Examen de Teoría de la Programación

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

Final Junio – Curso 2008-2009 10 de junio de 2009

DNI ______________ Nombre ________________ Apellidos ___________________________________

Titulación: ☐ Gestión ☐ Sistemas

3. (1,75 puntos) Sea un conjunto de letras de tamaño *n*. El problema consiste en obtener todas las palabras posibles que podamos (combinaciones de letras sin repetir ninguna) con un tamaño *m*, siendo *m*<=*n*. Ejemplo: Con el conjunto de letras ABCDE queremos construir todas las palabras posibles con 3 letras, obtendríamos la siguiente respuesta: ABC, ABD... así hasta EDC. El problema se quiere resolver utilizando la técnica de Backtracking.

- **a) (0,5 puntos) Dibujar el árbol completo que genera la técnica del backtraking para el ejemplo propuesto.**
- **b) (1,25 puntos) Completar el código Java para dar solución a este problema (escribirlo en los recuadros preparados a tal efecto).**

```java
import java.io.*;

public class Palabras {
    private char[] arrayLetras;   // conjunto de letras de tamaño n
    int numeroLetras;             // tamaño m de las combinaciones de letras
    private boolean[] estaElegida;// permite comprobar si una letra ya está elegida
    char[] solucion;              // almacena la solucion

    public Palabras(char[] arrayLetras, int numeroLetras) {
         this.arrayLetras= arrayLetras;
         this.numeroLetras= numeroLetras;
         estaElegida = new boolean[arrayLetras.length];
         solucion = new char[numeroLetras];
    }

    public void construirPalabras(________int posi________){
         for(____int i=0; i<arrayLetras.length; i++____{
             if(_________!estaElegida[i]___________){
                 estaElegida[i]=true;
                 solucion[posi]=arrayLetras[i];
                 if(____posi<numeroLetras-1______)
                  construirPalabras(_______posi+1_________);
                 else{
                     imprimirArray(solucion);
                 }
             estaElegida[i]=false;
             }
         }
    }

    private void imprimirArray(char[] array){ … }

    public static void main(String[] args) {
        char[] arrayLetras= null;
        int numeroLetras= 0;

         // Lectura del conjunto de letras de tamaño n
         …
         // Lectura del tamaño m de las combinaciones de letras
         …
        Palabras palabras = new Palabras(arrayLetras,numeroLetras);
        palabras.construirPalabras(¿?);
    }
}
```

4. (0,5 puntos) Sea la siguiente especificación de una función:

- **{Q ≡ b≠0}**
- **fun *funcion*(*a, b: entero*) dev (*q, r: entero*)**

- **{R ≡ (a=b*q+r) ∧ (r<b) ∧ (r≥0)}**

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 ResultadoDivEntera division(int a, int b) throws InvalidArgumentException
  {
    // precondición
    if (b==0)
      throw new IllegalArgumentException();

    ResultadoDivEntera res;  //
    int q, r;        // Cociente y resto

    // operaciones ...

    // postcondición
    assert ((a==b*q+r)  && (r<b) && (r>=0));
    return res;
  }
```

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) Cuál sería la forma de ordenar los objetos para ir metiéndolos en la mochila y alcanzar la solución óptima.

- Calcular valor por unidad de peso (**valor / peso**) de cada objeto.
- Ordenar los objetos de mayor a menos por este valor

b) (0,5 puntos) Cuál sería la condición de parada del bucle para ir seleccionando los objetos.

```
pesoActual < pesoMaximo && i<numObjetos
```

Siendo: pesoMaximo, el límite de peso que puede transportar la mochila; pesoActual, el peso acumulado de los objetos que introducimos en la mochila, teniendo en cuenta que hemos introducido el último objeto introducido.

c) (0,5 puntos) Qué sería lo último que meteríamos en la mochila.

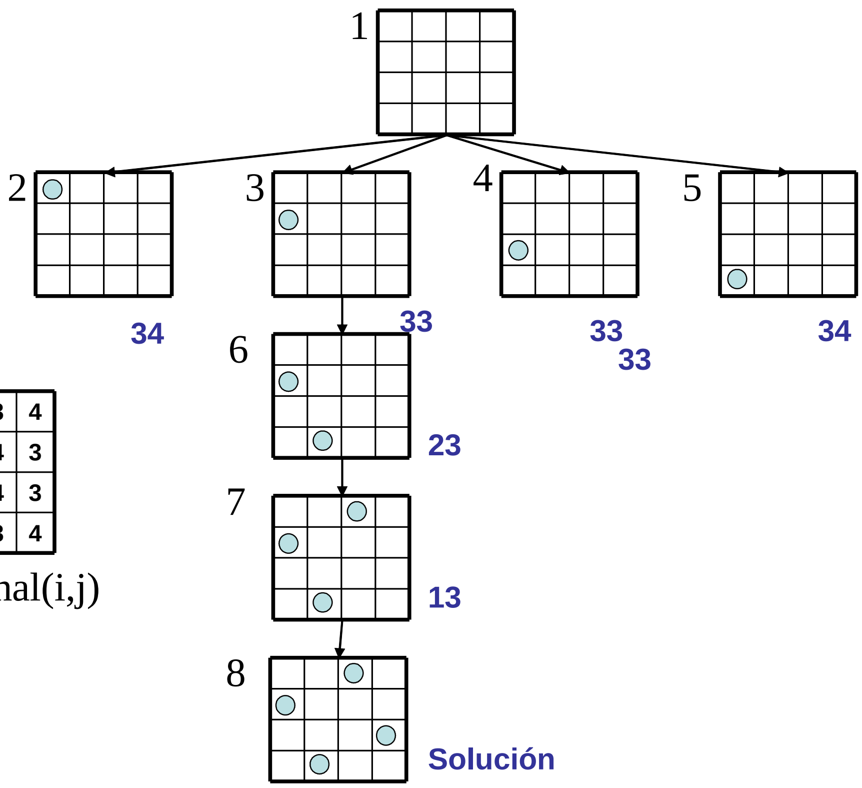
Se introduce el objeto que corresponde en función del orden valor/peso, teniendo en cuenta que si su peso supera la capacidad restante de la mochila, se fragmenta en la cantidad exacta para ocupar el peso restante de la mochila.

Pseudocódigo del algoritmo devorador completo:

```
  // Previamente se habrán ordenado los objetos según el heurístico
  public float[] algoritmo() {
    int i = 0;
    float pesoActual = 0;
    for (int j = 0; j < numObjetos; j++) {
      pctSeleccionados[j] = 0; // 0% de cada objeto
    }
    do {
      i = heuristicoObtenerObjeto();
      if (pesoActual + pesos[i] <= pesoMaximo) {
        pctSeleccionados[i] = 1; // se coge el objeto entero (100%)
        pesoActual += pesos[i];
      }
      else {
        pctSeleccionados[i] = ( (pesoMaximo - pesoActual) / pesos[i]);
        pesoActual = pesoMaximo;
      }
    }
    while (pesoActual < pesoMaximo && i<numObjetos);
    return pctSeleccionados;
  }
```

6. (2 puntos) El problema de las n reinas consiste en colocar n reinas en un tablero de ajedrez sin que ninguna reina pueda comer a otra. Pretendemos buscar la primera solución a este problema con un tablero de 4 x 4.
Utilizaremos la técnica de ramificación y poda. El heurístico de ramificación que utilizaremos será: (*n – nº reinas colocadas*) * 10 + *diagonal(i,j) para la última reina colocada*, siendo *diagonal(i,j)* la longitud de la diagonal más larga que pasa por la casilla (i,j).

a) **(0,75 puntos) Dibujar gráficamente el árbol de estados del algoritmo hasta encontrar la primera solución (sólo los estados en los que las reinas no se coman). En cada estado debemos marcar orden en el que se ha desarrollado y el valor del heurístico de ramificación.**



4	3	3	4
3	4	4	3
3	4	4	3
4	3	3	4

diagonal(i,j)

b) **(0,5 puntos) Dibujar gráficamente la cola de prioridad, en el momento en que encontramos esa primera solución, representando el orden en el que quedan los estados.**

Compuesta por los estados 4, 2 y 5 en este orden.

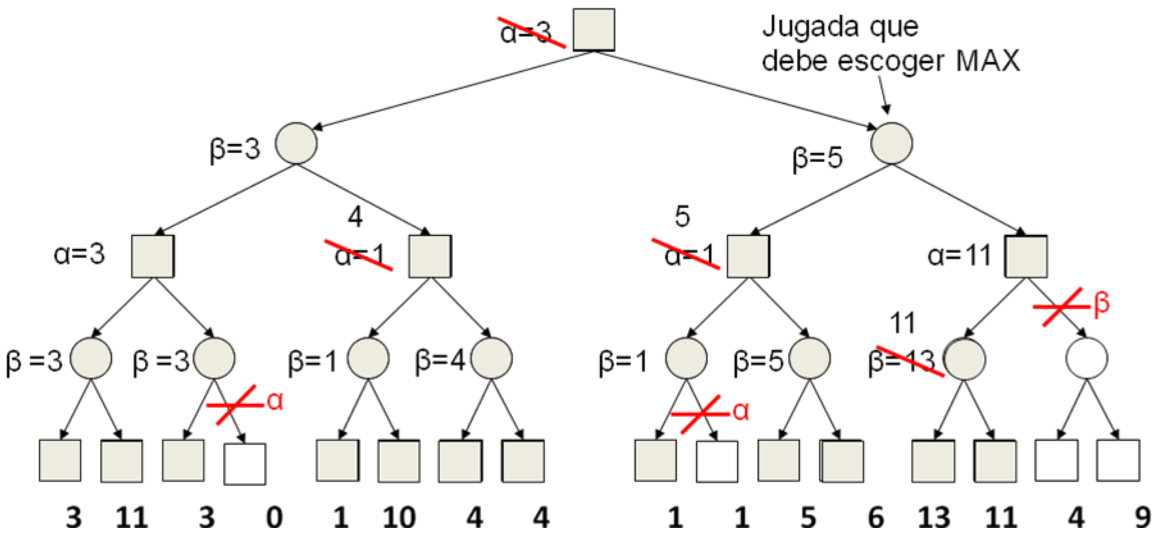
c) **(0,25 puntos) Razonar que técnica es más eficiente para buscar la primera solución a este problema: ramificación y poda (utilizando el heurístico de ramificación dado) o backtracking.**

La técnica de **ramificación y poda** gracias al **heurístico de ramificación** definido permite acercarse más rápidamente que backtracking a la solución final. Cómo ejemplo sólo hay que estudiar el problema propuesto, backtracking empezaría explorando en la rama de la izquierda donde no encuentra solución y luego tiene que recorrer la segunda rama para encontrarla.

d) **(0,25 puntos) ¿Y para buscar todas las soluciones posibles?**

La técnica de ramificación y poda para este problema, no dispone de **ningún heurístico de poda** (estos normalmente están asociados a problemas de optimización y este no es de ese tipo) que **evite el desarrollo de alguna rama del árbol**, por tanto, igual que en backtracking para buscar todas las soluciones tenemos que desarrollar todos los nodos del árbol, por lo que en tiempo de ejecución no mejora el backtracking (incluso el cálculo del heurístico de ramificación lo ralentiza). Si miramos la complejidad espacial el algoritmo de ramificación y poda al tener que guardar todos los nodos no desarrollados en una cola necesita más espacio de almacenamiento. Por tanto, es mejor inclinarse por **bakctracking** en este caso.

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.