Examen de Teoría de la Programación

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

Final Junio – Curso 2006-2007 — 14 de junio de 2007

DNI _____ Nombre _____ Apellidos _____

Titulación: ☐ Gestión ☐ Sistemas

3. (1,5 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 y queremos construir todas las palabras posibles con 3 letras, obtendríamos la siguiente respuesta: ABC, ABD... así hasta EDC.

El problema lo tenemos 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 Palabras {
    private char[] arrayLetras;// conjunto de letras de tamaño n
    int numeroLetras;              // tamaño m de las combinaciones de letras
    private boolean[] estaElegida;
    char[] 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) 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

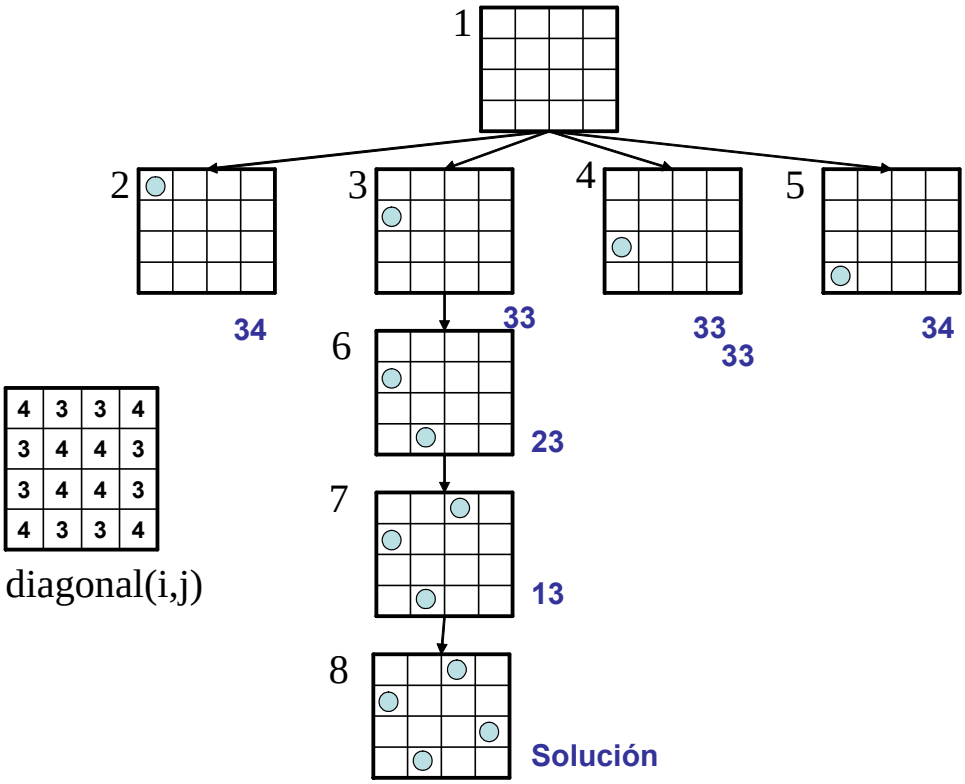
b) (1 punto) Escribir el código Java del método que permita obtener la solución a este problema mediante un algoritmo voraz (escribir exclusivamente el método que realiza el algoritmo voraz suponer ya cargados los arrays necesarios con los datos de los objetos, no realizar ninguna entrada / salida en el método).

```
// 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) (1,5 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.



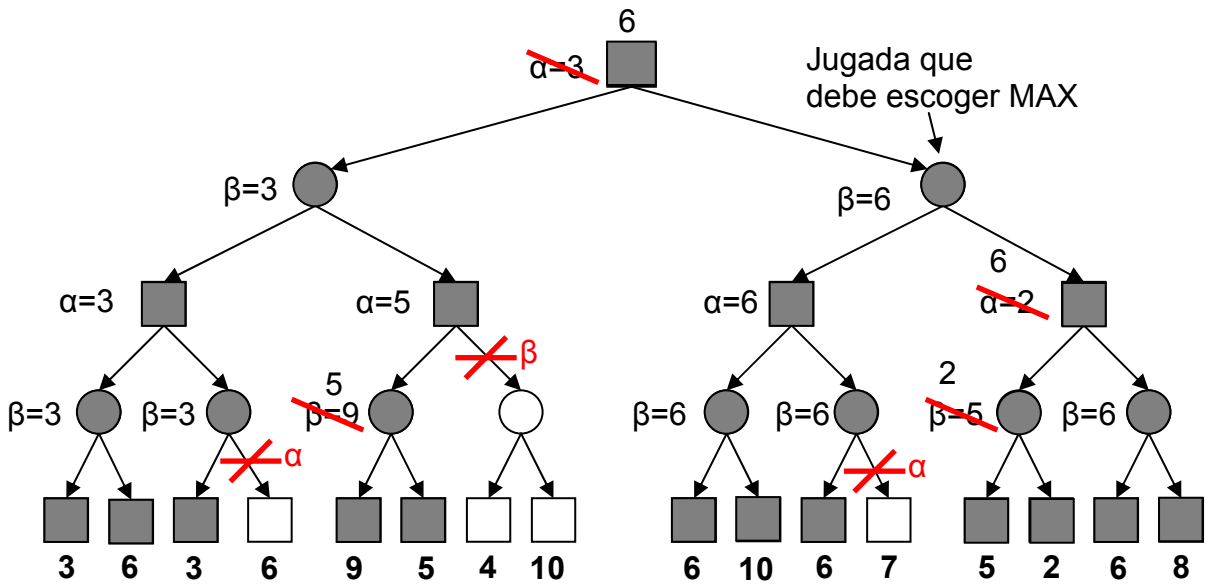
b) (0,25 puntos) Razonar que técnica es más eficiente para buscar la primera solución a este problema: ramificación y poda (utilizando el heurístico de ramificación dado) o backtracking.

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.

c) (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.