Examen de Teoría de la Programación

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

Final Septiembre – Curso 2006-2007 10 de septiembre de 2007

DNI _____ Nombre _____ Apellidos _____

Titulación: ☐ Gestión ☐ Sistemas

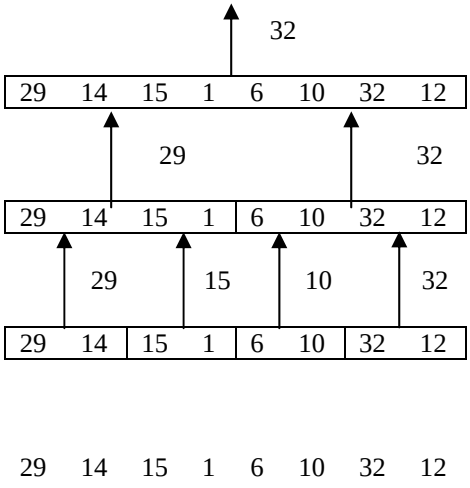
3. (1,5 puntos) Se debe resolver ecuaciones matemáticas de n incógnitas. A partir de un resultado se generaran todas las posibles soluciones, teniendo en cuenta que las incógnitas sólo pueden tomar valores enteros entre 0 y 9. Se considera previamente introducido el número de incógnitas de que consta la ecuación, las operaciones de la misma y el resultado. Ejemplo de ecuación con 5 incógnitas y resultado igual a 23: $x+y*z/a-b=23$

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

```
import java.io.*;

public class Ecuaciones {
        private int enteros[];          /** Valor de las incógnitas */
        private char operaciones[];     /** Signos de las operación */
        int resultado = 0;      /** Resultado que debe dar la ecuación */

        public void backtracking(____int posicion _______) {
                for (__int i = 0; i < 10; i++_) {
                        if (___posicion < enteros.length ___) {
                                enteros[posicion] = i;
                                if (probarSolucion()==resultado) {
                                        imprimirEcuacionResuelta();
                                } else {
                                        backtracking(_posicion+1_);
                                }
                        }
                }
        }

        /** Imprime la ecuación con sus incognitas resueltas   */
        public void imprimirEcuacionResuelta(){ … }

        /** Resuelve ecuación utilizando los valores de las incognitas */
        public int probarSolucion() { … }

        /** Lee por teclado todos los datos de entrada  */
        public void leerDatos() {
                // Lee y almacena resultado de la ecuación
                // Almacena los signos indicativos de la operación
                // Se crea array para almacenar el valor de las incognitas
        }

        public static void main(String[] args) {
                Ecuaciones e = new Ecuaciones();
                e.leerDatos();
                e.backtracking(¿?);
        }
}
```

4. (1,5 puntos) Tenemos un conjunto *S* de *n* números enteros. Queremos buscar el máximo de este conjunto utilizando la técnica de Divide y vencerás.

a) (1 punto) Dado el siguiente conjunto de 8 números. Representar gráficamente (utilizando los números que se representan abajo) las divisiones que se realizan en el vector en el esquema recursivo de división (rodeando con una línea los números que quedan en cada división) y el resultado que devolvería cada llamada recursiva (con una flecha hacia el vector anterior).



29 14 15 1 6 10 32 12

b) (0,5 puntos) ¿Qué complejidad tiene este algoritmo? Justifica el resultado.

DV con división
a= 2
b= 2
K= 0
a= 2 > $b^K=2^0=1$ → $O(n^{\log_b a})=O(n)$

5. (0,5 puntos) Dadas las siguientes funciones razonar a través de su complejidad temporal cuál es la mejor técnica para resolverlas: Divide y Vencerás (DV) o Programación dinámica (PD).

a) Complejidad temporal para DV y PD. ¿Cuál es la mejor técnica?

$$H(x,y)=\begin{cases} 1 \text{ si } x=0 \text{ y } y>0 \\ 0 \text{ si } x>0 \text{ y } y=0 \\ a*H(x-1,y)+b*H(x,y-1) \text{ si } x>0 \text{ y } y>0 \end{cases}$$

DV con sustracción
a= 2
b= 2
K= 0
a>1 → $O(a^{n/b})=O(2^n)$
Prog. Dinámica
Bucle que recorre la tabla $O(n^2)$
La mejor técnica de **Programación dinámica**.

b) Complejidad temporal para DV y PD. ¿Cuál es la mejor técnica?

$$G(x)=\begin{cases} 0 \text{ si } x=0 \\ 1 \text{ si } x=1 \\ G(x-1)+G(x-2) \text{ si } x>1 \end{cases}$$

DV con sustracción; pero la constante que decrementa el tamaño del problema no es igual en todas las llamadas (como en fibonacci), no podemos emplear directamente una ecuación para calcular su complejidad. Podemos acotarla como en fibonacci.
DV → $O(2^{n/2}) \le O(G) \le O(2^n)$
Programación dinámica → $O(n)$
La mejor técnica de **Programación dinámica**.

6. (2 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.

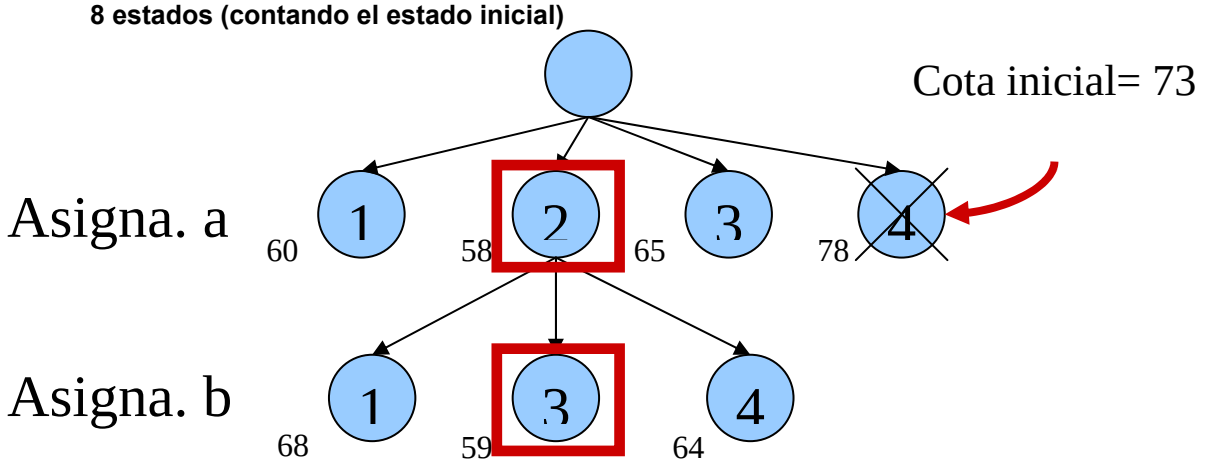
a) **(0,25 puntos) Explicar como se calcula el heurístico de ramificación para el estado donde asignamos la tarea 3 al agente b, cuando ya tenemos asignada la tarea 1 al agente a.**

Heurístico de ramificación:
La suma de lo ya asignado en cada estado más el caso mejor (mínimo de columnas) de lo que queda por asignar.
Al estado a→1, b→3, le corresponde un coste de 11+13+14+23=61.

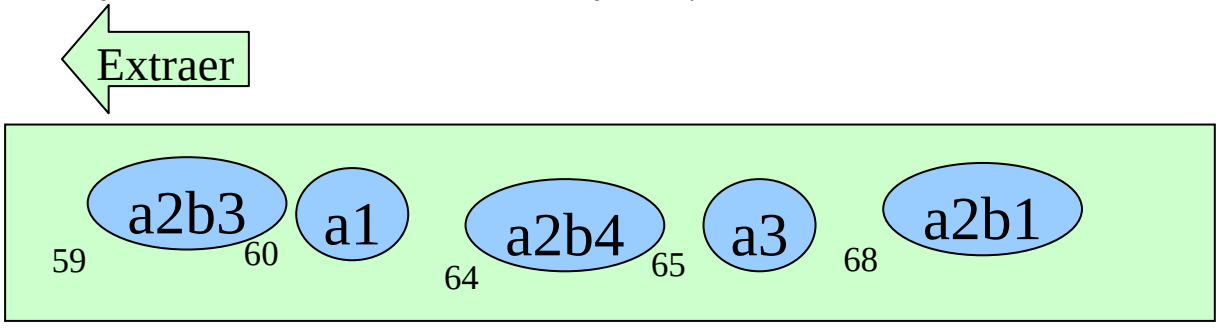
b) **(0,25 puntos) Explicar como se calcula la cota inicial de poda y razonar cuándo se produce el cambio de esta cota.**

Inicialmente, la cota de poda es la menor de la sumas de las dos diagonales de la matriz de costes.
Valor cota inicial: min(73,87)=73.
Cuando en el desarrollo de los estados del árbol, lleguemos a una solución válida cuyo valor sea mejor que el de la cota actual, se cambiará la cota a este nuevo valor.

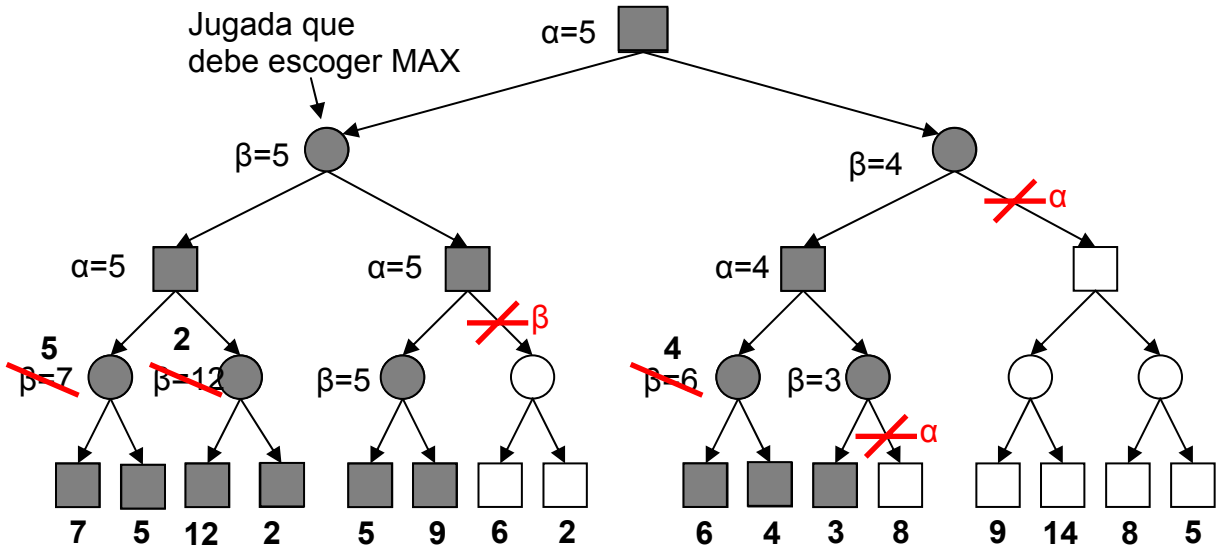
c) **(1 punto) Representar el árbol de estados después de haber desarrollado los primeros 8 estados (contando el estado inicial)**



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).**



7. (1,5 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: