Examen de Teoría de la Programación

ESCUELA DE INGENIERÍA INFORMÁTICA – UNIVERSIDAD DE OVIEDO

Final Febrero – Curso 2010-2011 17 de enero de 2011

DNI _____ Nombre _____ Apellidos _____

Titulación: ☐ Gestión ☐ Sistemas

3. (1 punto) Sea la función:

- ***Tipo vector= array [1..1000] de entero;***
- ***fun maximoValor(a:vector;n:entero) dev (x:entero)***

que realiza extrae el máximo valor de los n primeros elementos del vector a.

a) Realizar la especificación formal de la función con la técnica pre/post.

- $\{Q \equiv 1 \leq n \leq 1000\}$
- Tipo vector= array [1..1000] de entero;
- fun *maximoValor*(*a:vector;n:entero*) dev (*x:entero*)
- $\{R \equiv (\forall i \in \{1..n\}.x \geq a[i])\}$
- Pero el implementador podría *inventarse* un valor x muy grande que no existiese en el vector:
- $\{R \equiv (\forall i \in \{1..n\}.x \geq a[i]) \wedge (\exists j \in \{1..n\}.x = a[j])\}$

b) Codificar en Java el esqueleto de esta función utilizando un método público: cabecera, precondición y postcondición (no hace falta codificar la funcionalidad del método en sí).

```
public int maximoValor(int[] a,int n)
{
      // precondición: al ser un método público mediante if
      if (n<0 || n>=a.length-1 /*n>=999*/)
            throw new IllegalArgumentException("El argumento n tiene un
valor no permitido: "+n);

      // Implementación
      …
      // postcondición
      assert comprobarMaximoValor(a,n,max):"Valor de max= "+max;
      return max;
}

private boolean comprobarMaximoValor(int[] a, int n, int x) {
      boolean existe= false;
      boolean todos= true;
      for (int i= 0; i<n; i++)
      {
            if (x<a[i]) todos= false;     // condición para todo
            if (x==a[i]) existe= true;    // condición existe
      }

      return existe && todos;
}
```

4. (1,75 puntos) El problema de las "Parejas estables" consiste en que tenemos n hombres y n mujeres, y una matriz nxn que contiene la compatibilidad entre ellos. El problema consistiría en encontrar la manera de emparejarlos a todos, de tal forma que cada hombre acabe con una mujer diferente y la suma de las compatibilidades sea máxima.

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

```
public class Backtracking {
      int solucion[];
      int compatibilidadActual;
      int solucionOptima[];
```

```
      int compatibilidadOptima;
      int matrizValores[][];  // Compatibilidad para cada una de las parejas
      boolean  selecionado[];

    […]

private void ensaya(int pos) {
      for(int est=0;est<matrizValores.length;est++){
            if(!selecionado[est]){
                  solucion[pos]=est;
                  selecionado[est]=true;
                  compatibilidadActual+=matrizValores[pos][est];
                  if(pos==solucion.length-1){
                        if(compatibilidadActual>compatibilidadOptima){
                        compatibilidadOptima=compatibilidadActual;
                        for(int i=0;i<solucionOptima.length;i++)
                              solucionOptima[i]=solucion[i];
                        }
                  }
                  else ensaya(pos+1);
                  selecionado[est]=false;
                  compatibilidadActual-=matrizValores[pos][est];
            }
      }
}
```

5. (1,5 puntos) El problema de la mochila 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*. En el caso de que un objeto no se pueda meter entero, se fraccionará, quedando la mochila totalmente llena. El objetivo del problema es maximizar valor de los objetos respetando la limitación de peso. Queremos resolver este problema mediante un método voraz.

a) (0,5 puntos) Escribir pseudocódigo para la función heurística que permita alcanzar la solución óptima.

- Calcular valor por unidad de peso sea el mayor posible (valor / peso) de cada objeto.
- Ordenar los objetos de mayor a menos por este valor (el heurístico funcionaría sin este paso aunque también aumentaría la complejidad temporal)
- Seleccionar los objetos en este orden

b) (0,5 puntos) Escribir el código Java del método que permita obtener la solución a este problema mediante un algoritmo voraz (escribir exclusivamente el método que realiza el algoritmo voraz suponer ya cargados los arrays necesarios con los datos de los objetos, no realizar ninguna entrada / salida en el método).

```
/** Devuelve el array con la cantidad transportada de cada objeto,
    con el podemos calcular, el valor total o saber que objetos
    transportamos **/
public float[] algoritmo() {
int i = 0;
float pesoActual = 0;
for (int j = 0; j < numObjetos; j++) {
  pctSeleccionados[j] = 0; // 0% de cada objeto
}
do {
  // Proporciona el índice del objeto más adecuado de los que restan
  i = heuristicoObtenerObjeto();
  if (pesoActual + pesos[i] <= pesoMaximo) {
    pctSeleccionados[i] = 1; // se coge el objeto entero (100%)
    pesoActual += pesos[i];
  }
```

```
      else {  // Proporción del objeto para ajustarse a la mochila
        pctSeleccionados[i] = ( (pesoMaximo - pesoActual) / pesos[i]);
        pesoActual = pesoMaximo;
      }
    }
    while (pesoActual < pesoMaximo && i<numObjetos);
    return pctSeleccionados;
  }
```

c) (0,5 puntos) Qué ocurre si no se pudieran fraccionar los objetos. ¿Seguiría siendo óptimo el heurístico propuesto? Demostrar la respuesta.

El heurístico ya no sería óptimo.
Demostramos la respuesta con un contraejemplo:

Objeto	1	2	3
w_i	6	5	5
v_i	8	5	5
v_i/w_i	1,33	1	1

– Peso límite de la mochila: W=10

Empleando Heurístico, cogeríamos el objeto de mejor proporción, el 1 y no cabrían más en la mochila, resultado= 8
Podemos coger los objetos 2 y 3, que también cumplen los requisitos del problema (no hay fragmentación y no sobrepasamos el peso límite), resultado óptimo= 10
Hemos obtenido un mejor resultado que empleando el heurístico, con lo cual hemos demostrado que no es óptimo.

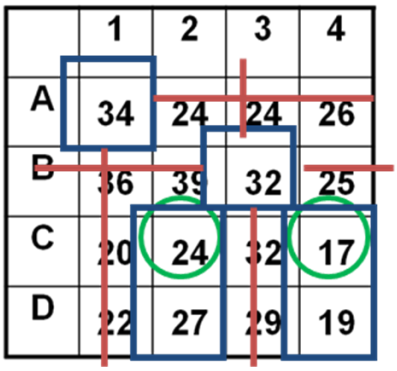
6. (1,5 puntos) Sea el problema de la asignación de tareas a agentes visto en el ejercicio 5. Se dispone de una matriz de costes de ejecución de una tarea j por un agente i.
Dados 4 agentes: a..d y 4 tareas: 1..4. Y la siguiente matriz de costes:

	1	2	3	4
A	34	24	24	26
B	36	39	32	25
C	20	24	32	17
D	22	27	29	19

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

a) (0,25 puntos) Explicar cómo 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.

El cálculo del heurístico de ramificación consiste en: La suma de lo ya asignado en cada estado más el caso mejor (mínimo de columnas) de lo que queda por asignar.
Al estado 1→a,3→b, le corresponde un coste de 34+32+24+17=107.



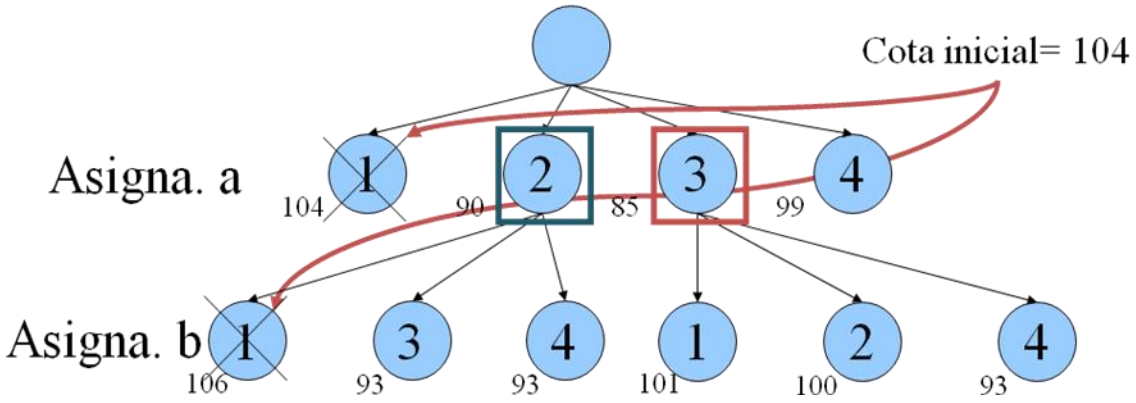
b) (0,25 puntos) Explicar cómo se calcula la cota inicial de poda y razonar cuándo se produce el cambio de esta cota.

Inicialmente, es la menor de la sumas de las dos diagonales de la matriz de costes.
Valor cota inicial: min(124,104)= 104.
Cuando en el desarrollo de los estados del árbol, lleguemos a una solución válida cuyo valor sea menor que el de la cota actual, se cambiará la cota a este nuevo valor.

	1	2	3	4
A	34	24	24	26
B	36	39	32	25
C	20	24	32	17
D	22	27	29	19

c) (0,5 punto) Representar el árbol de estados después de haber expandido los primeros 3 estados (incluyendo el estado inicial)

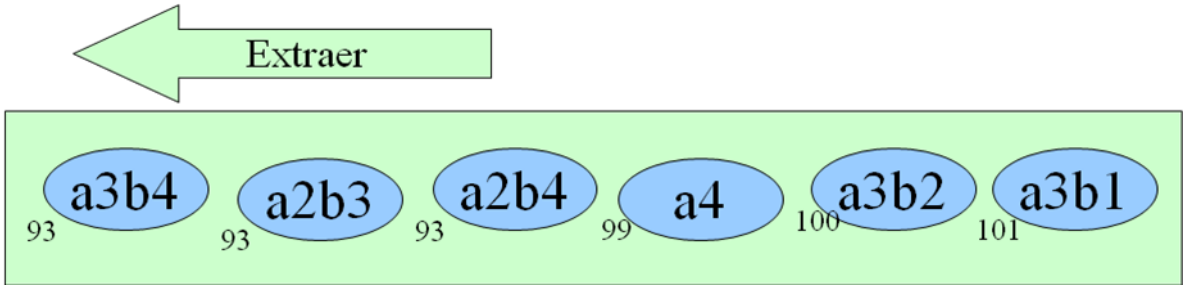
Se trata de desarrollar el árbol de estados en anchura, es decir, precisamente el verbo expandir hace referencia a que se desarrollan a la vez todos los hijos de un nodo dado. Se pide expandir los 3 primeros estados (incluyendo el inicial), para seleccionar el otro estado que expandimos hay que fijarse en el heurístico de ramificación y buscamos el *menor* valor, independientemente de en qué nivel del árbol se encuentre. Además de esto, hay que tener en cuenta el heurístico de poda que eliminará los estados que igualen o superen la cota inicial.



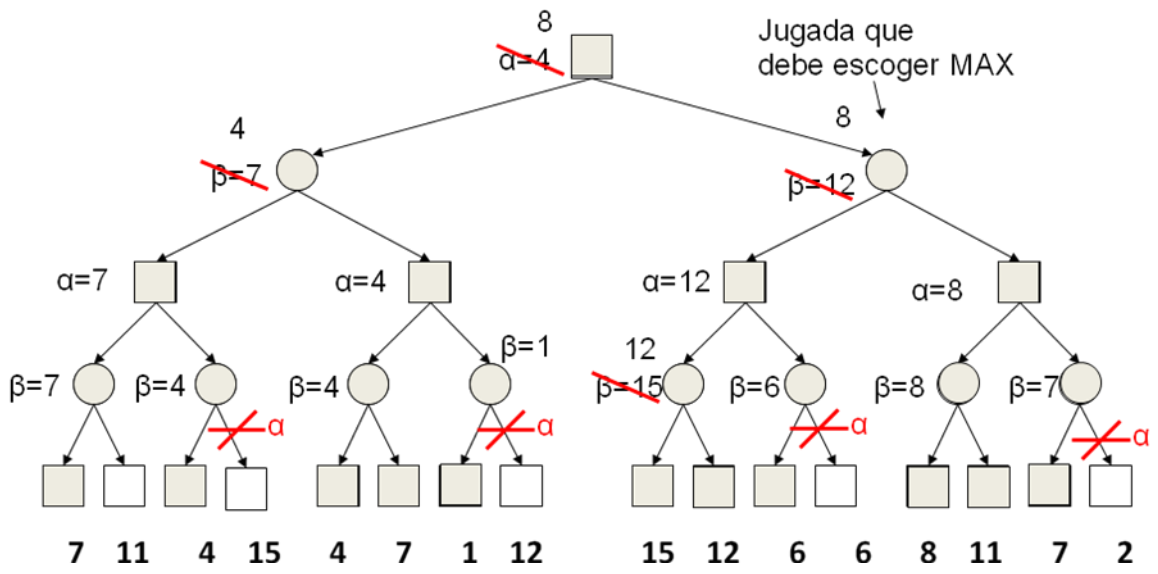
Este es el árbol desarrollado por agentes, en cada nivel se prueban las asignaciones a un agente; pero también es válido el árbol desarrollado por tareas, lógicamente este árbol se desarrollará de otra forma porque tendrá estados distintos.

d) (0,5 puntos) Representar de forma ordenada los estados que quedan en la cola de prioridad en la situación descrita en el punto c).

La cola de prioridad es una estructura de datos ordenada en la que se van introduciendo los nodos pendientes de expandir, por tanto nunca aparecerán en la cola los estados no válidos, los estados podados ni los estados solución.



7. (1,25 puntos) Desarrollar la poda α-β para conocer que jugada debe realizar el jugador MAX, sobre el siguiente árbol:



a) **Sombrear los nodos que haya que desarrollar**
b) **Escribir las cotas α y β,**
c) **Marcar los cortes e indicar si son de tipo α o β,**
d) **Por último, indicar que jugada debe elegir MAX para situarse en la mejor posición posible.**

Notas: El jugador que realiza el primer movimiento en el árbol es MAX. Los nodos del árbol se desarrollan de izquierda a derecha.