Examen de Teoría de la Programación

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

Final Febrero – Curso 2006-2007 12 de febrero de 2007

DNI ____________ Nombre ______________ Apellidos ______________________________
Titulación: ☐ Gestión ☐ Sistemas
Grupo al que asistes en teoría: ______________________________

3. (1,00 puntos) El siguiente método es una implementación de la búsqueda binaria siguiendo la técnica divide y vencerás:

```
public int buscar(double x, double a[], int inicio, int fin)
{
  if (inicio<=fin)
  {
    int m= inicio + ((fin-inicio)/2);

    if (a[m]==x)
      return m;
    else
      if (x<a[m])
        return (buscar(x, a, inicio, m-1));
      else
        return (buscar(x, a, m+1, fin));
  }
}
```

a) En esta técnica, se diferencian tres pasos: descomposición del problema, resolución de los problemas elementales, combinación de los resultados para obtener la solución al problema inicial. Especificar qué operaciones del método se corresponden con cada uno de estos pasos.

Descomposición

El problema se divide en dos subproblemas de la mitad del tamaño, de los que realmente sólo se resuelve uno.

```
int m= inicio + ((fin-inicio)/2);

if (x<a[m])
        return (busqueda(x, a, inicio, m-1));
      else
        return (busqueda(x, a, m+1, fin));
```

Resolución subproblema

Problema elemental, vector de una posición → encontramos elemento y devolvemos índice, no encontramos elemento.

```
    if (a[m]==x)
      return m;
```

Combinación

No existe combinación propiamente dicha para este problema, simplemente se devuelve el resultado anterior.

b) Cuál es la complejidad de esta solución por Divide y Vencerás. Justifica el resultado.

DV Div: a=1, b=2, K=0
$O(n^K \cdot \log n)$ si $a= b^K$
→ $O(\log n)$

4. (1,25 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.

La función que nos proporciona la solución al problema es la siguiente:

$$V(i,j) = \begin{cases} -\infty \text{ si } j < 0 \\ 0 \;\; si \;\; i = 0 \text{ y } j \geq 0 \\ \max(V(i\text{-}1,j), V(i\text{-}1,j\text{-}w_i) + v_i) \text{ en otro caso} \end{cases}$$

Donde i, representa a los objetos y j, representa el peso de la mochila.
Utilizaremos la técnica de Programación Dinámica para obtener la solución óptima al problema.

a) **Representar la tabla necesaria para almacenar los valores intermedios (sin rellenar). Sabiendo que tenemos tres objetos y el peso máximo que puede llevar la mochila es 6.**

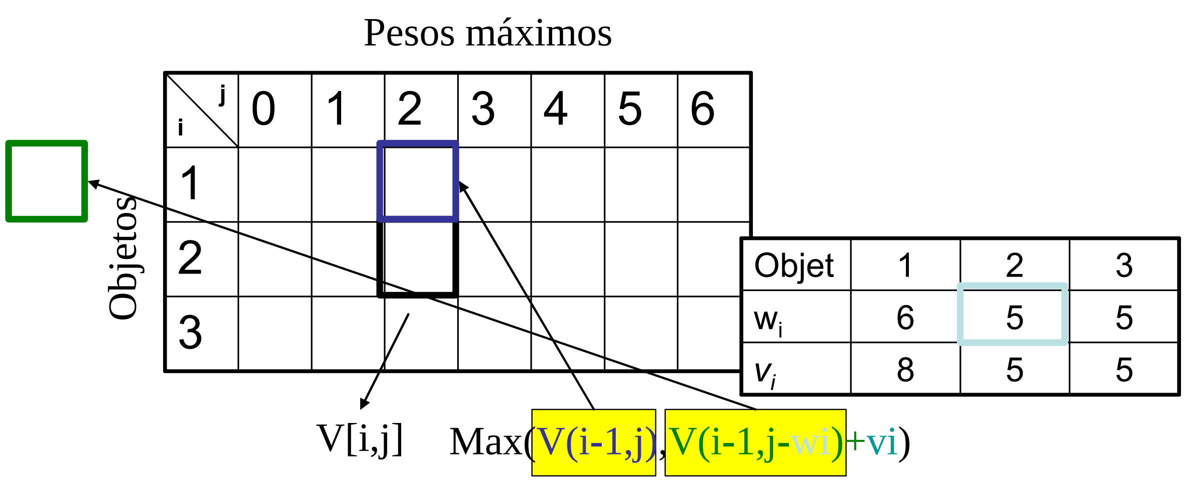
i \ j	0	1	2	3	4	5	6
1							
2							
3							

i, objetos
j, pesos máximos de la mochila

b) **Explicar que valores podemos rellenar de forma directa en la tabla.**

En la tabla no incluimos ninguno de los valores que podríamos rellenar de forma directa. Por un lado, no incluimos la fila i= 0 (aunque podría hacerse) y por eso no hay valores 0 y por otro lado no incluimos las columnas con j<0. Ninguno de estos valores va a tener que ser devuelto como resultado de la función y por tanto, los asignamos a la función cuando uno de sus parámetros toma uno de estos valores.

c) **Buscar un patrón de dependencia de una celda cualquiera, es decir, marcar las celdas que son necesarias para calcular una celda dada. Se pueden plantear las hipótesis que sean necesarias sobre el valor y peso de los objetos.**



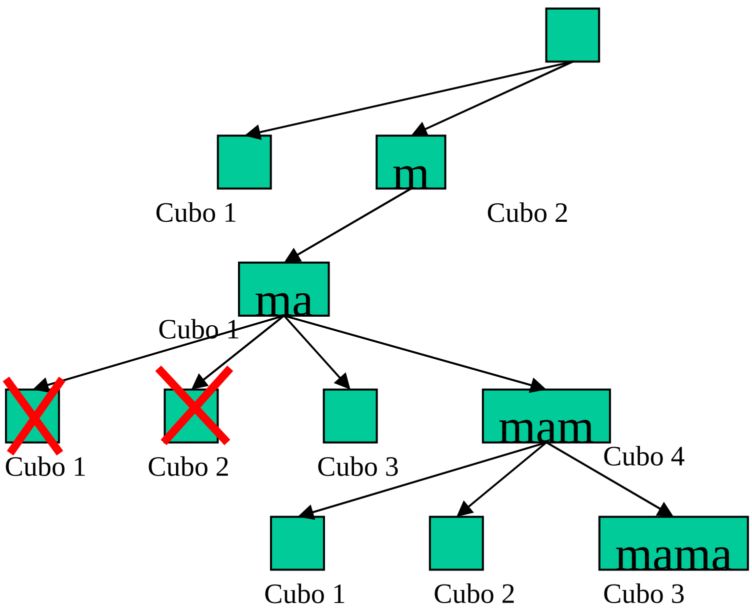
d) **(0,5 puntos) Escribir el código Java del método principal que implementaría la solución a este problema.**

```
public int mochilaPd(int a, int b)
{
	int i, j;

	for (j= 0; j<=W; j++)
		for (i= 1; i<=objetos.length; i++)
tablaV[i][j]= Math.max(funcionV(i-1,j),funcionV(i-1,j-objetos[i].peso)+objetos[i].valor);

	return tablaV[a][b];
}
```

5. (1,75 puntos) Se desea formar palabras de n (o menos) letras utilizando N cubos. Para ello disponemos de un número limitado de cubos (n) con una letra diferente en cada una de sus caras y una palabra objetivo. Mediante la combinación de las caras de los cubos (sin repetir ninguno), se debe obtener la palabra objetivo. Si la combinación no fuera posible a partir de los cubos de los que se dispone, se descarta la posibilidad de formar la palabra, se informaría que es imposible hacerlo.

a) **(0,5 puntos) Dibujar el árbol (sólo los estados desarrollados) que desarrollará la técnica de backtracking para buscar una combinación de cubos que proporcione la palabra "mama" con las siguientes letras en los cubos.**

```
private final char[] cubo1= { 'a', 'b', 'c', 'f', 'g', 'z' };
private final char[] cubo2 = { 'b', 'a', 'm', 'h', 's', 't'};
private final char[] cubo3 = { 'c', 'b', 'a', 'j', 'v', 'w'};
private final char[] cubo4 = { 'c', 'm', 'a', 'p', 'f', 'g'};
```



b) **(1 punto) Escribir el método que realiza el backtracking, que llamaremos *ensayar*.**

```
public boolean ensayar(int posicion) {
  boolean haySolucion= false;
  for (int i = 0; i < listaCubos.length && !haySolucion; i++) {
    //Seleccionamos un estado accesible desde el actual
    if (!listaCubos[i].getUsado()) {
      for (int j = 0; j < listaCubos[i].getTamanyo(); j++) {
        //Si el estado es posible
        if (listaCubos[i].getLetra(j) == palabra.charAt(posicion)) {

          //Anotamos el nuevo estado
          palabraEncontrada.append(palabra.charAt(posicion));
          listaCubos[i].setUsado(true);
          recorridoActual[posicion] = i;

          //Si no es solución final
          if (!palabra.equalsIgnoreCase(palabraEncontrada.toString())) {
            ensayar(posicion + 1);
```

```
          }
          //Si es solución final, escribir la solución
          else {
            haySolucion= true;
          }

        if (!haySolucion) {
          //Finalmente, se borra la anotación del estado
          palabraEncontrada.delete(posicion, posicion + 1);
          listaCubos[i].setUsado(false);
          recorridoActual[posicion] = -1;
          }

        }
      }

    }
  }
  return haySolucion;

}
```

c) **(0,25 puntos) Escribir el método main() que hace la llamada a *ensayar*.**

```
public static void main(String args[]) {
          Principal p = new Principal();
          if (p.probarNuevoEstado(0)) {
            System.out.println("Encontrada solución!!!");
             for (int l = 0; l < p.getRecorridoActual().length; l++) {
               System.out.print(p.getRecorridoActual[l] + ", ");
            }
          }
          else {
            System.out.println("No hay solución");
          }

}
```

(Problema expuesto en el grupo A)

6. (1,0 puntos) Se dispone de n ficheros de los que se conoce: una prioridad de valores entre 1 y 10 (correspondiendo 1 a la prioridad más baja y 10 a la más alta) y un tamaño. Se trata de maximizar la suma de las prioridades de los ficheros almacenados en un disco de una capacidad determinada, teniendo en cuenta que los ficheros no se pueden partir. Se ha de utilizar un algoritmo que calcule de manera inmediata los ficheros que se deben de incluir.

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

Cuando nos piden que se ejecute "de forma inmediata", es necesaria la técnica que nos asegura una ejecución rápida: devorador.

b) **Explicar el heurístico más adecuado para este caso.**

Ordenar los ficheros por el cociente prioridad / tamaño y coger el mayor no introducido en el disco; así hasta que ya no quepan más ficheros en el disco.

c) **Demostrar si el heurístico propuesto es óptimo o no.**

Nº ficheros= 3, Capacidad del disco= 10

Objeto	1	2	3
Tamaño	6	5	5
Prioridad	8	5	5
Cociente	1,33	1	1

Heurístico toma el fichero 1, obteniendo una prioridad total de 8. Sin embargo, se podrían meter en el disco los ficheros 2 y 3 con una suma de prioridades de 10.

Por tanto, el heurístico no es óptimo.

(Problema expuesto en el grupo C)

7. (1,0 puntos) Técnica de Ramificación y poda.

a) **Especifica las ventajas que puede tener el recorrido en anchura sobre Backtracking.**

- Si estamos en el caso de buscar sólo la primera solución y hay estados solución cerca de la raíz.
- En backtracking (profundidad) es posible que nos metamos por una rama con muchos nodos que no llegue a ninguna solución o incluso que no tenga fin.

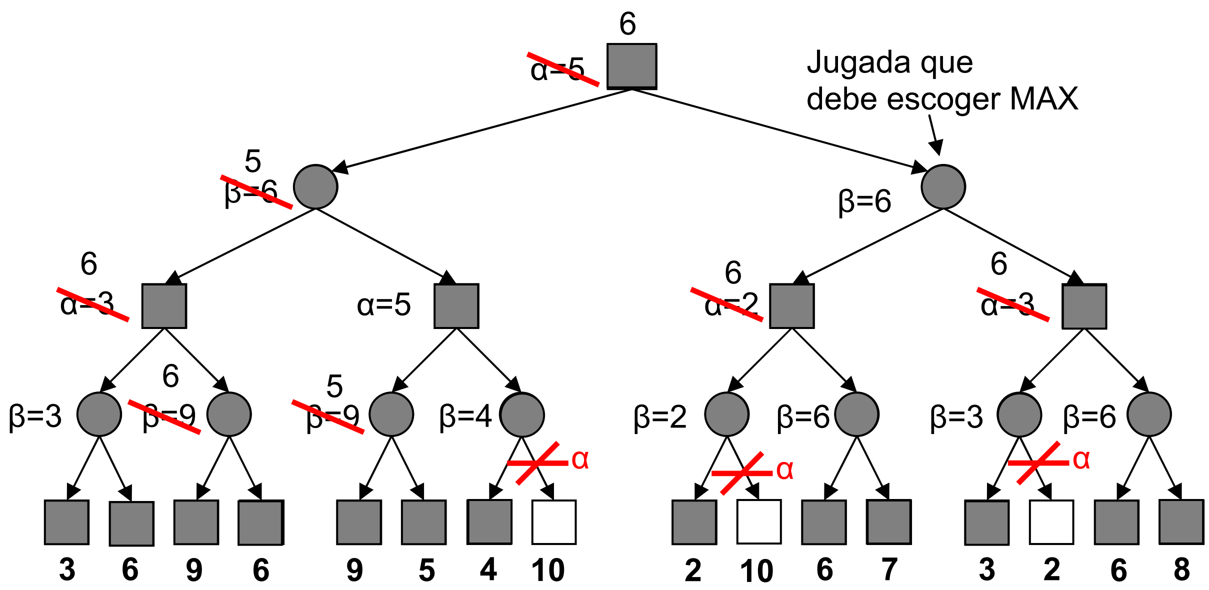
b) **Qué diferencias de la implementación implica el añadir un heurístico de ramificación al recorrido en anchura puro.**

El esquema general es igual al visto para la exploración en anchura, cambiando la cola FIFO por una cola de prioridad, para almacenar los estados “activos” por desarrollar.

c) **¿Es importante el heurístico de poda a la hora de buscar la primera solución a un problema? (Justifica la respuesta)**

No, para buscar la primera solución es más importante el heurístico de ramificación que nos guía hasta la primera solución. Cómo paramos de explorar el árbol cuando encontramos la primera solución da igual que otros estados estén podados o por desarrollar.

8. (1,0 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: