<=1; k++

```
		)
		{
```

```
                    if ((pesoMochila+objetos[i].getPeso()*k)<=w)
                    {
                          x[i]= k;
                          pesoMochila+= objetos[i].getPeso()*k;
                          valorMochila+= objetos[i].getValor()*k;
                          if (i<objetos.length-1)
                                rellenar(
```

i+1

```
                                      );
                          else
                                if (valorMochila>valorMejor)
                                {
                                      sol= x;
                                      valorMejor= valorMochila;
                                      imprimir(sol);
                                }
                          pesoMochila-= objetos[i].getPeso()*k;
                          valorMochila-= objetos[i].getValor()*k;

                    }  // del if
              }  // del for
        }  // del método public void rellenar
```

5. (2 puntos) En el nuevo superpuerto del Musel ampliará la terminal especializada en la manipulación de graneles sólidos, *European Bulk Handling Installation* (EBHI). Básicamente se trata de unas instalaciones que permiten almacenar distintos tipos de minerales y cargar / descargar buques. En el momento actual se dispone de varios tipos de minerales de los cuales se conoce el número total de toneladas almacenado y el valor por tonelada. Se pide el pseudocódigo del programa que permita calcular que cantidad de cada mineral se debe cargar en cada buque que atraca en la terminal del que conocemos su capacidad máxima de carga; este cálculo debe poder realizarse en el mínimo tiempo posible para comenzar las operaciones que conllevan la carga inmediatamente.

a) Razonar por qué se debería utilizar un algoritmo devorador.

Cuando nos piden que se ejecute en el mínimo tiempo posible, es necesaria la técnica que nos asegura acabar más rápido: devorador.

b) Describir un heurístico que proporcione una solución óptima.

Ordenar los minerales por el cociente valor / peso y coger el mayor disponible.

c) Escribir el pseudocódigo del algoritmo devorador.

```
procedure CargaMineral(mineral:conjunto_minerales;W:real):vector_fracciones;
begin
  Ordenar vector mineral de mayor a menor v/w;
  Inicializamos vector x a 0 (* no hay nada en la mochila *)
  while (i<n) and (peso_ya_cargado<W) do (* seguimos metiendo *)
    begin
      Siguiente mineral; (*Selección siguiente mineral del vector ordenado *)
      if (mineral[i].peso + peso_ya_cargado)<=W then x[i]:= 1
            else x[i]:= (W-peso_ya_cargado)/mineral[i].peso
      peso_ya_cargado:= mineral[i].peso + peso_ya_cargado;
    end;
    return x;
end;
```

6. (2,5 puntos) En el problema de la asignación de tareas a agentes, hay que asignar n tareas a n agentes, de forma que cada agente realizará sólo una tarea. Se dispone de una matriz de costes de ejecución de una tarea j por un agente i. El objetivo es minimizar coste total ejecutar las n tareas.

Dados 4 agentes: a..d y 4 tareas: 1..4. Y la siguiente matriz de costes:

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23

d	17	14	20	28

Nos planteamos resolver el problema mediante la técnica de ramificación y poda.

a) **Explicar como se calcula el heurístico de ramificación para el estado donde asignamos la tarea 3 al agente b, cuando ya tenemos asignada la tarea 1 al agente a.**
b) **Explicar como se calcula la cota inicial de poda y decir y razonar cuándo se produce el cambio de esta cota.**
c) **Representar el árbol de estados hasta que llegamos a la primera solución factible y dibujar los estados que quedan en la cola de prioridad en ese momento.**

Ver cuaderno didáctico Tecnicas de diseño de algoritmos capítulo Ramificación y Poda para ver la solución.