Examen de Teoría de la Programación

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

Final Febrero – Curso 2009-2010 26 de enero de 2010

DNI _____ Nombre _____ Apellidos _____

Titulación: ☐ Gestión ☐ Sistemas

3. (1 punto) Sea el algoritmo de ordenación *quicksort*, cuyo código se muestra a continuación:

```
public void quicksort(int[] a, int izq, int der) {
    int i = izq;
    int j = der;
    int pivote = a[ (izq + der) / 2];
    do {
      while (a[i] < pivote) {
        i++;
      }
      while (a[j] > pivote) {
        j--;
      }
      if (i <= j) {
        int aux = a[i];
        a[i] = a[j];
        a[j] = aux;
        i++;
        j--;
      }
    } while (i <= j);
    if (izq < j) {
      quicksort(a, izq, j);
    }
    if (i < der) {
      quicksort(a, i, der);
    }
  }
```

La complejidad de este algoritmo está en función de la elección del *pivote*. En este caso, como se aprecia, se ha escogido el elemento central del vector como *pivote*. Con este planteamiento el caso peor se daría cuando los elementos son todos iguales, o cuando el vector está inicialmente ordenado en orden creciente o decreciente. Calcular y justificar el caso mejor y peor de la complejidad de este algoritmo basándose en las tablas vistas en Divide y Vencerás.

- Caso mejor: todas las particiones son de tamaño similar, n/2.
  - división
  - a=2, b=2, K=1
  - $O(n^K \cdot \log n)$ si $a= b^K$ → **$O(n \log n)$**
- Caso peor: cuando el vector está ya ordenado. Una partición tiene tamaño 0 y la otra n-1.
  - sustracción
  - a=1, b=1, K=1
  - $O(n^{K+1})$ si a=1 → **$O(n^2)$**

4. (1,5 puntos) Antonio se pasa todo el día escuchando música y quiere cargar en el MP3 de su coche, que tiene una capacidad limitada, la mejor selección posible de canciones. Tiene que realizar este recopilatorio de las 2000 canciones que tiene disponibles, a las cuales pacientemente les ha ido asignado una puntuación en función de sus gustos y además tiene almacenado su tamaño. En el MP3 no caben todas las canciones, para crear el mejor recopilatorio debe maximizar la puntuación total de las canciones que contiene, teniendo en cuenta que los ficheros de canciones no se pueden partir. Como Antonio tenía que estudiar para diversos exámenes de la convocatoria de febrero, lo ha dejado para última hora y debe utilizar un algoritmo que calcule las canciones que debe incluir de forma inmediata.

a) Justificar por qué debe aplicar un algoritmo voraz para resolver este problema.

b) Explicar el heurístico más adecuado para este caso.

c) ¿Permite este heurístico obtener la solución óptima en cualquier caso? Demostrar la respuesta.

a)

Para elegir un algoritmo voraz para resolver el problema nos fijamos en los siguientes requisitos:

- Se trata de un problema de optimización: maximizar la puntuación total de las canciones del MP3. Este es el tipo de problemas que encajan con los algoritmos voraces.
- En el enunciado del problema deja claro que se **debe primar que el algoritmo resuelva el problema en un tiempo de ejecución lo más corto posible**, ya que dice que debe ser inmediato. Un **algoritmo voraz es más rápido** de ramificación y poda y sobre todo que *backtracking* para calcular el máximo, por tanto, debemos elegirlo para resolver este problema.
- Por último, podemos disponer, como mostraremos más delante de un heurístico que aunque no asegure la solución óptima, si obtiene soluciones cercanas al óptimo.

b)

Debemos calcular para cada canción el **cociente de su puntuación entre su tamaño** y **ordenar las canciones de mayor a menor en base a este cociente**, para ir seleccionándolas en este orden. Este heurístico no asegura la solución óptima para este problema; pero es el más adecuado.

(El código no se pedía pero se proporciona a modo ilustrativo)

```
public Class Cancion {
  private int identificador;
  private float tamaño;
  private int puntuación;
  private float relacion;

  public int getIdentificador() { return identificador }
  public float getTamaño() { … }
  public int getPuntuacion() { … }
   public float getRelacion() { … }
   public void setRelacion(float cociente) { relacion= cociente; }
  }

public Class SeleccionCanciones {
  Cancion[] canciones;   // array con toda la información sobre las canciones
  Cancion[] cd;          // array que almacenará la solución con las canciones escogidas
  int n= 2000;           // número total de canciones

  […]

  public void crearMp3() {
    float capacidadRestante = capacidadMp3;
    float puntuacionTotalMp3 = 0;
    int contador= 0;  // índice del array donde vamos almacenando las canciones del CD

    // Calcula la relación puntuación / tamaño para todas las canciones
    calcularRelacion(canciones);

    // Ordena canciones de mayor a menor por: puntuación / tamaño
    ordenar(canciones);

    // Ya tengo el array de canciones ordenado,
    // Voy metiendo en el MP3 las canciones en el orden que las tengo,
    // hasta que haya una que no quepa en el espacio restante
    for (int indice=0; indice<n && capacidadRestante>0; indice++) {
      if (canciones[indice].getTamaño() <= capacidadRestante)
      {
        cd[contador++]= cancion[indice];

        puntuacionTotalMp3 += canciones[indice].getPuntuación;
        capacidadRestante-= canciones[indice].getTamaño;
      }
    }

  }
```

c)

Este heurístico **no siempre proporciona la solución óptima**.

Vamos a demostrarlo con un **contraejemplo**:

Supongamos que tenemos lo siguientes datos: Número de canciones: n=3, Tamaño del MP3: T=10

Canción	1	2	3
t_i	6	5	5
p_i	8	5	5
p_i / t_i	1,33	1	1

Valor conseguido con el heurístico: 8
Valor que se pueden conseguir cumpliendo las condiciones: 10
El heurístico no proporciona el valor óptimo.

5. (1,75 puntos) El problema de la asignación de tareas consiste en asignar *n* tareas a *n* agentes, de tal forma que cada agente realiza una única tarea y cada tarea sólo puede ser realizada por un único agente. A cada agente le cuesta realizar unas tareas más que otras, así que cada pareja *(agente, tarea)* tendremos asociado un coste. Esto se representa en una matriz. El objetivo del problema es buscar la asignación que minimice el coste total de realizar todas las tareas.

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 Main
{
    private int costes[][],  // almacena lo que le cuesta a cada agente realizar una tarea
            almacenEstados[],// estado actual del problema
            mejorSolucion[]; // contiene la mejor solución en cada momento
    private boolean tareasAsignadas[];// tareas ya asignadas (las que están a true)

    private int n, // Tamaño del problema.
            coste, // Almacena el coste acumulado en el cálculo de una solución.
            mejorCoste; // El mejor coste hasta el momento de una solución.

    […]

    public void ensayar(int agente)
    {
        for (int tarea = 0; tarea < n; tarea++)
        {
            if (!tareasAsignadas[tarea])
            {
                almacenEstados[agente] = tarea;
                coste += costes[tarea][agente];
                tareasAsignadas[tarea] = true;


                if (agente < (n - 1))   // no es solución
                {
                    ensayar(agente + 1);
                 }
                else         // es una solución posible
                    if (coste < mejorCoste)
                    {
                        mejorCoste = coste;
                        System.arraycopy(almacenEstados, 0,
mejorSolucion, 0, n);
                    }

                coste -= costes[tarea][agente];
                tareasAsignadas[tarea] = false;
            }
        }
     } // Fin método
```

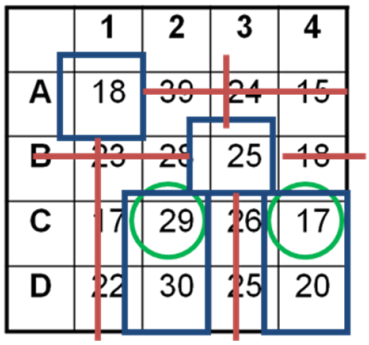
6. (1,5 puntos) Sea el problema de la asignación de tareas a agentes visto en el ejercicio 5. Se dispone de una matriz de costes de ejecución de una tarea j por un agente i.
Dados 4 agentes: a..d y 4 tareas: 1..4. Y la siguiente matriz de costes:

	1	2	3	4
A	18	39	24	15
B	23	28	25	18
C	17	29	26	17
D	22	30	25	20

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

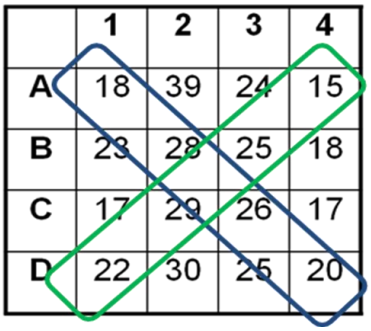
a) (0,25 puntos) Explicar cómo se calcula el heurístico de ramificación para el estado donde asignamos la tarea 3 al agente b, cuando ya tenemos asignada la tarea 1 al agente a.

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 18+25+29+17=89.



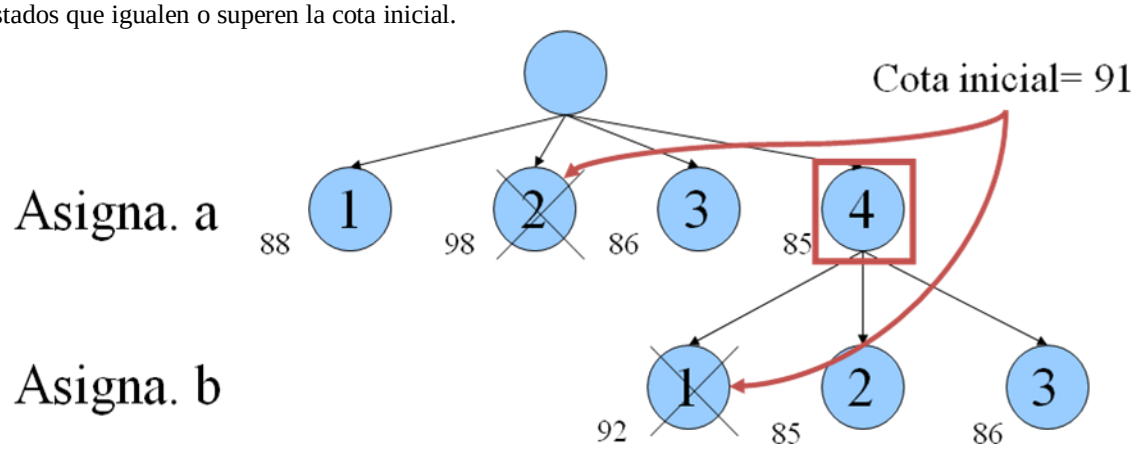
b) (0,25 puntos) Explicar cómo se calcula la cota inicial de poda y razonar cuándo se produce el cambio de esta cota.

Inicialmente, es la menor de la sumas de las dos diagonales de la matriz de costes.
Valor cota inicial: min(91,92)=91.
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.



c) (0,5 punto) Representar el árbol de estados después de haber expandido los primeros 2 estados (incluyendo el estado inicial)

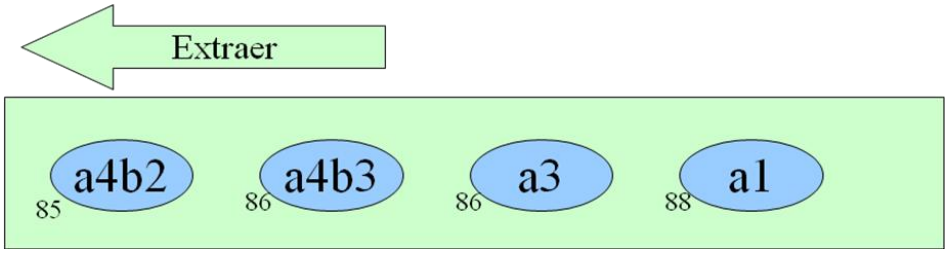
Se trata de desarrollar el árbol de estados en anchura, es decir, precisamente el verbo expandir hace referencia a que se desarrollan a la vez todos los hijos de un nodo dado. Se pide expandir los 2 primeros estados (incluyendo el inicial), para seleccionar el otro estado que expandimos hay que fijarse en el heurístico de ramificación y buscamos el *menor* valor. Además de esto, hay que tener en cuenta el heurístico de poda que eliminará los estados que igualen o superen la cota inicial.



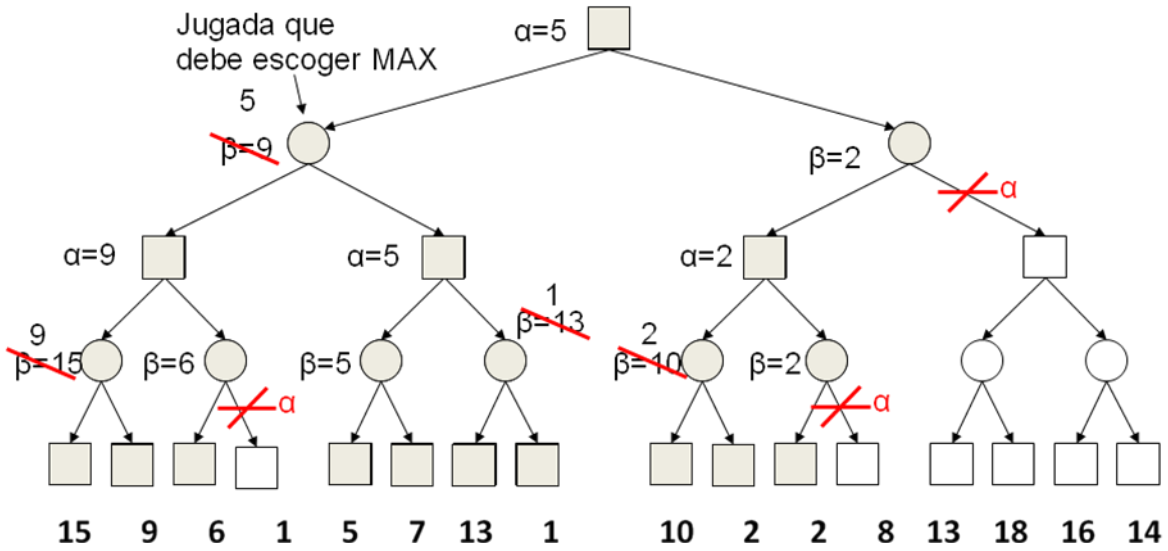
Este es el árbol desarrollado por agentes, en cada nivel se prueban las asignaciones a un agente; pero también es válido el árbol desarrollado por tareas, lógicamente este árbol se desarrollará de otra forma porque tendrá estados distintos.

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.

Ejemplo de indicaciones: